

Módulo I – Programação (Cliente e Servidor)

1) Introdução e Visão geral do PostgreSQL - 3

- 1.1) Resumo da História do PostgreSQL
- 1.2) Características Avançadas
- 1.3) Limites do PostgreSQL
- 1.4) Principais Atribuições de um DBA
- 1.5) Suporte ao PostgreSQL no Brasil e Quem Usa
- 1.6) Modelo Relacional e Objeto-Relacional
- 1.7) Instalação do PostgreSQL 8.3 no Windows

2) Utilizando SQL para selecionar, filtrar e agrupar registros - 23

- 2.1) A linguagem SQL
- 2.2) Principais Palavras-Chave e Identificadores
- 2.3) Trabalhando e Manipulando Valores Nulos
- 2.4) Utilizando Comentários
- 2.5) Entendendo os Tipos de Dados
- 2.6) Utilizando Expressões e Constantes
- 2.7) Ocultando Linhas Duplicadas em uma Consulta (DISTINCT)
- 2.8) Limitando o Resultado do Select
- 2.9) Utilizando o Comando Case
- 2.10) Substituindo Valores Nulos para Formatar dados Retornados na Consulta
- 2.11) Uso do Select

3) Criação e Manipulação de Tabelas - 48

- 3.1) Visualizando a estrutura de tabelas criadas
- 3.2) Entendendo as colunas de sistema
- 3.3) Sintaxe de criação de tabelas
- 3.4) Entendendo o comando Alter Table
- 3.5) Alterando tabelas e colunas
- 3.6) Comentários em objetos
- 3.7) Eliminando tabelas

4) Trabalhando com Conjuntos de Dados - 56

5) Consultando dados em múltiplas tabelas - 65

- 5.1) Utilizando Apelidos para as tabelas
- 5.2) Cruzando dados entre tabelas distintas
- 5.3) Entendendo os Tipos de Join disponíveis
- 5.4) Trabalhando com CROSS JOIN
- 5.5) Trabalhando com INNER e OUTER JOINS
- 5.6) Trabalhando com NATURAL JOIN

6) Utilizando Operadores - 96

- 6.1) Introdução aos operadores
- 6.2) Entendendo os Operadores de texto
- 6.3) Entendendo as Expressões regulares
- 6.4) Entendendo os Operadores matemáticos

6.5) Entendendo a importância da Conversão de tipos

7) Reuso de Código: Utilizando Funções - 114

7.1) Introdução às Funções

7.2) Utilizando Funções matemáticas

7.3) Utilizando Funções de data e hora

7.4) Utilizando Funções de Texto (Strings)

7.5) Utilizando Funções de Conversão de Tipos (Casts)

7.6) Utilizando Outras Funções

7.7) Entendendo as Funções de Agregação

8) Entendendo e Utilizando sub-consultas - 126

9) Alterando dados nas tabelas - 133

9.1) Adicionando dados com Insert

9.2) Adicionando dados com Select

9.3) Modificando dados com Update

9.4) Removendo dados com Delete

9.5) Removendo dados com Truncate

10) Filtrando dados nas tabelas - 139

10.1) Filtrando dados com a cláusula where

10.2) Entendendo os operadores Like e Ilike

10.3) Utilizando o Operador Between

10.4) Utilizando o IN

10.5) Cuidados ao fazer Comparações com valores “NULL“

10.6) Utilizando a Cláusula Order By

11) Entendendo a execução das transações no PostgreSQL - 154

12) Entendendo as constraints e a integridade referencial - 178

13) Entendendo a Herança de tabelas - 191

14) Funções em SQL e em PL/pgSQL, Gatilhos e Regras - 203

15) Usando Matrizes/Arrays no PostgreSQL - 292

16) Utilizando o Comando CASE ... WHEN - 305

17) Funções para formatar tipo de dado - 311

18) Trabalhando com pontos e linhas - 318

1) Introdução e Visão geral do PostgreSQL

- 1.1) Resumo da História do PostgreSQL
- 1.2) Características Avançadas
- 1.3) Limites do PostgreSQL
- 1.4) Principais Atribuições de um DBA
- 1.5) Suporte ao PostgreSQL no Brasil e Quem Usa
- 1.6) Modelo Relacional e Objeto-Relacional
- 1.7) Instalação do PostgreSQL 8.3 no Windows

1.1) Resumo da História do PostgreSQL

O PostgreSQL atual é derivado do POSTGRES, escrito pela universidade de Berkeley na Califórnia (EUA). O POSTGRES foi inicialmente patrocinado pela Agência de Projetos de Pesquisa Avançados de Defesa (DARPA), pelo Escritório de Pesquisa sobre Armas (ARO), pela Fundação Nacional de Ciências (NSF) e pelo ESF, Inc.

PostgreSQL é o trabalho coletivo de centenas de desenvolvedores, trabalhando sob vinte-um anos de desenvolvimento que começou na Universidade da Califórnia, Berkeley. Com seu suporte de longa data a funcionalidades de nível corporativo de bancos de dados transacionais e escalabilidade, o PostgreSQL está sendo utilizado por muitas das empresas e agências de governo mais exigentes. O PostgreSQL é distribuído sob a licença BSD, que permite uso e distribuição sem ônus para aplicações comerciais e não-comerciais.

POSTGRES - 1986

POSTGRES 1 - 06/1989

POSTGRES 2 - 06/1990

POSTGRES 3 - 06/1991

O Informix (adquirido pela IBM) foi originado do código do POSTGRES

Em 1994 Andrew Yu and Jolly Chen adiciona um interpretador de SQL substituindo o PostQUEL

Ele muda de nome para Postgres95 e teve seu código completamente reescrito em ANSI C

Postgres95 - 01/05/1995

PostgreSQL 1 – 05/09/1995

Em 1996 muda novamente de nome, agora para PostgreSQL

PostgreSQL 6 - 29/01/1997 (do 1.09 passou para o 6)

PostgreSQL 7 - 08/05/2000

PostgreSQL 8 - 19/01/2005 (primeira versão for Windows)

PostgreSQL 8.1 - 08/11/2005

PostgreSQL 8.2 - 05/12/2006

PostgreSQL 8.3 - 04/02/2008 (versão atual)

Uma nova versão sai aproximadamente a cada ano.

Fontes:

<http://www.postgresql.org/docs/8.3/interactive/release.html> e

<http://www.postgresql.org/docs/8.3/interactive/history.html>

1.2) Características Avançadas do PostgreSQL

- Exporta em XML
- Busca Textual, com TSearch2
- Novos tipos de dados: ENUM e matrizes de tipos compostos
- Suporte a SNMP
- Transações
- Replicação
- Banco de dados objeto-relacional
- Suporte a transações (padrão ACID)
- Lock por registro (row level locking)
- Integridade referencial
- Número ilimitado de linhas e índices em tabelas
- Extensão para GIS (base de dados geo-referenciados)
- Acesso via drivers ODBC e JDBC
- Interface gráfica de gerenciamento
- Uso otimizado de recursos do sistema operacional
- Suporte aos padrões ANSI SQL 92 e 99
- Joins: Implementa todos os tipos de join definidos pelo padrão SQL99: inner join, left, right, full outer join, natural join.
- Triggers, views e stored procedures
- Suporte ao armazenamento de BLOBs (binary large objects)
- Sub-queries e queries definidas na cláusula FROM
- Backup online
- Sofisticado mecanismo de tuning
- Suporte a conexões de banco de dados seguras (criptografia)
- Modelo de segurança para o acesso aos objetos de banco de dados por usuários e grupos de usuários

Fonte: site do Dextra Treinamentos

Lista completa: <http://www.postgresql.org/about/press/features83> e <http://www.postgresql.org/docs/8.3/static/release-8-3.html>

1.3) Limites do PostgreSQL

Tamanho de um Banco de Dados - ilimitado
Tamanho de uma tabela - 64 TB
Quantidade de registros por tabela - ilimitados
Quantidade de campos por tabela - 250 a 1600 (depende do tipo)
Quantidade de índices por tabela - ilimitados
Tamanho máximo de um campo - 1GB para a versão 7.1 e posteriores
Máximo de colunas numa tabela - 1600

Licença

O PostgreSQL usa a [licença BSD](#), que apenas requer que o código fonte sob licença mantenha o seu direito de cópia e a informação da licença. Essa [licença certificada pela OSI](#) é amplamente vista como flexível e amigável a empresas, já que ela não restringe o uso do PostgreSQL com aplicações comerciais e proprietárias. Juntamente com suporte de várias empresas e propriedade pública do código, a licença BSD torna o PostgreSQL muito popular entre empresas que querem embutir o banco de dados nos seus próprios produtos sem temor de taxas, enclausuramento a uma empresa, ou mudanças nos termos da licença.

1.4) Principais Atribuições de um DBA

DBA - DataBase Administrator – Administrador de Banco de Dados

É o profissional responsável pela administração e gerenciamento do banco de dados. Isto normalmente envolve:

- A instalação do próprio (PostgreSQL, SQL Server, Oracle, Saybase, MySQL, etc) em um servidor específico para as bases de dados da organização.
 - Cálculo e dimensionamento de clusters e blocagem para criação/instalação das bases de dados.
 - A criação de usuários e o controle sobre o que cada um pode ou não fazer na base de dados (grants) e toda a segurança relacionada a acessos às bases de dados (pg_hba.conf).
 - A criação de objetos de dados como bancos, esquemas, tabelas, índices, constraints, views, funções, triggers, sequences, etc, seguindo as especificações (scripts de criação) de um analista de sistemas, engenheiro, arquiteto de software ou técnico responsável pelas atividades de modelagem de um projeto de software.
 - Analisar os scripts de criação de objetos, propondo determinadas alterações eventualmente necessárias ao aumento de performance e ganho do banco de dados.
 - Análise periódica das bases de dados considerando aspectos como estimativas e dados de crescimento das bases, performance, estatísticas de utilização, transações, etc.
 - Elaboração (eventual) e garantia (normal) de nomenclaturas e padrões que devem ser seguidos pelos modeladores de software para objetos de dados (nomes de tabelas, atributos, índices, etc).
 - Dimensionamento de hardware e software para instalação de bases de dados da organização ou de um determinado projeto de software.
 - Planejamento, execução e guarda de cópias de segurança das bases de dados, bem como, eventualmente, a execução de atividades necessárias à restauração.
- Eventualmente os DBA's envolvem-se com o desenvolvimento de Stored Procedures, Pckages/Procedures nos bancos de dados. Mas isto não é normal.
- Fonte: Espia em <http://forum.wmonline.com.br/index.php?showtopic=25911>

1.5) Suporte ao PostgreSQL no Brasil e quem usa

Empresas que Patrocinam o PostgreSQL:

- InterpriseDB
- Fijitsu
- RedHat
- Skype
- Sun

Lista completa - <http://www.postgresql.org/about/sponsors>

Oferecem Suporte ao PostgreSQL (consultores individuais e empresas):

Lista completa: http://www.postgresql.org/support/professional_support

Mais empresas e consutores que oferecem suporte ao PostgreSQL:

http://www.opensourceexperts.com/Index/index_html/PostgreSQL/index.html

Oferecem Suporte ao PostgreSQL (no Brasil):

- Carlos Smanioto (Bauru)
- dbExperts (SP)
- Dextra (Campinas)
- Flexsolutions Consultores (Goiânia)
- Locadata (Belo Horizonte)
- Mosman Consultoria e Desenvolvimento (São Pedro - SP)
- Studio Server Hospedagem de Sites (Santos)

Detalhes e Lista completa: http://www.postgresql.org/support/professional_support_southamerica

Algumas Empresas que usam o PostgreSQL (internacionais):

- Apple
- BASF
- Cisco
- OMS (Organização Mundial de Saúde)

Algumas Empresas que usam o PostgreSQL (nacionais):

- FAB (Força Aérea Brasileira)
- Prefeitura Municipal de Sobral
- Vivo
- DNOCS

Obs.: Comenta-se que o governo do estado do Ceará vai adotar o PostgreSQL como SGBD oficial do Estado, ficando todas as suas secretarias obrigadas a converter suas bases para PostgreSQL. O governo do Rio de Janeiro já faz isso há alguns anos.

Equipe de Desenvolvimento do PostgreSQL

These are the fine people that make PostgreSQL what it is today!

Core Team

Major Contributors

Contributors

Gevik Babakhani (pgdev at xs4all.nl)	Dennis Björklund (d at zigo.dhs.org)
Christopher Browne (cbbrowne at ca.afiliat.info)	D'Arcy Cain (darcy at druid.net)
Sean Chittenden (sean at chittenden.org)	Fabien Coelho (coelho at cri.ensmp.fr)
Korry Douglas (korryd at enterprisedb.com)	Francisco Figueiredo (fxjrlists at yahoo.com.br)
Stephen Frost (sfrost at snowman.net)	Michael Fuhr (mike at fuhr.org)
Michael Glaesemann (grzm at myrealbox.com)	Thomas Hallgren (thomas at tada.se)
John Hansen (john at geeknet.com.au)	Jonah Harris (jonah.harris at gmail.com)
Kris Jurka (jurka at ejurka.com)	Mark Kirkwood (markir at paradise.net.nz)
Manfred Koizar (mkoi-pg at aon.at)	Sergey E. Koposov (math at sai.msu.ru)
Marko Kreen (marko at l-t.ee)	Hannu Krosing (hannu at skype.net)
Heikki Linnakangas (hlinnaka at iki.fi)	Robert Lor (Robert.Lor at Sun.com)
Abhijit Menon-Sen (ams at oryx.com)	Fernando Nasser (fnasser at redhat.com)
Matthew T. O'Connor (matthew at zeut.net)	Euler Taveira de Oliveira (eulerto at yahoo.com.br)
Larry Rosenman (ler at lerctr.org)	Greg Stark (stark at enterprisedb.com)
Pavel Stehule (stehule at kix.fsv.cvut.cz)	Sven Suursoho (sven at spam.pri.ee)
ITAGAKI Takahiro (itadaki.takahiro at lab.ntt.co.jp)	Philip Warner (pjwt at rhyme.com.au)
Joachim Wieland (joe at mcknight.de)	Mark Wong (markw at osdl.org)
Andreas Zeugswetter (ZeugswetterA at wien.spardat.at)	Qingqing Zhou (zhouqq at cs.toronto.edu)

Hackers Emeritus

The following hackers were previously part of the core team. Although they no longer work on the project, they are included here in recognition of their valuable contributions over the years.

Contributor	Contribution
Thomas G. Lockhart (lockhart at alumni.caltech.edu) Jet Propulsion Laboratory Pasadena, California, USA	Worked on documentation, data types (particularly date/time and geometric), and SQL standards compliance.
Vadim B. Mikheev (vadim4o at yahoo.com) San Francisco, California, USA	Undertook large projects, like vacuum, subselects, triggers, Write Ahead Log (WAL) and multi-version concurrency control (MVCC).

Past Contributors

Julian Assange (proff at suburbia.net)	David Bennett (dave at bensoft.com)
Alexey Borzov (borz_off at cs.msu.su)	Emily Boyd (emily at emilyboyd.com)
Justin Clift (justin at postgresql.org)	Massimo Dal Zotto (dz at cs.unitn.it)
Dr_George_D Detlefsen (drgeorge at ilt.com)	Chris Dunlop (chris at onthe.net.au)
J. Douglas Dunlop (dunlop at eol.ists.ca)	Oliver Elphick (olly at lfix.co.uk)
Brian E. Gallew (geek+ at cmu.edu)	Cees de Groot (C.deGroot at inter.NL.net)
Christopher Kings-Lynne (chris.kingslynne at gmail.com)	Anoop Kumar (anoopk at pervasive-postgres.com)
Randy Kunkee (kunkee at Starbase.NeoSoft.COM)	Kurt J. Lidl (lidl at va.pubnix.com)
Barry Lind (barry at xythos.com)	Dan McGuirk (mcguirk at indirect.com)
Roberto Mello (roberto.mello at gmail.com)	Ernst Molitor (ernst.molitor at uni-bonn.de)
Claudio Natoli (claudio.natoli at memetrics.com)	Lamar Owen (lamar.owen at wgcr.org)
Ross J. Reedstrom (reedstrm at rice.edu)	Thomas van Reimersdahl (reimersd at dali.techinfo.rwth-aachen.de)
Sergej Sergeev (sergej at commandprompt.com)	Michael Siebenborn (michael.siebenborn at ae3.Hypo.DE)
Erich Stamberger (eberger at gewi.kfunigraz.ac.at)	Adam Sussman (myddryn at vidya.com)
Reini Urban (urban at x-ray.at)	Paul "Shag" Walmsley (ccshag at cclabs.missouri.edu)
Rick Weldon (rick at wisetech.com)	Jason Wright (jason at shiloh.vnet.net)
Karel Zak (kzak at redhat.com)	

All developers are listed in alphabetical order. Please report omissions or corrections

Lista completa em: <http://www.postgresql.org/community/contributors/>

Guide to PostgreSQL version numbers

Josh Berkus

Thursday, September 20, 2007

With the [recent release of a PostgreSQL update](#), I was reminded again by a bewildered user that a lot of people don't seem to get the essentially simple structure of the PostgreSQL version numbers. There's a [nice page about versioning](#), but I doubt most users have read it.

To add a little to what it says there, consider the PostgreSQL version:

PostgreSQL 8 . 1 . 10

The **8** is a "major version number", but could also be known as the "marketing version number". Every few years, based on making some long-sought landmark, we bump the first digit. For **7** (in 1999) it was not crashing; for **8** it was the Windows port, combined with cumulative improvements. When we'll do **9**, nobody knows; probably when we start to beat proprietary databases on *all* workloads.

The **1** in the second digit place is *also* a major release. It indicates the annual release of PostgreSQL. You combine the first two digits ... in this case "**8.1**" to make your major version, or major release, or branch, or sometimes just "version" of PostgreSQL. Users upgrading between major versions can expect an involved upgrade process and testing, which is why we support back branches so long. This using the first two digits confuses many commercial software developers, who assume that only the first digit is significant. If we used only the first digit, though, we'd be close to releasing PostgreSQL version **19**.

That **10** in the last place is the "minor version", which is a synonym for "patch release number". **8.1.10** thus includes 10 cumulative patches to major version **8.1**. Whatever your major version, it's very important that you update to the latest minor version of it. The PostgreSQL project issues only bug fixes in patches, and never feature enhancements, so if you are not on the latest minor version, then you're exposing yourself to known potential security and data loss threats.

Updating to the latest minor release does not require an "initdb" or "dump/reload", and most releases is possible within 2 minutes of downtime (as long as it takes you to copy over the binaries and restart). Occasionally, however, a security fix requires some API changes which will be documented in the release notes. If you're jumping a half dozen minor releases, make sure you scroll back through the release notes for warnings.

Now, go and update your PostgreSQL servers!

1.7) Instalação do PostgreSQL 8.3 no Windows

Download

Fazer o download de - <http://www.postgresql.org/ftp/binary/v8.3.0/win32/>
Caso esteja baixando outra versão deverá acessar:
<http://www.postgresql.org/ftp> e depois clicar em win32.

No caso da 8.3 selecionar e baixar o arquivo [postgresql-8.3.0-1.zip](#), que tem 20MB. É importante verificar o tamanho do arquivo baixado para garantir que o download está correto.

Remover Usuário Anterior

Caso tenha uma instalação anterior do postgresql deverá, antes de instalar a nova, remover o super usuário da versão anterior:

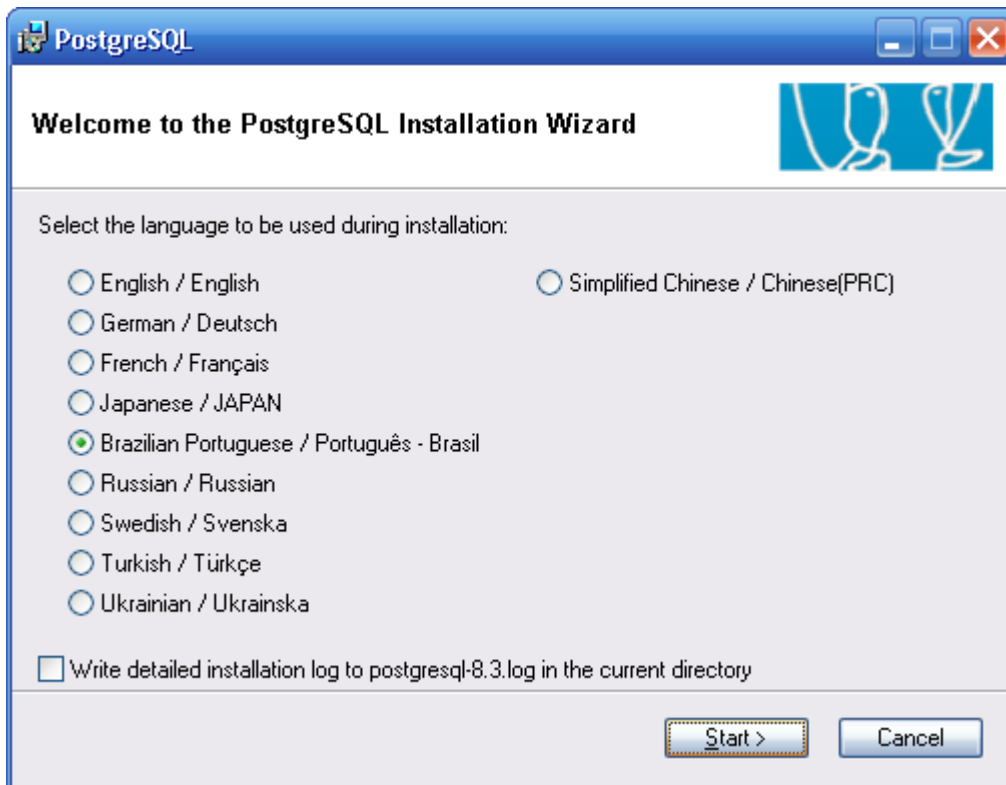
- Painel de controle – Ferramentas Administrativas – Administração de Computadores – Usuários e Grupos locais – Usuários (remover o postgres)

Descompactar e Instalar

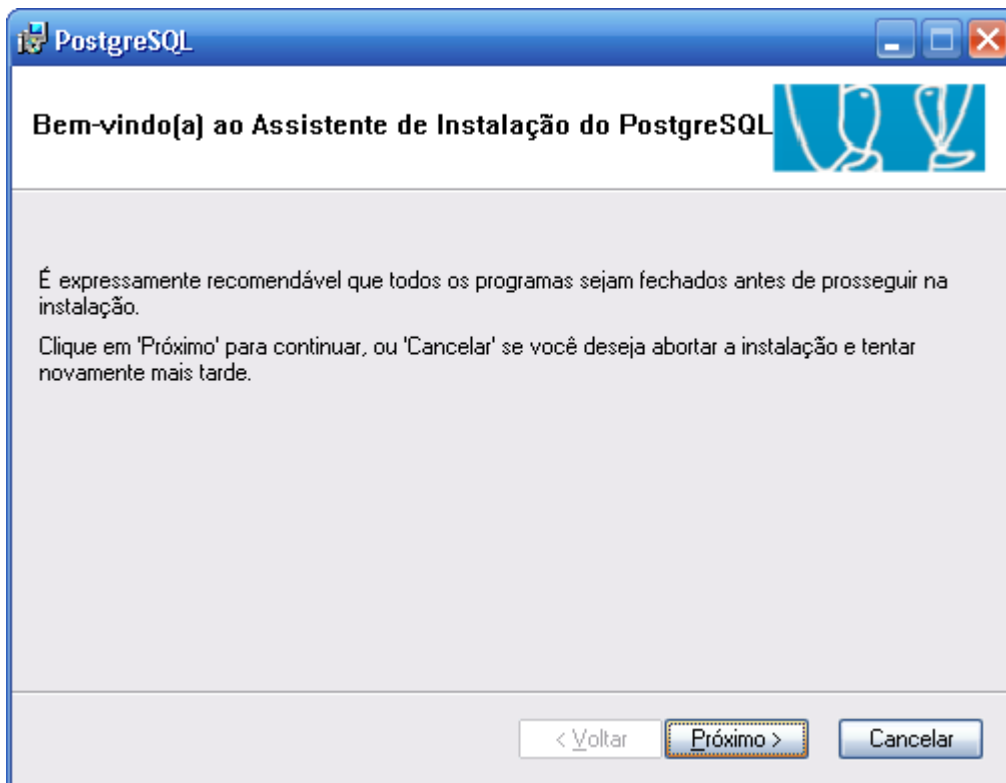
Descompacte o arquivo e verá 5 arquivos: postgresql-8.3.msi, postgresql-8.3-int.msi, README.txt, SETUP.bat e UPGRADE.bat.

O arquivo UPGRADE.bat é para efetuar uma atualização de versão existente.
O arquivo postgresql-8.3.msi é que deve ser executado para a instalação.

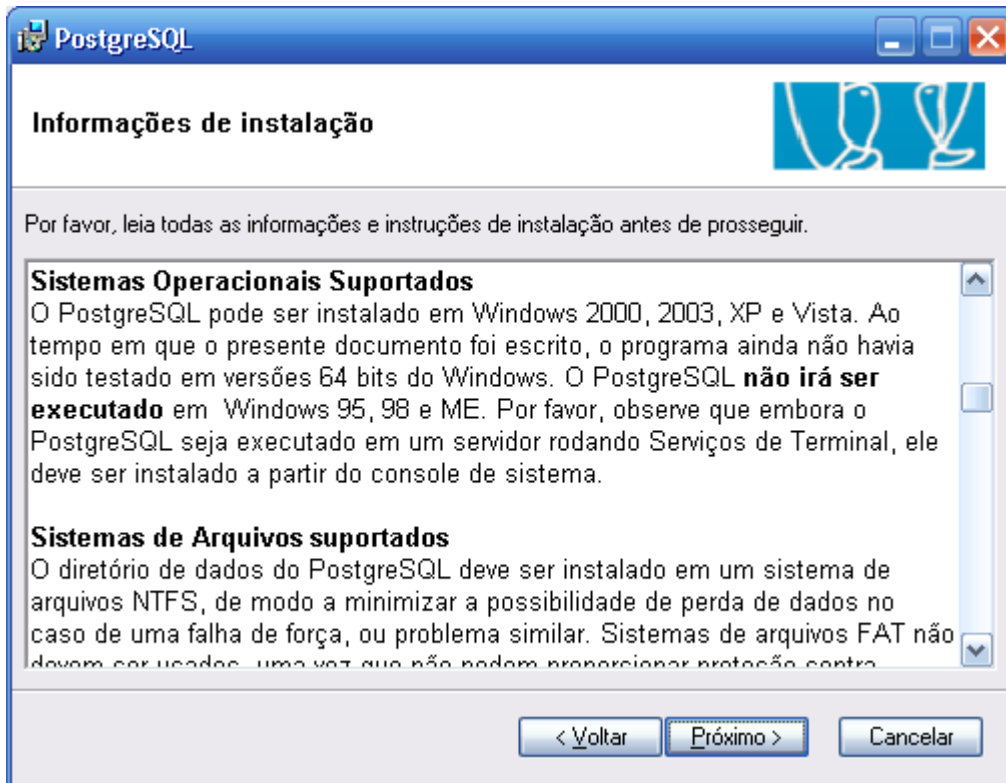
Execute o arquivo postgresql-8.3.msi:



Clique em Start.

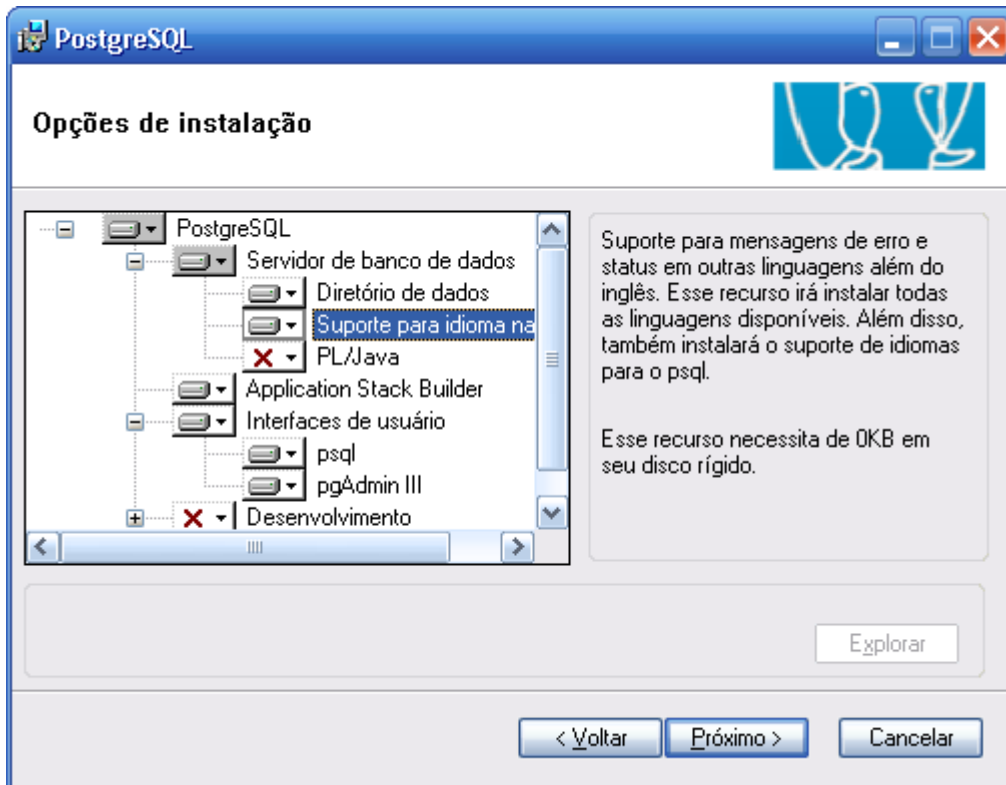


Desta tela em diante a instalação estará em português do Brasil.
Feche todos os programas abertos e clique em próximo.



Neste tela temos informações importantes, como os sistemas operacionais suportados pela versão for Windows do PostgreSQL, que são o 2000, 2003, XP e Vista, como também o sistema de arquivos que é somente o NTFS.

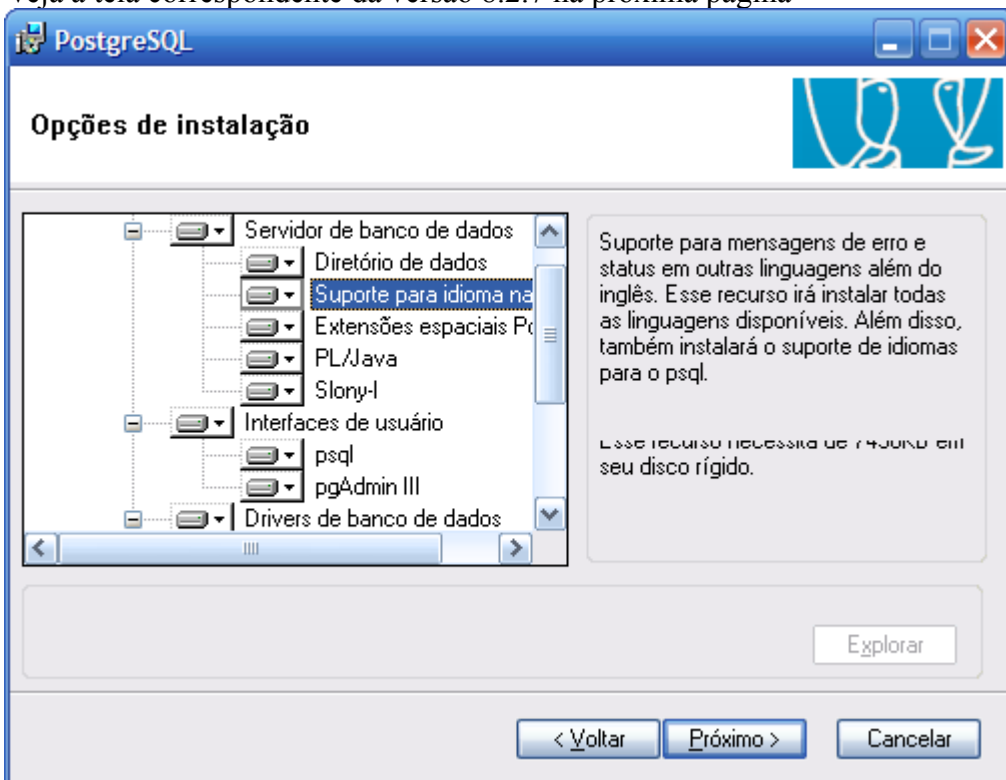
Clique em Próximo.

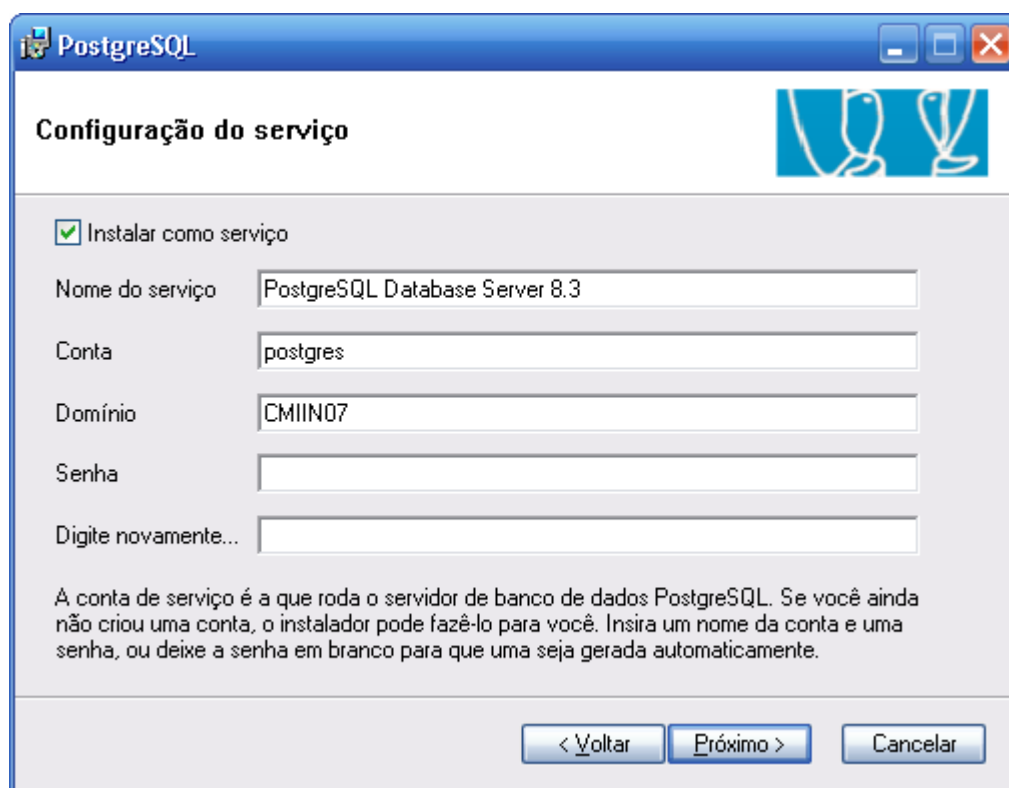


Nesta tela devemos ativar o Suporte para idiomas nativos, se quisermos ver mensagens de erro e o psql em português.

Clique em Próximo.

Veja a tela correspondente da versão 8.2.7 na próxima página





The screenshot shows a Windows-style window titled "PostgreSQL" with a standard title bar (minimize, maximize, close buttons). The window's content area has a header "Configuração do serviço" (Service Configuration) with a PostgreSQL logo to its right. Below the header, there is a checkbox labeled "Instalar como serviço" (Install as service) which is checked. Underneath, there are five input fields: "Nome do serviço" (Service name) containing "PostgreSQL Database Server 8.3", "Conta" (Account) containing "postgres", "Domínio" (Domain) containing "CMIIN07", "Senha" (Password), and "Digite novamente..." (Type again...). The password fields are currently empty. A paragraph of text explains that the service account is used to run the PostgreSQL database server and that the installer can create one if needed. At the bottom, there are three buttons: "< Voltar" (Back), "Próximo >" (Next), and "Cancelar" (Cancel).

PostgreSQL

Configuração do serviço

☒ Instalar como serviço

Nome do serviço: PostgreSQL Database Server 8.3

Conta: postgres

Domínio: CMIIN07

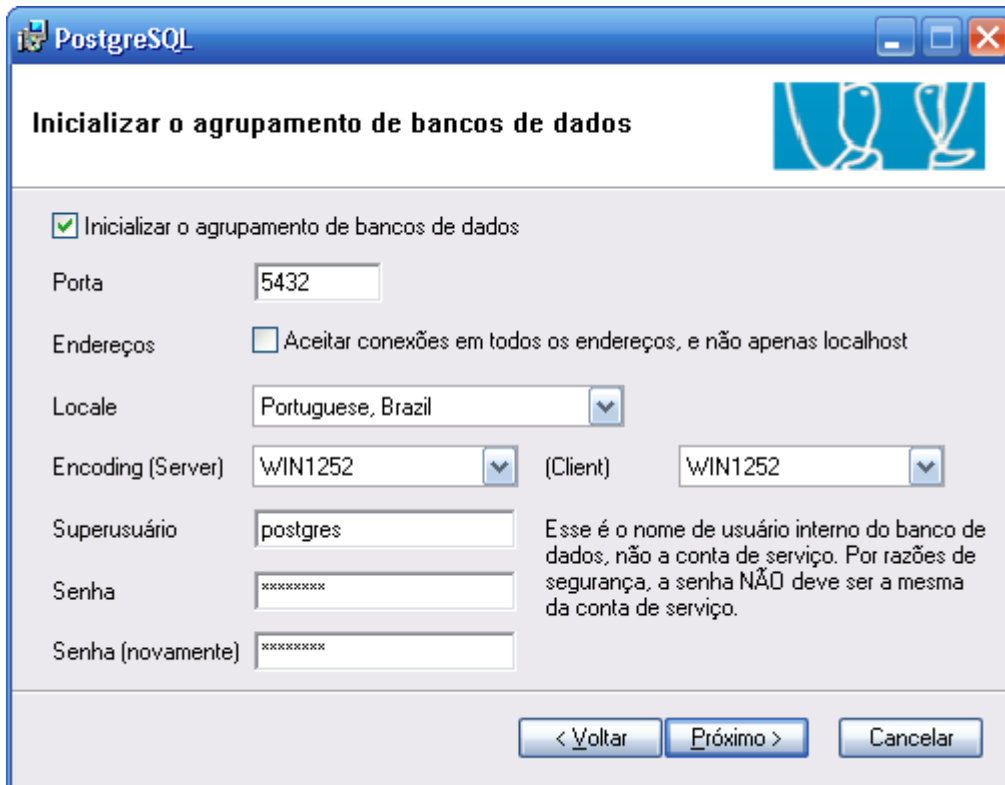
Senha:

Digite novamente...

A conta de serviço é a que roda o servidor de banco de dados PostgreSQL. Se você ainda não criou uma conta, o instalador pode fazê-lo para você. Insira um nome da conta e uma senha, ou deixe a senha em branco para que uma seja gerada automaticamente.

< Voltar Próximo > Cancelar

Nesta tela deixe a senha em branco, clique em Próximo e confirme. Veja que é opcional instalar como serviço, mas altamente recomendado.



PostgreSQL

Inicializar o agrupamento de bancos de dados

☒ Inicializar o agrupamento de bancos de dados

Porta: 5432

Endereços: ☐ Aceitar conexões em todos os endereços, e não apenas localhost

Locale: Portuguese, Brazil

Encoding (Server): WIN1252 (Client): WIN1252

Superusuário: postgres

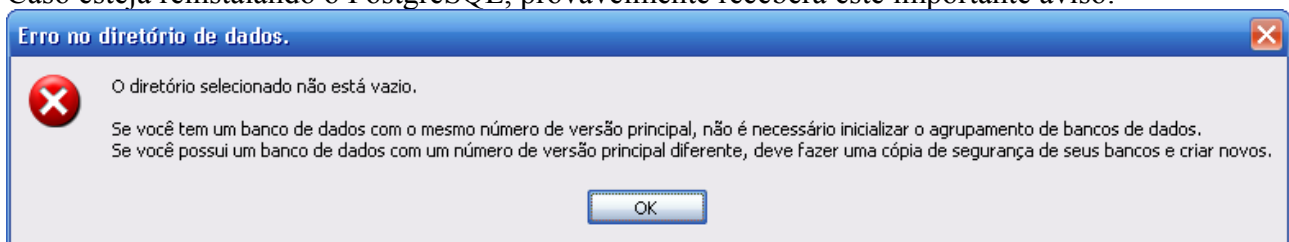
Senha: xxxxxxxx

Senha (novamente): xxxxxxxx

Esse é o nome de usuário interno do banco de dados, não a conta de serviço. Por razões de segurança, a senha NÃO deve ser a mesma da conta de serviço.

< Voltar Próximo > Cancelar

Caso esteja reinstalando o PostgreSQL, provavelmente receberá este importante aviso:



Erro no diretório de dados.

O diretório selecionado não está vazio.

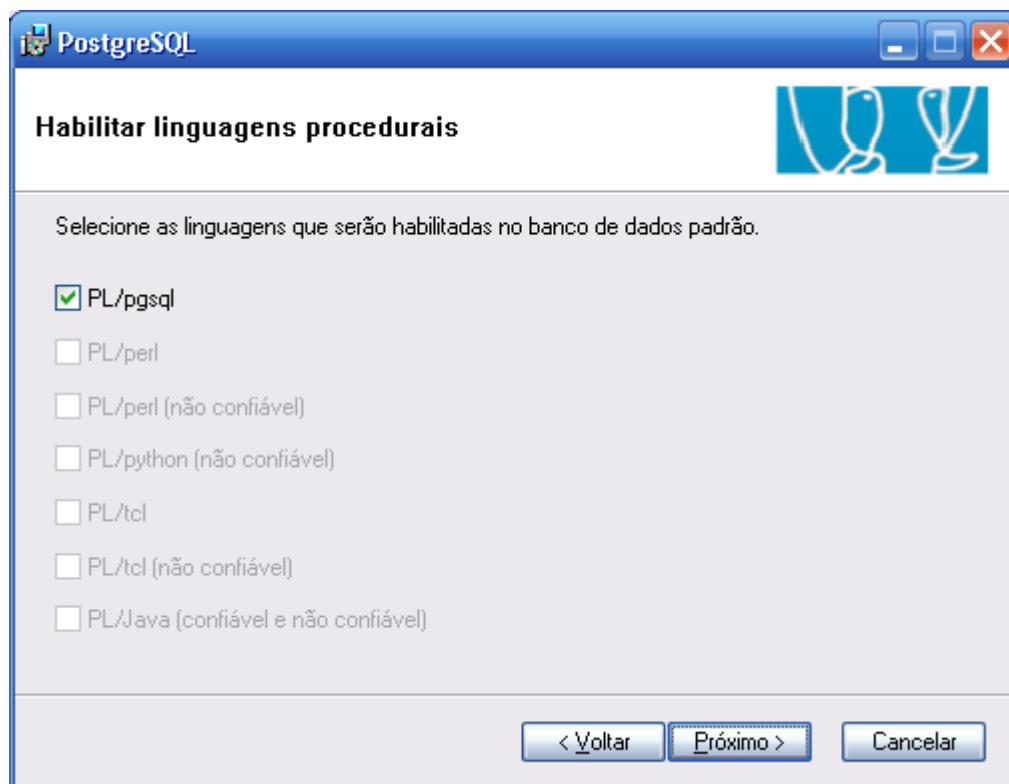
Se você tem um banco de dados com o mesmo número de versão principal, não é necessário inicializar o agrupamento de bancos de dados.
Se você possui um banco de dados com um número de versão principal diferente, deve fazer uma cópia de segurança de seus bancos e criar novos.

OK

Na tela abaixo deixemos o **Encoding** do servidor e do cliente como Win1252, que são o padrão do Windows. Entre também com a senha do super usuário e clique em Próximo.

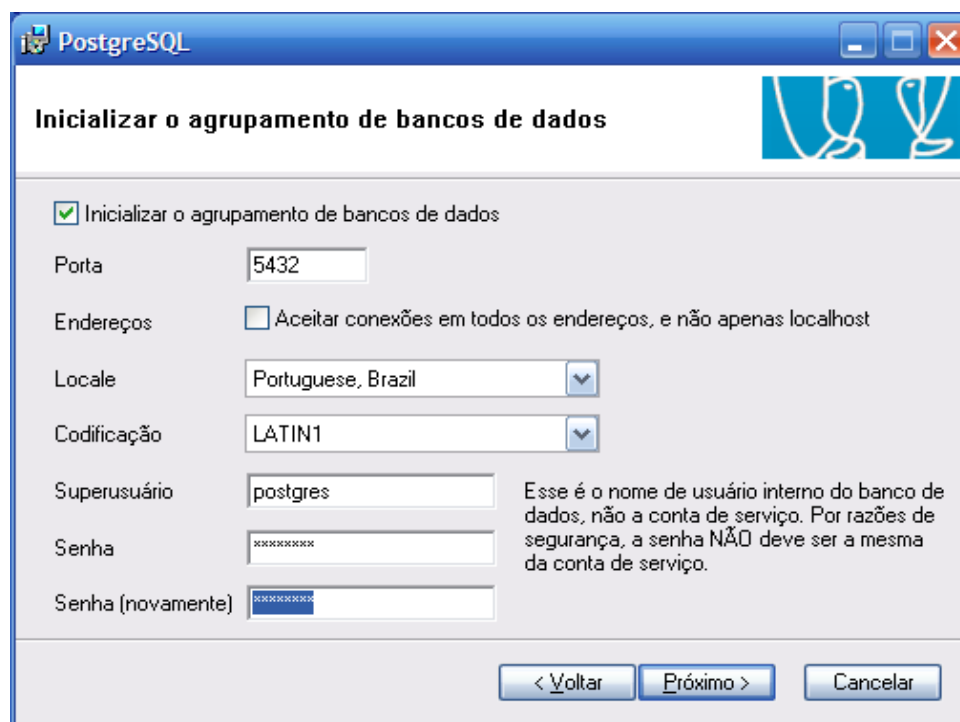
O **Locale** deve ser de acordo com o sistema operacional.

Nesta tela existe a opção de já configurar para aceitar conexões em todos os endereços, mas somente utilize com grande segurança do que está fazendo, pois o ideal em termos de segurança é permitir apenas acesso local.

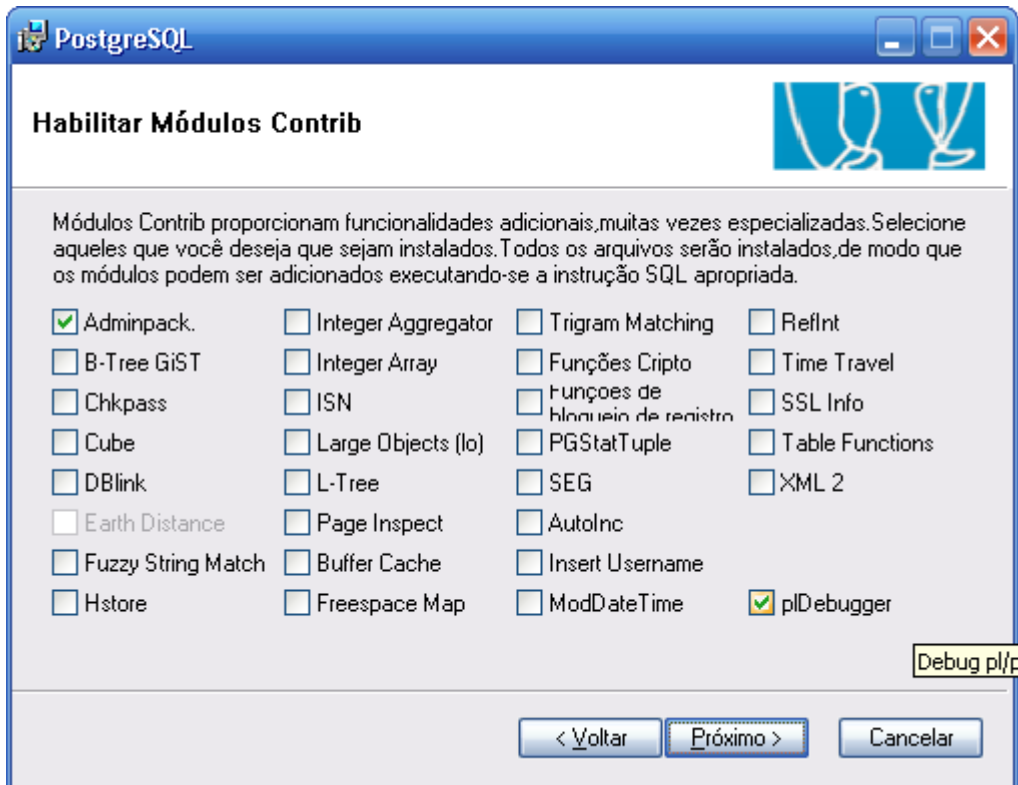


Nesta tela desmarque o item PL/pgsql e lembre de ativar o suporte a plpgsql apenas nos bancos em que for usar essa linguagem. Assim terá os bancos mais limpos.

A tela correspondente da versão 8.2.7

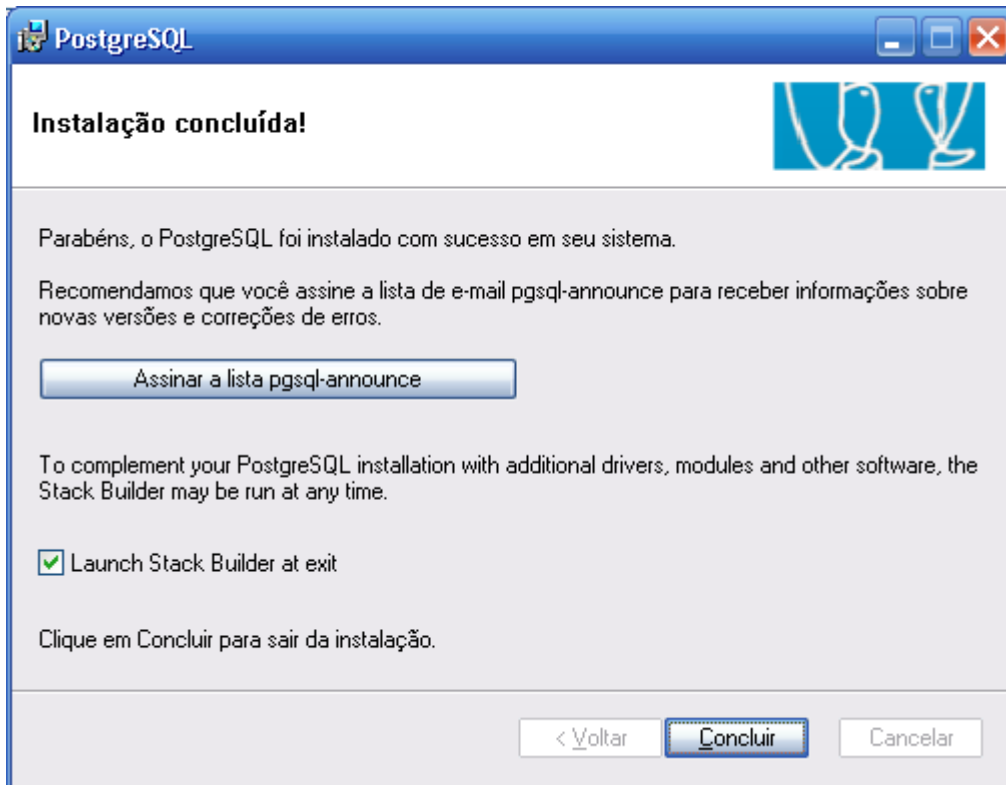


Clique em
Próximo.



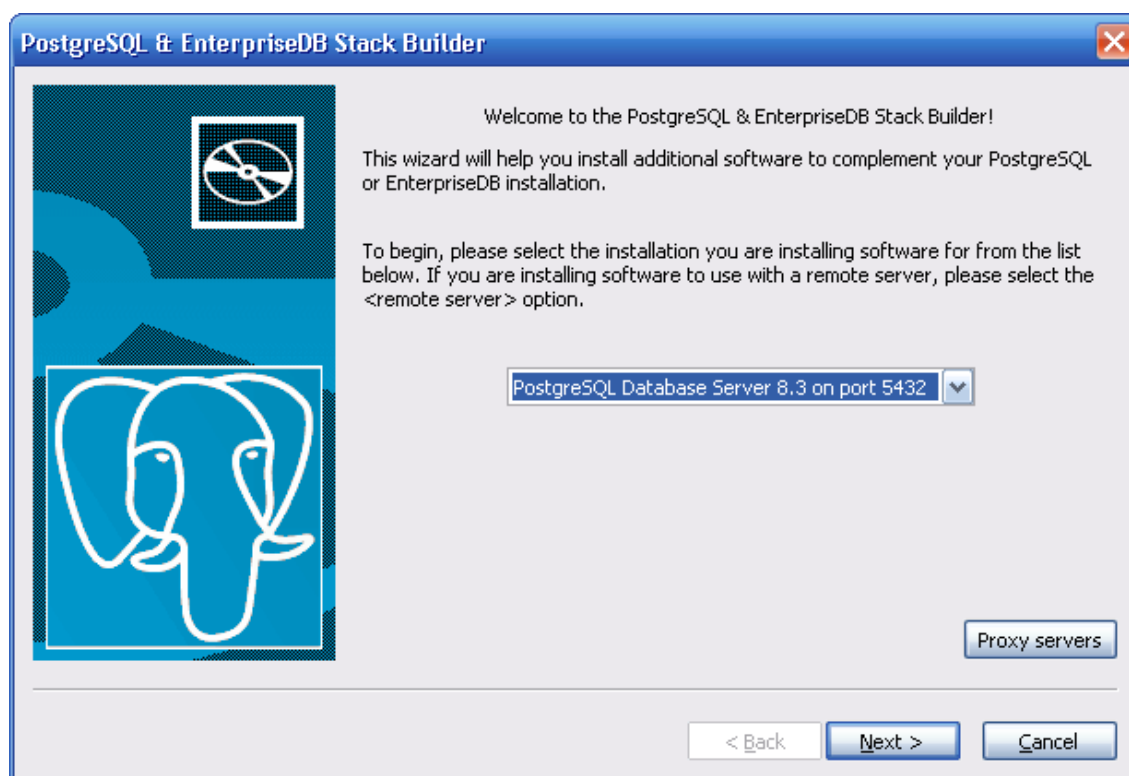
Esta tela é a das extensões chamadas cotribs. Marque apenas se precisar e souber de fato o que está fazendo. Deixe como está e clique em Próximo e confirme novamente na próxima tela..

Caso tenha um bom firewall instalado verá duas solicitações de uso das portas 5432 e da 2427. Aceite o acesso.



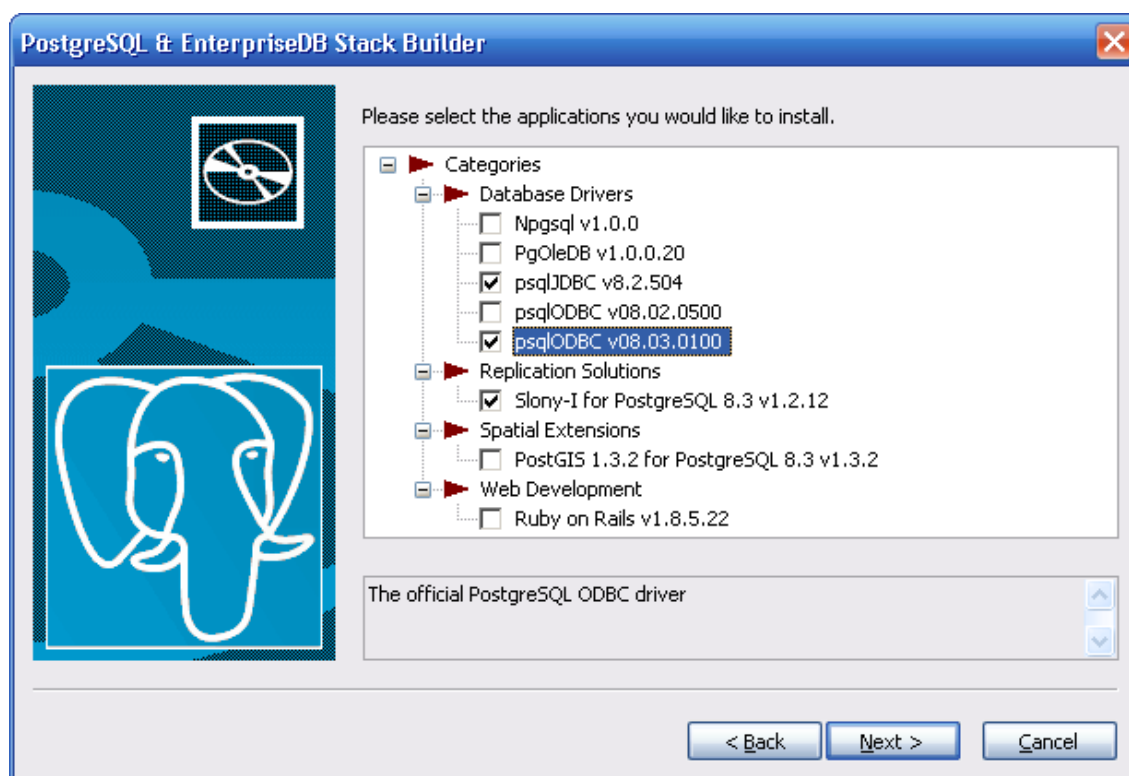
Caso você esteja realmente empenhado em aprender sobre o PostgreSQL aproveite esta oportunidade para assinar a lista pgsql-announce clicando no botão. Esta lista envia em torno de um e-mail por dia sobre as novidades do postgresql, novas ferramentas e informações importantes de segurança, bugs, etc.

Também nesta tela temos a opção de instalar drivers: JDBC para o postgresql, ODBC e outros, como outros softwares adicionais. Basta deixar marcado o checkbox e clicar em Concluir.

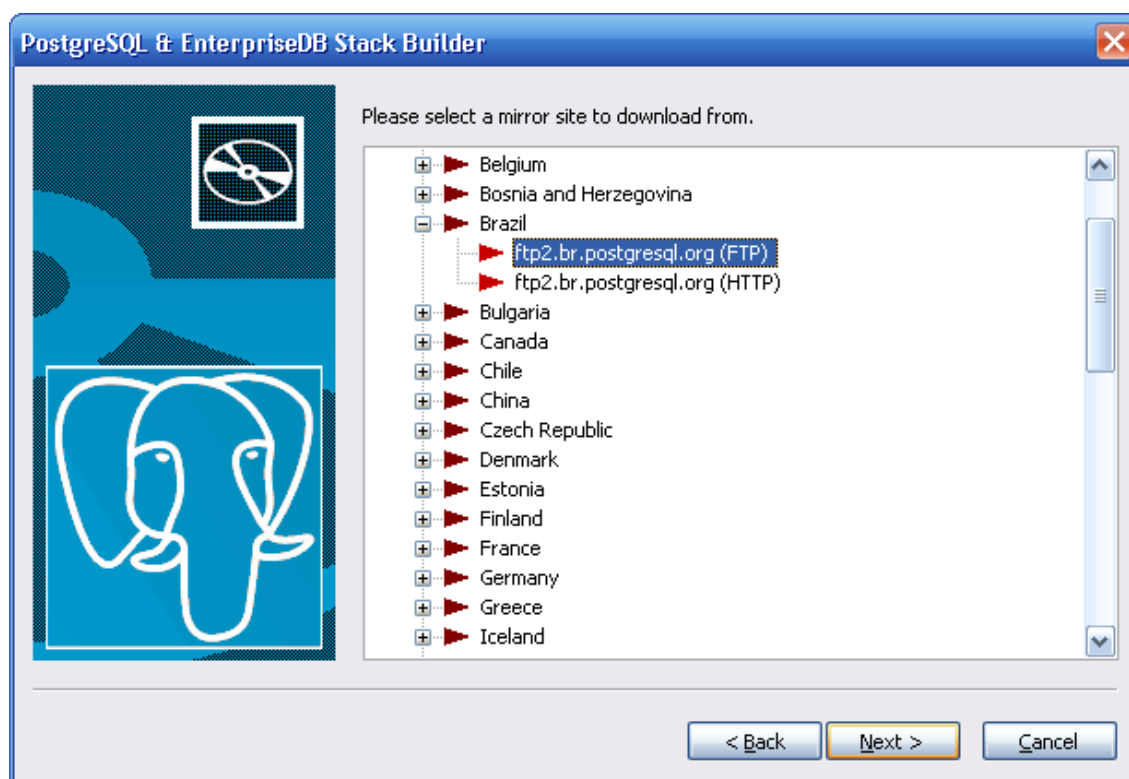


Selecione o servidor e caso esteja numa rede com proxy entre com as informações e clique em Next.

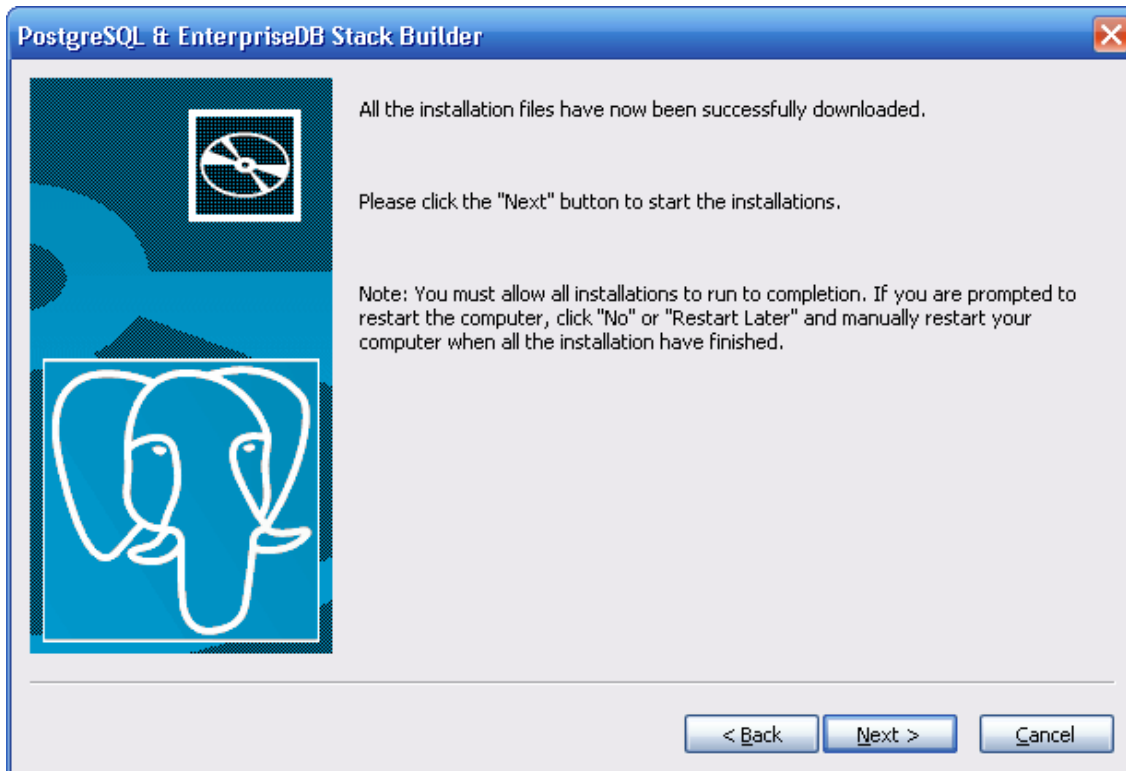
Novamente deverá dar permissão no firewall para que acesse o site para os downloads.



Marquei dois drivers importantes e o software para replicação Slony.
Deve seleccionar os que deseja e clicar em Next.



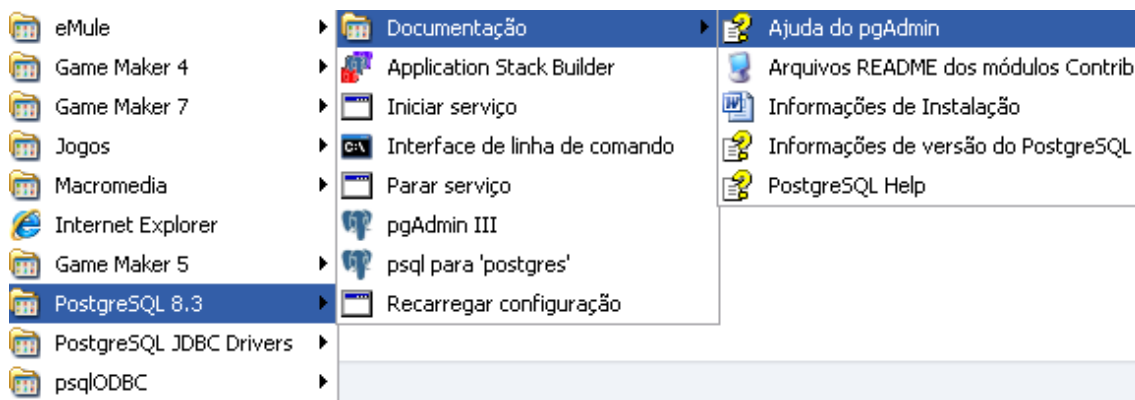
Apenas selecione o espelho e clique em Next e na próxima tela confirme em Next novamente e aguarde o download.



Veja a recomendação de caso seja solicitado reboot no micro, cancele e faça o reboot manualmente depois, quando todos os processos da instalação se concluírem.

Agora ele irá instalar os programas selecionados e finalizar a instalação.

Veja que agora temos alguns grupos de programas junto ao do PostgreSQL:



Temos aí o postgresql, os drivers, o pgadmin e o psql, além da documentação oficial em inglês sobre o postgresql e sobre o pgadmin. Além de alguns links para parar o serviço e iniciar e outros.

"Instalação" e Configurações do PostgreSQL for Windows do tipo Non-installer

Download do pacote para Win32 tipo binaries-no-installer:

[postgresql-8.3.0-2-binaries-no-installer.zip](http://www.postgresql.org/ftp/binary/latest/8.3.0-2/binaries-no-installer.zip)

Descompactar (ex.: raiz do c:\)

Acessar o prompt de comando e entrar no diretório criado (pgsql)

Criar o cluster

cd pgsql\bin

initdb -D pgdata # ou outro diretório, como c:\data

Com o comando acima será criado o cluster de bancos do PostgreSQL na subpasta (pgdata) no diretório bin

Iniciar o processo

bin\pg_ctl -D pgdata start

Criar usuário

bin\createuser postgres

Acessar psql

bin\psql -U postgres

Parar o processo

bin\pg_ctl -U postgres -D pgdata stop

Para tornar mais ágeis e não precisar lembrar destes comandos podemos criar arquivos .bat que executem estes comandos acima.

Sugiro que criemos:

- Para a criação do cluster e do usuario não há necessidade, já que será uma única vez
- um para startar o processo
- um para acessar o psql
- um para parar o processo

Para facilitar ainda mais os .bats devem ficar no diretório bin.

pgstart.bat

pg_ctl -D pgdata start

Obs.: após este comando tecle Enter

pgpsql.bat

psql -U postgres

Obs.: este comando acessa o banco default do postgres, que é o postgres.

pgstop.bat

pg_ctl -D pgdata stop

<http://www.microsoft.com/downloads/details.aspx?FamilyID=200b2fd9-ae1a-4a14-984d-389c36f85647&DisplayLang=en>

2) Utilizando SQL para selecionar, filtrar e agrupar registros

- 2.1) A linguagem SQL
- 2.2) Principais Palavras-Chave e Identificadores
- 2.3) Trabalhando e Manipulando Valores Nulos
- 2.4) Utilizando Comentários
- 2.5) Entendendo os Tipos de Dados
- 2.6) Utilizando Expressões e Constantes
- 2.7) Ocultando Linhas Duplicadas em uma Consulta (DISTINCT)
- 2.8) Limitando o Resultado do Select
- 2.9) Utilizando o Comando Case
- 2.10) Substituindo Valores Nulos para Formatar dados Retornados na Consulta

2.1) A linguagem SQL

Origem: Wikipédia, a enciclopédia livre.

Structured Query Language, ou **Linguagem de Consulta Estruturada** ou **SQL**, é uma linguagem de pesquisa declarativa para [banco de dados relacional](#) (base de dados relacional). Muitas das características originais do SQL foram inspiradas na [álgebra relacional](#).

O SQL foi desenvolvido originalmente no início dos anos 70 nos laboratórios da [IBM](#) em San Jose, dentro do projeto [System R](#), que tinha por objetivo demonstrar a viabilidade da implementação do [modelo relacional](#) proposto por [E. F. Codd](#). O nome original da linguagem era **SEQUEL**, acrônimo para "**Structured English Query Language**" (**Linguagem de Consulta Estruturada em Inglês**) [\[1\]](#), vindo daí o fato de, até hoje, a sigla, em inglês, ser comumente pronunciada "síquel" ao invés de "és-kiú-él", letra a letra. No entanto, em português, a pronúncia mais corrente é a letra a letra: "ése-quê-éle".

A linguagem SQL é um grande padrão de banco de dados. Isto decorre da sua simplicidade e facilidade de uso. Ela se diferencia de outras linguagens de consulta a banco de dados no sentido em que uma consulta SQL especifica a forma do resultado e não o caminho para chegar a ele. Ela é uma linguagem declarativa em oposição a outras linguagens procedurais. Isto reduz o ciclo de aprendizado daqueles que se iniciam na linguagem.

Embora o SQL tenha sido originalmente criado pela [IBM](#), rapidamente surgiram vários "dialectos" desenvolvidos por outros produtores. Essa expansão levou à necessidade de ser criado e adaptado um padrão para a linguagem. Esta tarefa foi realizada pela [American National Standards Institute](#) (ANSI) em [1986](#) e [ISO](#) em [1987](#).

O SQL foi revisto em [1992](#) e a esta versão foi dado o nome de SQL-92. Foi revisto novamente em [1999](#) e [2003](#) para se tornar SQL:1999 (SQL3) e SQL:2003, respectivamente. O SQL:1999 usa [expressões regulares](#) de emparelhamento, *queries* recursivas e [gatilhos](#) (*triggers*). Também foi feita uma adição controversa de tipos não-escalados e algumas características de [orientação a objeto](#). O SQL:2003 introduz características relacionadas ao [XML](#), seqüências padronizadas e colunas com valores de auto-generalização (inclusive colunas-identidade).

Tal como dito anteriormente, o SQL, embora padronizado pela ANSI e [ISO](#), possui muitas variações e extensões produzidos pelos diferentes fabricantes de sistemas gerenciadores de bases de dados. Tipicamente a linguagem pode ser migrada de plataforma para plataforma sem mudanças estruturais principais.

Outra aproximação é permitir para código de idioma procedural ser embutido e interagir com o [banco de dados](#). Por exemplo, o [Oracle](#) e outros incluem [Java](#) na base de dados, enquanto o [PostgreSQL](#) permite que funções sejam escritas em [Perl](#), [Tcl](#), ou [C](#), entre outras linguagens.

2.2) Principais Palavras-Chave e Identificadores

DML - Linguagem de Manipulação de Dados

Primeiro há os elementos da DML (Data Manipulation Language - Linguagem de Manipulação de Dados). A DML é um subconjunto da linguagem usada para selecionar, inserir, atualizar e apagar dados.

- **SELECT** é o comumente mais usado do DML, comanda e permite ao usuário especificar uma query como uma descrição do resultado desejado. A questão não especifica como os resultados deveriam ser localizados.
- **INSERT** é usada para somar uma fila (formalmente uma tupla) a uma tabela existente.
- **UPDATE** para mudar os valores de dados em uma fila de tabela existente.
- **DELETE** permite remover filas existentes de uma tabela.

DDL - Linguagem de Definição de Dados

O segundo grupo é a DDL (Data Definition Language - Linguagem de Definição de Dados). Uma DDL permite ao usuário definir tabelas novas e elementos associados. A maioria dos bancos de dados de SQL comerciais tem extensões proprietárias no DDL.

Os comandos básicos da DDL são poucos

- **CREATE** cria um objeto (uma [Tabela](#), por exemplo) dentro da base de dados.
- **DROP** apaga um objeto do banco de dados.

Alguns sistemas de banco de dados usam o comando ALTER, que permite ao usuário alterar um objeto, por exemplo, adicionando uma coluna a uma tabela existente.

outros comandos DDL:

- **ALTER TABLE**
- **CREATE INDEX**
- **ALTER INDEX**
- **DROP INDEX**
- **CREATE VIEW**
- **DROP VIEW**

DCL - Linguagem de Controle de Dados

O terceiro grupo é o DCL (Data Control Language - Linguagem de Controle de Dados). DCL controla os aspectos de autorização de dados e licenças de usuários para controlar quem tem acesso para ver ou manipular dados dentro do banco de dados.

Duas palavras-chaves da DCL:

- **GRANT** - autoriza ao usuário executar ou setar operações.
- **REVOKE** - remove ou restringe a capacidade de um usuário de executar operações.

DTL - Linguagem de Transação de Dados

- **BEGIN WORK** (ou **START TRANSACTION**, dependendo do dialeto SQL) pode ser usado para marcar o começo de uma transação de banco de dados que pode ser completada ou não.
- **COMMIT** envia todos os dados das mudanças permanentemente.
- **ROLLBACK** faz com que as mudanças nos dados existentes desde que o último **COMMIT** ou **ROLLBACK** sejam descartadas.

COMMIT e **ROLLBACK** interagem com áreas de controle como transação e locação. Ambos terminam qualquer transação aberta e liberam qualquer cadeado ligado a dados. Na ausência de um **BEGIN WORK** ou uma declaração semelhante, a semântica de SQL é dependente da implementação.

outros comandos DCL:

- **ALTER PASSWORD**
- **CREATE SYNONYM**

DQL - Linguagem de Consulta de Dados

Embora tenha apenas um comando a DQL é a parte da SQL mais utilizada. O comando **SELECT** é composta de várias cláusulas e opções, possibilitando elaborar consultas das mais simples as mais elaboradas.

Fonte: <http://pt.wikipedia.org/wiki/Sql>

Palavras-chaves: <http://pgdocptbr.sourceforge.net/pg80/sql-keywords-appendix.html> e <http://www.postgresql.org/docs/8.3/interactive/sql-keywords-appendix.html>

Tutoriais de SQL Online - <http://sqlcourse.com/> <http://sqlzoo.net/> e <http://www.cfxweb.net/modules.php?name=News&file=article&sid=161>

2.3) Trabalhando e Manipulando Valores Nulos

Em SQL NULL é para valores inexistentes. Regra geral: NULL se propaga, o que significa que com quem NULL se combina o resultado será um NULL. NULL não é zero, não é string vazia nem string de comprimento zero.

Um exemplo: num cadastro de alunos, para o aluno que ainda não se conhece a nota, não é correto usar zero para sua nota, mas sim NULL.

Não se pode efetuar cálculos de expressões onde um dos elementos é NULL.

COMPARANDO NULLs

NOT NULL com NULL -- Unknown

NULL com NULL -- Unknown

CONVERSÃO DE/PARA NULL

NULLIF() e COALESCE()

NULLIF(valor1, valor2)

NULLIF – Retorna NULL se, e somente se, valor1 e valor2 forem iguais, caso contrário retorna valor1.

Algo como:

```
if (valor1 == valor2){  
  then NULL  
  else valor1;  
}
```

Retorna valor1 somente quando valor1 == valor2.

COALESCE – retorna o primeiro de seus argumentos que não for NULL. Só retorna NULL quando todos os seus argumentos forem NULL.

Uso: mudar o valor padrão cujo valor seja NULL.

```
create table nulos(nulo int, nulo2 int, nulo3 int);
```

```
insert into nulos values (1,null,null);
```

```
select coalesce(nulo, nulo2, nulo3) from nulos; - - Retorna 1, valor do campo nulo;
```

```
select coalesce(nulo2, nulo3) from nulos; - - Retorna NULL, pois ambos são NULL.
```

GREATEST - Retorna o maior valor de uma lista - SELECT GREATEST(1,4,6,8,2); - - 8

LEAST - Retorna o menor valor de uma lista.

Todos os valores da lista devem ser do mesmo tipo e nulos são ignorados.

Obs.: Ambas as funções acima não pertencem ao SQL standard, mas são uma extensão do PostgreSQL.

CONCATENANDO NULLs

A regra é: NULL se propaga. Qualquer que concatene com NULL gerará NULL.

STRING || NULL -- NULL

Usos:

- Como valor default para campos que futuramente receberão valor.
- Valor default para campos que poderão ser sempre inexistentes.

2.4) Utilizando Comentários

Ao criar scripts SQL a serem importados pelo PostgreSQL, podemos utilizar dois tipos de comentários:

```
-- Este é um comentário padrão SQL e de uma única linha
/* Este é um comentário
oriundo da linguagem C
e para múltiplas linhas e que também é aceito
nos scripts SQL */
```

2.5) Entendendo os Tipos de Dados

Relação completa de tipos de dados da versão 8.3.1:

8.1. [Numeric Types](#)

8.1.1. [Integer Types](#)

8.1.2. [Arbitrary Precision Numbers](#)

8.1.3. [Floating-Point Types](#)

8.1.4. [Serial Types](#)

8.2. [Monetary Types](#)

8.3. [Character Types](#)

8.4. [Binary Data Types](#)

8.5. [Date/Time Types](#)

8.5.1. [Date/Time Input](#)

8.5.2. [Date/Time Output](#)

8.5.3. [Time Zones](#)

8.5.4. [Internals](#)

- 8.6. [Boolean Type](#)
- 8.7. [Enumerated Types](#)
 - 8.7.1. [Declaration of Enumerated Types](#)
 - 8.7.2. [Ordering](#)
 - 8.7.3. [Type Safety](#)
 - 8.7.4. [Implementation Details](#)
- 8.8. [Geometric Types](#)
 - 8.8.1. [Points](#)
 - 8.8.2. [Line Segments](#)
 - 8.8.3. [Boxes](#)
 - 8.8.4. [Paths](#)
 - 8.8.5. [Polygons](#)
 - 8.8.6. [Circles](#)
- 8.9. [Network Address Types](#)
 - 8.9.1. [inet](#)
 - 8.9.2. [cidr](#)
 - 8.9.3. [inet vs. cidr](#)
 - 8.9.4. [macaddr](#)
- 8.10. [Bit String Types](#)
- 8.11. [Text Search Types](#)
 - 8.11.1. [tsvector](#)
 - 8.11.2. [tsquery](#)
- 8.12. [UUID Type](#)
- 8.13. [XML Type](#)
 - 8.13.1. [Creating XML Values](#)
 - 8.13.2. [Encoding Handling](#)
 - 8.13.3. [Accessing XML Values](#)
- 8.14. [Arrays](#)
 - 8.14.1. [Declaration of Array Types](#)
 - 8.14.2. [Array Value Input](#)
 - 8.14.3. [Accessing Arrays](#)
 - 8.14.4. [Modifying Arrays](#)
 - 8.14.5. [Searching in Arrays](#)
 - 8.14.6. [Array Input and Output Syntax](#)
- 8.15. [Composite Types](#)
 - 8.15.1. [Declaration of Composite Types](#)
 - 8.15.2. [Composite Value Input](#)
 - 8.15.3. [Accessing Composite Types](#)
 - 8.15.4. [Modifying Composite Types](#)
 - 8.15.5. [Composite Type Input and Output Syntax](#)
- 8.16. [Object Identifier Types](#)
- 8.17. [Pseudo-Types](#)

Mais detalhes em: <http://www.postgresql.org/docs/8.3/interactive/datatype.html>

Tipos de Dados Mais Comuns

Numéricos			
Tipo	Tamanho	Apelido	Faixa
smallint (INT2)	2 bytes	inteiro pequeno	-32768 a +32767
integer (INT ou INT4)	4 bytes	inteiro	-2147483648 até +2147483647
bigint (INT8)	8 bytes	inteiro longo	-9223372036854775808 a +9223372036854775807
numeric (p,e)			tamanho variável, precisão especificada pelo usuário. Exato e sem limite
decimal (p,e)			e – escala (casas decimais) p – precisão (total de dígitos, inclusive estala)
real (float)	4 bytes	ponto flutuante	precisão variável, não exato e precisão de 6 dígitos
double precision	8 bytes	dupla precisão	precisão variável, não exato e precisão de 15 dígitos
int (INT4)			mais indicado para índices de inteiros
serial	4 bytes	Inteiro autoinc	1 até 2147483647
bigserial	8 bytes	Inteiro longo autoinc	1 até 9223372036854775807
Caracteres			
character varying(n)		varchar(n)	comprimento variável, com limite
character(n)		char(n)	comprimento fixo, completa com brancos
text			comprimento variável e ilimitado
Desempenho semelhante para os tipos caractere.			
Data/Hora			
timestamp[(p)] [without time zone]	8 bytes	data e hora sem zona	4713 AC a 5874897 DC
timestamp [(p)][with time zone]	8 bytes	data e hora com zona	4713 AC a 5874897 DC
interval	12 bytes	intervalo de tempo	178000000 anos a 178000000 anos
date	4 bytes	somente data	4713 AC até 32767 DC
time [(p)] [without time zone]	8 bytes	somente a hora	00:00:00.00 até 23:59:59.99
time [(p)] [with time zone]	8 bytes	somente a hora	00:00:00.00 até 23:59:59.99
[(p)] - é a precisão, que varia de 0 a 6 e o default é 2.			

Boleanos			
Tipo	Tamanho	Apelido	Faixa
TRUE		Representações:	't', 'true', 'y', 'yes' e '1'
FALSE		Representações:	'f', 'false', 'n', 'no', '0'
Apenas um dos dois estados. O terceiro estado, desconhecido, é representado pelo NULL.			

Exemplo de consulta com boolean:

```
CREATE TEMP TABLE teste1 (a boolean, b text);
INSERT INTO teste1 VALUES (TRUE, 'primeiro');
INSERT INTO teste1 VALUES (FALSE, 'segundo');
SELECT * FROM teste1;
```

Retorno:

```
a | b
---+-----
t | primeiro
f | segundo
```

```
SELECT * FROM teste1 WHERE a = TRUE;    -- Retorna 't'
SELECT * FROM teste1 WHERE a = t;      -- Erro
SELECT * FROM teste1 WHERE a = 't';    -- Retorna 't'
SELECT * FROM teste1 WHERE a = '1';    -- Retorna 't'
SELECT * FROM teste1 WHERE a = 'y';    -- Retorna 't'
SELECT * FROM teste1 WHERE a = '0';    -- Retorna 'f'
SELECT * FROM teste1 WHERE a = 'n';    -- Retorna 'f'
```

Alerta: a entrada pode ser: 1/0, t/f, true/false, TRUE/FALSE, mas o retorno será sempre t ou f.

Obs.: Para campos tipo data que permitam NULL, devemos prever isso na consulta SQL e passar NULL sem delimitadores e valores não NULL com delimitadores.

Obs2: Evite o tipo MONEY que está em obsolescência. Em seu lugar use NUMERIC. Prefira INT (INTEGER) em lugar de INT4, pois os primeiros são padrão SQL. Em geral evitar os nomes INT2, INT4 e INT8, que não são padrão. O INT8 ou bigint não é padrão SQL.

Obs3: Em índices utilize somente INT, evitando smallint e bigint, que nunca serão utilizados.

Tipos SQL Padrão

bit, bit varying, boolean, char, character varying, character, varchar, date, double precision, integer, interval, numeric, decimal, real, smallint, time (com ou sem zona horária), timezone (com ou sem zona horária).

O tipo **NUMERIC** pode realizar cálculos exatos. **Recomendado para quantias monetárias** e outras quantidades onde a exatidão seja importante. Isso paga o preço de queda de desempenho comparado aos inteiros e flutuantes.

Pensando em portabilidade evita usar NUMERIC(12) e usar NUMERIC (12,0).

Alerta: A comparação de igualdade de dois valores de ponto flutuante pode funcionar conforme o esperado ou não.

O PostgreSQL trabalha com datas do calendário Juliano.

Trabalha com a faixa de meio dia de Janeiro de 4713 AC (ano bisexto, domingo de lua nova) até uma data bem distante no futuro. Leva em conta que o ano tem 365,2425 dias.

SERIAL

No PostgreSQL um campo criado do “tipo” SERIAL é internamente uma seqüência, inteiro positivo.

Os principais SGBDs utilizam alguma variação deste tipo de dados (auto-incremento). Serial é o “tipo” auto-incremento do PostgreSQL. Quando criamos um campo do tipo SERIAL ao inserir um novo registro na tabela com o comando INSERT omitimos o campo tipo SERIAL, pois ele será inserido automaticamente pelo PostgreSQL.

```
CREATE TABLE serial_teste (codigo SERIAL, nome VARCHAR(45));  
INSERT INTO serial_teste (nome) VALUES ('Ribamar FS');
```

Obs.: A regra é nomear uma seqüência “serial_teste_codigo_seq”, ou seja, tabela_campo_seq.

```
select * from serial_teste_codigo_seq;
```

Esta consulta acima retorna muitas informações importantes sobre a seqüência criada: nome, valor inicial, incremento, valor final, maior e menor valor além de outras informações.

Veja que foi omitido o campo código mas o PostgreSQL irá atribuir para o mesmo o valor do próximo registro de código. Por default o primeiro valor de um serial é 1, mas se precisarmos começar com um valor diferente veja a solução abaixo:

Setando o Valor Inicial do Serial

```
create sequence produtos_codigo_seq start 9;  
ALTER SEQUENCE tabela_campo_seq RESTART WITH 1000;
```

2.6) Utilizando Expressões e Constantes

Expressões SQL em: http://db.apache.org/derby/docs/dev/pt_BR/ref/rrefsqli19433.html

Precedência dos operadores (decrecente)

Operador/Elemento	Associatividade	Descrição
.	esquerda	separador de nome de tabela/coluna
::	esquerda	conversão de tipo estilo PostgreSQL
[]	esquerda	seleção de elemento de matriz
-	direita	menos unário
^	esquerda	exponenciação
* / %	esquerda	multiplicação, divisão, módulo
+ -	esquerda	adição, subtração
IS		IS TRUE, IS FALSE, IS UNKNOWN, IS NULL
ISNULL		teste de nulo
NOTNULL		teste de não nulo
(qualquer outro)	esquerda	os demais operadores nativos e os definidos pelo usuário
IN		membro de um conjunto
BETWEEN		contido em um intervalo
OVERLAPS		sobreposição de intervalo de tempo
LIKE ILIKE SIMILAR		correspondência de padrão em cadeia de caracteres
< >		menor que, maior que
=	direita	igualdade, atribuição
NOT	direita	negação lógica
AND	esquerda	conjunção lógica
OR	esquerda	disjunção lógica

Detalhes em: <http://pgdocptbr.sourceforge.net/pg80/sql-syntax.html#SQL-PRECEDENCE>
<http://www.postgresql.org/docs/8.3/interactive/sql-syntax-lexical.html#SQL-PRECEDENCE-TABLE>

2.7) Ocultando Linhas Duplicadas em uma Consulta (DISTINCT)

Cláusula *DISTINCT*

Se for especificada a cláusula *DISTINCT*, todas as linhas duplicadas serão removidas do conjunto de resultados (será mantida uma linha para cada grupo de duplicatas).

A cláusula *ALL* especifica o oposto: todas as linhas serão mantidas; este é o padrão.

DISTINCT ON (expressão [, ...]) preserva apenas a primeira linha de cada conjunto de linhas onde as expressões fornecidas forem iguais. As expressões em *DISTINCT ON* são interpretadas usando as mesmas regras da cláusula *ORDER BY* (veja acima). Deve ser observado que a "primeira linha" de cada conjunto é imprevisível, a menos que seja utilizado *ORDER BY* para garantir que a linha desejada apareça na frente. Por exemplo,

```
create temp table dup (c int);
insert into dup (c) values (1);
insert into dup (c) values (2);
insert into dup (c) values (1);
insert into dup (c) values (2);
insert into dup (c) values (1);
select * from dup;
select distinct (c) from dup;
```

retorna o relatório de condição climática mais recente para cada local, mas se não tivesse sido usado *ORDER BY* para obrigar a ordem descendente dos valores da data para cada local, teria sido obtido um relatório com datas imprevisíveis para cada local.

As expressões em *DISTINCT ON* devem corresponder às expressões mais à esquerda no *ORDER BY*. A cláusula *ORDER BY* normalmente contém expressões adicionais para determinar a precedência desejada das linhas dentro de cada grupo *DISTINCT ON*.

DISTINCT – Escrita logo após ***SELECT*** desconsidera os registros duplicados, retornando apenas registros exclusivos.

2.8) Limitando o Resultado do Select

Caso utilizemos a forma:

```
SELECT * FROM tabela;
```

Serão retornados todos os registros com todos os campos. Podemos filtrar tanto os registros quanto os campos que retornam da consulta. Também, ao invés de *, podemos digitar todos os campos da tabela.

Filtrando os Campos que Retornam

Para filtrar os campos, retornando apenas alguns deles, basta ao invés de * digitar apenas os campos que desejamos retornar, por exemplo:

```
SELECT cpf, nome FROM clientes;
```

Filtrando os Registros que Retornam

Para filtrar os registros temos várias cláusulas SQL, como WHERE, GROUP BY, LIMIT, HAVING, etc.

Exemplos:

```
SELECT nome FROM clientes WHERE email = 'ribafs@ribafs.net';
```

```
SELECT nome FROM clientes WHERE idade > 18;
```

```
SELECT nome FROM clientes WHERE idade < 21;
```

```
SELECT nome FROM clientes WHERE idade >= 18;
```

```
SELECT nome FROM clientes WHERE idade <= 21;
```

```
SELECT nome FROM clientes WHERE UPPER(estado) != 'CE';
```

```
select nome, (current_date – data_nasc)/365 as idade from clientes;
```

2.9) Utilizando o Comando Case

CASE é uma expressão condicional do SQL. Uma estrutura semelhante ao IF das linguagens de programação. Caso WHEN algo THEN isso. Vários WHEN podem vir num único CASE. Finaliza com END.

```
CASE WHEN condição THEN resultado
      WHEN condição2 THEN resultado
      [WHEN ...]
      [ELSE resultado]
END
```

Obs.: O que vem entre condeletes é opcional.

Mais detalhes em: <http://pgdocptbr.sourceforge.net/pg80/functions-conditional.html>

William Leite Araújo na lista pgbr-geral:
Ambas as formas são válidas. Você pode usar o

```
SELECT CASE WHEN [teste] THEN ... ELSE [saida] END;
ou
SELECT CASE [coluna]
      WHEN [valor1] THEN resultado
      WHEN [valor2] THEN resultado
      ...
      ELSE [saida] END
```

Quando o teste é entre 2 valores, a primeira forma é a mais aplicável. Quando quer se diferenciar mais de um valor de uma mesma coluna, a segunda é a mais apropriada. Por exemplo:

```
SELECT CASE tipo_credito WHEN 'S' THEN 'Salário' WHEN 'P' THEN 'Pró-labore' WHEN 'D'
THEN 'Depósito' ELSE 'Outros' END as tipo_credito;
```

```
SELECT nome, CASE WHEN EXISTS (SELECT nome FROM clientes WHERE nome=c.nome)
THEN 'sim'
ELSE 'não'
END AS cliente
FROM clientes c;
```

Trazer o nome sempre que existir o nome do cliente.

```
IN
SELECT nome, CASE WHEN nome IN (SELECT nome FROM clientes)
THEN 'sim'
ELSE 'não'
END AS cliente
FROM clientes;
```

```
NOT IN e ANY/SOME
SELECT cpf_cliente, CASE WHEN cpf_cliente = ANY (SELECT cpf_cliente FROM pedidos)
THEN 'sim'
ELSE 'não'
END AS cliente
FROM pedidos;
```

Trazer o CPF

Outros Exemplos:

```
create database dml;
\c dml
create table amigos(
codigo serial primary key,
nome char(45),
idade int
);
```

```
insert into amigos (nome, idade) values ('João Brito', 25);
insert into amigos (nome, idade) values ('Roberto', 35);
insert into amigos (nome, idade) values ('Antônio', 15);
insert into amigos (nome, idade) values ('Francisco Queiroz', 23);
insert into amigos (nome, idade) values ('Bernardo dos Santos', 21);
insert into amigos (nome, idade) values ('Francisca Pinto', 22);
insert into amigos (nome, idade) values ('Natanael', 55);
```

```
select nome,
idade,
case
when idade >= 21 then 'Adulto'
else 'Menor'
end as status
from amigos order by nome;
```

-- CASE WHEN cria uma coluna apenas para exibição

```
create table amigos2(  
codigo serial primary key,  
nome char(45),  
estado char(2)  
);
```

```
insert into amigos2 (nome, estado) values ('João Brito', 'CE');  
insert into amigos2 (nome, estado) values ('Roberto', 'MA');  
insert into amigos2 (nome, estado) values ('Antônio', 'CE');  
insert into amigos2 (nome, estado) values ('Francisco Queiroz', 'PB');  
insert into amigos2 (nome, estado) values ('Bernardo dos Santos', 'MA');  
insert into amigos2 (nome, estado) values ('Francisca Pinto', 'SP');  
insert into amigos2 (nome, estado) values ('Natanael', 'SP');
```

```
select nome,  
estado,  
case  
when estado = 'PB' then 'Fechado'  
when estado = 'CE' or estado = 'SP' then 'Funcionando'  
when estado = 'MA' then 'Funcionando a todo vapor'  
else 'Menor'  
end as status  
from amigos2 order by nome;
```

```
create table notas(  
nota numeric(4,2)  
);
```

```
insert into notas(nota) values (4),  
(6),  
(3),  
(10),  
(6.5),  
(7.3),  
(7),  
(8.8),  
(9);
```

-- Mostrar cada nota junto com a menor nota, a maior nota, e a média de todas as notas.

```
SELECT nota,  
(SELECT MIN(nota) FROM notas) AS menor,  
(SELECT MAX(nota) FROM notas) AS maior,
```

```
(SELECT ROUND(AVG(nota),2) FROM notas) AS media  
FROM notas;
```

2.10) Substituindo Valores Nulos para Formatar dados Retornados na Consulta

COALESCE – retorna o primeiro de seus argumentos que não for NULL. Só retorna NULL quando todos os seus argumentos forem NULL.

Uso: mudar o valor padrão cujo valor seja NULL.

```
create temp table nulos(  
    nulo1 int,  
    nulo2 int,  
    nulo3 int  
);
```

```
insert into nulos values (1,null,null);  
select * from nulos;  
select coalesce(nulo1, nulo2, nulo3) from nulos; - - Retorna 1, valor do campo nulo;  
select coalesce(nulo2, nulo3) from nulos; - - Retorna NULL, pois ambos são NULL.
```

Todos os valores da lista devem ser do mesmo tipo e nulos são ignorados.

Obs.: Ambas as funções acima não pertencem ao SQL standard, mas são uma extensão do PostgreSQL.

CONCATENANDO NULLs

A regra é: NULL se propaga. Qualquer que concatene com NULL gerará NULL.

STRING || NULL -- NULL

NULLIF

NULLIF(valor1, valor2)

A função NULLIF retorna o valor nulo se, e somente se, valor1 e valor2 forem iguais. Senão, retorna valor1. Pode ser utilizada para realizar a operação inversa do exemplo para COALESCE mostrado acima:

```
SELECT nullif(valor, '(nenhuma)') ...
```

Inserir nulo quando a cadeia de caracteres estiver vazia

Neste exemplo são utilizadas as funções NULLIF e TRIM para inserir o valor nulo na coluna da tabela quando a cadeia de caracteres passada como parâmetro para o comando INSERT preparado estiver vazia ou só contiver espaços.

```
CREATE TEMPORARY TABLE t (c1 SERIAL PRIMARY KEY, c2 TEXT);  
PREPARE inserir (TEXT)  
AS INSERT INTO t VALUES(DEFAULT, nullif(trim(' ' from $1),''));  
  
EXECUTE inserir('linha 1');
```

```
EXECUTE inserir('');
EXECUTE inserir('  ');
EXECUTE inserir(NULL);
```

```
\pset null nulo
SELECT * FROM t;
```

```
c1 | c2
---+-----
 1 | linha 1
 2 | nulo
 3 | nulo
 4 | nulo
```

Mais detalhes em: <http://pgdocptbr.sourceforge.net/pg80/functions-conditional.html>

2.11) Uso do Select (Desvendando o SELECT)

O SQL (Structured Query Language) é uma linguagem que visa padronizar e facilitar o gerenciamento de informações em bancos de dados relacionais. No entanto, a padronização dos comandos sofre com as implementações proprietárias, ou seja, cada fabricante de banco de dados, embora em sua maioria implemente ao menos em parte o SQL ANSI, implementa também características próprias que o torna diferente do SQL padrão, criando vícios de programação que acabam por prender o desenvolvedor a esse ou aquele banco.

Isso ocorre com praticamente todos os SGBDs, sejam eles comerciais, "OpenSources" ou "Freewares". Nesse artigo, trataremos instruções que operam de maneira similar na maioria dos SGBDs, entre eles o *MySQL*, *PostgreSQL*, *Firebird* e *Oracle*.

O SQL é uma linguagem bastante simples, com instruções de alto nível, mas que também permite a escrita de códigos complexos.

Nesse artigo, estaremos abordando exclusivamente o comando **SELECT**. Essa instrução é sem dúvida uma das mais utilizadas do SQL, sendo responsável por realizar consultas na base de dados.

A seguir temos um resumo da sintaxe do SELECT:

```
SELECT [ DISTINCT | ALL ] campos FROM
tabela1 [, tabela n]
[ JOIN condição ]
[ WHERE condição ]
[ GROUP BY expressão ]
[ HAVING condição ]
[ ORDER BY expressão [ASC | DESC] ]
```

A instrução SELECT não se restringe somente a sintaxe apresentada, mas será esta sintaxe o objeto do nosso estudo. Com exceção da cláusula JOIN, toda cláusula deve aparecer somente uma única vez no comando, ou seja, não há como usar um WHERE duas vezes para o mesmo SELECT, ou dois ORDER BY. Analisando a sintaxe apresentada, temos: Tudo que aparece entre colchetes é opcional, ou seja, você pode ou não utilizar essas cláusulas, sem que isso gere qualquer tipo de erro.

Campos: São as colunas retornadas pela instrução. Pode ser empregado um coringa "*" quando se deseja recuperar todos os campos.

A maioria dos SGBDs exige que a cláusula FROM esteja presente no SELECT.

DISTINCT | ALL: Indica se o select deve descartar informações repetidas, ou se deve trazer todas as linhas encontradas. A cláusula ALL é o padrão, podendo ser omitida, e recupera todas as linhas/registros.

Tabela1 – Tabela n São exemplos de nomes de tabelas, sendo que quando mais de uma tabela forem especificadas, os nomes devem ser separados por vírgula. Podemos também designar apelidos para as tabelas (alias), indicando-os logo após o nome de cada tabela. Os apelidos são válidos apenas para o select em questão.

Condição: Fator pelo qual a query irá filtrar os registros. Podemos utilizar operadores lógicos nas comparações como, por exemplo, OR (ou), AND (e), etc.

Expressão: São as informações pela qual a cláusula irá operar. Pode ser um campo, lista de campos, ou em alguns casos até mesmo uma condição.

Estaremos utilizando como exemplo neste artigo duas tabelas (pessoas e acessos), com a seguinte estrutura:

PESSOAS	ACESSOS
ID INTEGER	ID INTEGER
UNAME CHAR(15)	DATA DATETIME
NOME CHAR(50)	PESSOA CHAR(15)
IDADE INTEGER	QTDE INTEGER

Cláusula FROM

Basicamente a cláusula FROM é utilizada para indicar de onde será extraída a informação retornada pela query, ou seja, de quais views, tabelas, ou em alguns bancos até mesmo stored procedures. Exemplo de utilização:

```
SELECT * FROM PESSOAS;
```

O * (asterisco) indica que desejamos receber todos os campos da tabela PESSOAS.

Podemos no lugar do * (asterisco) indicar o nome do campo desejado, ou nomes dos campos separados por vírgula.

Também é possível designar apelidos para as tabelas (alias), indicando-os logo após o nome da tabela. Os apelidos são válidos apenas para o select em questão, e são associados às colunas recuperadas, determinando portanto à qual fonte de dados a coluna está associada. Exemplo:

```
SELECT UNAME as "USER NAME"  
FROM PESSOAS;
```

Obs.: No exemplo anterior, a cláusula AS é opcional e poderia ser removida.

A definição explícita de qual é a fonte de dados de uma determinada coluna tornase obrigatória quando mais de uma fonte de dados envolvida no select possuí campos com o mesmo nome. Abaixo mostramos um exemplo disso com duas tabelas:

```
SELECT pessoas.codigo,  
vendedores.codigo,  
vendedores.codpes,  
vendedores.nome  
FROM pessoas, vendedores  
WHERE pessoas.codigo = vendedores.  
codpes  
ORDER BY vendedores.nome;
```

Nesse exemplo, observe que para toda coluna sendo recuperada, foi especificado o nome da tabela a quem ela pertence. Isso é necessário pois o campo CODIGO existe tanto na tabela PESSOAS como na VENDEDORES.

Recomendo que seja sempre especificado qual é a fonte de dados de cada coluna, pois torna o select mais legível.

Ainda no mesmo exemplo, podemos também atribuir apelidos para as tabelas, desta forma enxugando o código e tornando-o ainda mais legível.

Abaixo segue o mesmo exemplo anterior, só que usando apelidos (alias):

```
SELECT PES.CODIGO,  
VEN.CODIGO,  
VEN.CODPES,  
VEN.NOME  
FROM PESSOAS PES, VENDEDORES VES  
WHERE PES.CODIGO = VEN.CODPES  
ORDER BY VEN.NOME;
```

Comentários

Comentários no SQL podem ser representados por

-- (dois sinais de menos, sem espaços) que definem que o restante da linha é um comentário, ou

/* e */ para comentar um trecho dentro do código SQL, que pode inclusive se estender por mais de uma linha.

Exemplo:

```
SELECT *  
FROM PESSOAS -- Aqui é comentário  
WHERE 1;  
OU  
SELECT *  
FROM /* comentário */ PESSOAS  
WHERE 1;
```

Dica

Os SGBDs citados nesse artigo permitem cálculos diretamente pelo SELECT, sem o uso da cláusula FROM, ou seja o uso do SELECT para obter resultado de cálculos aritméticos básicos. Como no exemplo abaixo:

```
SELECT 10 - 2;  
10-2  
8
```

Cláusula JOIN

A cláusula JOIN é empregada para permitir que um mesmo select recupere informações de mais de uma fonte de dados (tabelas, views, etc.). Em geral, as tabelas referenciadas possuem algum tipo de relacionamento entre elas, através de um ou mais campos que definam a ligação entre uma tabela e a outra (integridade referencial).

Há duas maneiras de implementar um join:

- A primeira é chamada de **non-ANSI** ou estilo **theta**, que utiliza a cláusula WHERE para efetuar a junção de tabelas;
- A segunda é chamada de **ANSI Join**, e é baseada no uso da cláusula JOIN propriamente dita.

Simple ligação

Um exemplo de JOIN em estilo ANSI:

```
SELECT p.uname, p.nome, a.qtde
from PESSOAS p
CROSS JOIN ACESSOS a;
```

Um exemplo de JOIN em estilo *theta*:

```
SELECT p.uname, p.nome, a.qtde
from PESSOAS p, ACESSOS a;
```

Note que na chamada ANSI utilizamos CROSS JOIN, que é a sintaxe utilizada para recuperar todos os registros das tabelas ligadas, formando um produto cartesiano. É basicamente um INNER JOIN (citado adiante) sem condições.

Tipos de junções

Inner Joins

Somente as linhas/registros que satisfaçam a ligação determinada pelo JOIN serão recuperados pelo select, sendo assim, os registros que **não** se enquadram no relacionamento definido pelo join **não serão recuperados**.

Um exemplo de INNER JOIN em estilo ANSI:

```
SELECT p.uname,
p.nome,
a.qtde
from PESSOAS p
INNER JOIN ACESSOS a on p.uname=a.pessoa order by p.uname;
```

O mesmo JOIN em estilo theta:

```
SELECT p.uname,
p.nome,
a.qtde
from PESSOAS p, ACESSOS a WHERE p.uname = a.pessoa order by p.uname;
```

Left Joins

Através do uso do **LEFT**, todos os registros na tabela à esquerda da query serão listados, independente de terem ou não registros relacionados na tabela à direita. Nesse caso, as colunas relacionadas com a tabela da direita voltam nulos (NULL).

Um exemplo de uso LEFT JOIN:

```
SELECT p.uname,
p.nome,
a.pessoa,
a.qtde
from PESSOAS p
LEFT JOIN ACESSOS a on p.uname=a.pessoa order by p.uname;
```

No exemplo acima, todos os registros da tabela PESSOAS serão listados, independente de terem ou não registros associados na tabela ACESSOS. Caso não existam registros associados na tabela ACESSOS, os campos *a.pessoa* e *a.qtde* retornarão NULL.

Right joins

É o inverso do Left Join, ou seja, todos os registros da tabela à direita serão listados, independente de terem ou não registros relacionados na tabela à esquerda.

Um exemplo de uso RIGHT JOIN:

```
SELECT p.uname,  
p.nome,  
a.pessoa,  
a.qtde  
from pessoas p  
RIGHT JOIN acessos a on p.uname=a.pessoas order by p.uname;
```

Ou seja, todos os registros da tabela ACESSOS serão listados, e caso não haja correspondentes na tabela PESSOAS, a query devolve NULL para os campos p.uname e p.nome.

Cláusula WHERE

A cláusula WHERE permite aplicar filtros sobre as informações vasculhadas pelo SELECT. É extremamente abrangente e permite gerar condições complexas de pesquisa. Os operadores aceitos pela cláusula WHERE são:

=, >, <, <>, >=, <= e BETWEEN.

Há também a possibilidade de se utilizar operadores de proximidade como o LIKE, que permite realizar buscas por apenas uma parte de um string. Pode-se negar uma comparação com o operador **NOT**, bem como utilizar operadores lógicos (**OR** ou **AND**).

Outros operadores suportados são:

- **IN**: verificar se um valor está contido em um grupo de valores;
- **EXISTS**: verifica se um valor existe no resultset retornado por um select;

Vale a pena lembrar que podemos utilizar parênteses, determinando a ordem das condições a serem aplicadas.

NULO ou NÃO NULO, eis a questão...

NULL determina um estado e não um valor, por isso deve ser tratado de maneira especial. Quando queremos saber se um campo é nulo, a comparação a ser feita é "campo **is null**" e não "campo = null", sendo que essa última sempre retornará FALSO.

Do mesmo modo, para determinar se um campo não é nulo, usamos "campo **is not null**".

Exemplos de utilização

Adiante temos exemplos de aplicação de WHEREs:

```
SELECT *  
from PESSOAS  
WHERE (IDADE >= 90);
```

Filtra todos os registros da tabela PESSOAS no qual o campo IDADE é maior ou igual a 90.

```
SELECT *  
FROM PESSOAS  
WHERE (IDADE >= 1 AND IDADE <= 17);
```

ou

```
SELECT *
```

```
FROM PESSOAS  
WHERE IDADE BETWEEN 1 AND 17;
```

Lista todos os registros da tabela PESSOAS cujo conteúdo do campo IDADE seja maior e igual a 1 e menos e igual a 17.

```
SELECT *  
from PESSOAS  
WHERE NOME LIKE 'JO%';
```

ou

```
SELECT *  
from PESSOAS  
WHERE NOME STARTING WITH 'JO'; -- No Firebird
```

Retorna todos os campos e todos os registros da tabela pessoa que possuam as letras "JO" como iniciais do nome.

```
SELECT *  
FROM PESSOAS  
WHERE NOME LIKE '%AN%';
```

ou

```
SELECT *  
FROM PESSOAS  
WHERE NOME CONTAINING 'AN';
```

No caso anterior, retorna todos os registros que possuam no campo nome as letras AN, seja no começo, meio ou fim do campo.

Outro coringa que pode ser utilizado é o "_" (underline). Podemos utilizá-lo quando desejarmos mascarar uma ou mais letras.

Por exemplo:

```
SELECT *  
FROM PESSOAS  
WHERE NOME LIKE "LUI_";
```

Nesse caso, obteremos todos os registros cujo campo NOME possui as três primeiras letras LUI, sendo que a quarta letra pode ser qualquer caractere.

```
SELECT *  
FROM ACESSOS  
WHERE QTDE IS NULL;
```

Retorna todos os campos e todos os registros onde o campo QTDE é NULL.

```
SELECT PE.UNAME,  
PE.NOME  
FROM PESSOAS PE  
WHERE EXISTS (SELECT * from ACESSOS  
WHERE PESSOA = PE.UNAME);
```

Retorna os campos UNAME e NOME de todos os registros da tabela PESSOAS que possuem registros na tabela ACESSOS. O uso da cláusula EXISTS é na verdade uma junção com uma subquerie (segundo select). Nesse caso, somente serão exibidos os registros da query que atendam a condição da subquery. Caso um registro na tabela pessoas não possua um registro correspondente na tabela ACESSOS, ele não será recuperado.

```
SELECT *  
FROM PESSOAS  
WHERE IDADE IN (32,24);
```

Serão retornados os registros cuja idade é 32 ou 24. Poderia ser implementada por um OR, mas quando a lista de campos na condição é extensa, o OR pode deixar a instrução muito longa e de difícil compreensão.

Dica

Pode-se utilizar o NOT em conjunto com praticamente todos os operadores de condição. Exemplo: NOT com o BETWEEN (NOT BETWEEN), com o LIKE (NOT LIKE), com o IN (NOT IN) e outras tantas maneiras. Dessa forma, as possibilidades de filtragem aumentam significativamente.

Cláusula GROUP BY

As cláusulas GROUP BY e HAVING são geralmente utilizadas quando utilizamos funções de agrupamento. As principais funções de agrupamento são:

SUM Soma todos os valores da coluna informada
MAX Retorna o maior valor da coluna especificada.
MIN Retorna o menor valor da coluna informada.
COUNT Retorna o número de registros da tabela.
AVG Retorna a média aritmética dos valores da coluna informada.

Você Sabia?

Os registros cujo campo é NULL são desprezados quando se utiliza uma função de agregação.

Exemplos:

```
SELECT SUM(QTDE) FROM ACESSOS;
```

Query devolve a somatória do campo QTDE da tabela ACESSOS.

```
SELECT MAX(QTDE) FROM ACESSOS;
```

Query devolve o maior valor do campo QTDE da tabela ACESSOS.

```
SELECT NOME,  
COUNT( AC.QTDE)  
FROM PESSOAS PE, ACESSOS AC  
WHERE PE.UNAME = AC.PESSOA  
GROUP BY (AC.PESSOA);
```

Você Sabia?

As funções SUM e AVG somente podem ser utilizadas em campos com tipos numéricos. As funções MAX e MIN podem ser utilizados em numéricos, data e com caracteres também, e COUNT em campos de qualquer tipo de dado.

Quando um mesmo select recupera tanto campos individuais e valores agrupados através das funções de agrupamento, é necessário utilizar o *group by* listando nele as colunas individuais, para que o select possa ser executado.

Cláusula HAVING

A cláusula HAVING aplica um filtro sobre o resultado de um GROUP BY e, portanto, é utilizada em conjunto com o mesmo. Ela não interfere no resultado obtido na Query, pois age após a totalização, ou seja, não altera resultados de totalizações, apenas filtra o que será exibido. Exemplo:

```
SELECT PESSOA, SUM(QTDE) AS TOTAG
```

```
FROM ACESSOS  
GROUP BY PESSOA  
HAVING TOTAG > 5;
```

ou

```
SELECT PESSOA, SUM(QTDE) AS TOTAG  
FROM ACESSOS  
GROUP BY PESSOA  
HAVING SUM(QTDE) > 5;
```

A query acima agrupa os registros pelo campo PESSOA, depois soma os conteúdos do campo QTDE retornando-os com o alias TOTAG (por pessoa), e somente exibe os que possuem TOTAG maior que 5.

Cláusula ORDER BY

Como o próprio nome sugere, essa cláusula ordena informações obtidas em uma query, de forma [ASC]endente (menor para o maior, e da letra A para Z), que é o padrão e pode ser omitida, ou de maneira [DESC]endente. O NULL, por ser um estado e não um valor, aparece antes de todos na ordenação. Alguns SGBDs permitem especificar se os nulls devem aparecer no início ou fim dos registros, como por exemplo o Firebird, com o uso das cláusulas **NULLS FIRST** e **NULLS LAST**.

Exemplo:

```
PESSOAS  
ORDER BY PESSOA;
```

A query acima retorna os registros da tabela PESSOAS ordenados por PESSOA de maneira ascendente. Pode-se ordenar por "n" colunas, ou seja, fazer uma sub-ordenação. Isso é feito indicando os campos separados por vírgula, conforme a query abaixo:

```
SELECT *  
FROM ACESSOS  
ORDER BY PESSOA, QTDE;
```

Desta maneira, a query retornará o resultado ordenado por PESSOA e sub-ordenado pela QTDE. Pode-se também montar queries onde se ordena de maneira ascendente uma coluna, e de maneira descendente outra. Exemplo:

```
SELECT *  
FROM ACESSOS  
ORDER BY PESSOA DESC, QTDE ASC;
```

A query retorna todos os registros da tabela ACESSOS ordenados pelo campo PESSOA de maneira descendente, e pelo campo QTDE na forma ascendente.

O ORDER BY também permite ordenações referenciando as colunas como números. Assim, a primeira coluna retornada no query é a 1, a segunda é a 2, e assim sucessivamente.

Desta forma, podemos escrever códigos conforme o abaixo:

```
SELECT CAMPO1, CAMPO2, CAMPO3, CAMPO4  
FROM ACESSOS  
ORDER BY 3;
```

No exemplo, a ordenação será através da coluna CAMPO3.

Dicas adicionais

Existem recursos adicionais que podem ser utilizados nos SELECTs. Geralmente são funções e variáveis que podem nos poupar bastante trabalho. Abaixo listo algumas funções interessantes, presentes na maioria

dos bancos de dados, sejam de forma nativa, seja na forma de UDFs:

Funções ANSI para tratar Strings:

Lower(campo) - Devolve o conteúdo do campo em minúsculos

Upper(campo) - Devolve o conteúdo do campo em maiúsculos

Character_Length(campo) - Devolve o tamanho ocupado em bytes do campo.

Position(string1 IN string2) - Busca pelo String1 na String2. Se encontrar devolve o offset de posição, do contrário devolve 0 (zero). O LIKE internamente em alguns bancos utiliza essa função.

Concat(String1, String2, ..., StringN) - Devolve um String que é a concatenação dos strings informados.

Exemplo:

```
SELECT CONCAT( UNAME, '-', NOME ) AS  
"NOME" FROM PESSOAS;
```

Funções para implementar rotinas de segurança no banco:

System_user() - Devolve o nome do usuário do sistema operacional para o servidor SQL.

Session_user() - Devolve um string com a autorização da sessão SQL atual.

User() - Devolve o nome do usuário ativo. No Firebird é CURRENT_USER.

Constantes de data e hora:

Current_Date - Devolve a data atual

Current_Time - Devolve a hora atual Existem otimizações que poderiam ser feitas nas tabelas utilizadas em nossos exemplos como, por exemplo, a implementação de índices. Em um próximo artigo iremos falar sobre otimizações do banco.

Diferenças entre SGBDs

Cada SGBD possui sintaxes próprias, que adicionam recursos e facilidades ao SELECT padrão.

Autor - Luiz Paulo de Oliveira Santos na DBFree Magazine 002

3) Criação e Manipulação de Tabelas

- 3.1) Visualizando a estrutura de tabelas criadas
- 3.2) Entendendo as colunas de sistema
- 3.3) Sintaxe de criação de tabelas
- 3.4) Entendendo o comando Alter Table
- 3.5) Alterando tabelas e colunas
- 3.6) Comentários em objetos
- 3.7) Eliminando tabelas

3.1) Visualizando a estrutura de tabelas criadas

Para visualizar a estrutura de tabelas criadas podemos usar:

- Metacomando \d nometabela no psql ou então
- Selecionar a tabela no PGAdmin

3.2) Entendendo as colunas de sistema

Vejamos alguns exemplos de colunas das tabelas de sistema.

Acesse o psql em qualquer banco e execute:

```
\dS
```

```
\d pg_database
```

Vejamos algumas das colunas (campos):

```
dml=# \d pg_database
```

```
Table "pg_catalog.pg_database"
```

```
Column      | Type      | Modifiers
```

```
-----+-----+-----
datname      | name      | not null
datdba       | oid       | not null
encoding     | integer   | not null
datistemplate | boolean   | not null
dataallowconn | boolean   | not null
datconnlimit | integer   | not null
```

datame – representa o nome do banco

encoding – a codificação do banco

datistemplate – se o banco é ou não um template

3.3) Sintaxe de criação de tabelas

Nada melhor que o psql para nos informar toda a sintaxe completa da criação de tabelas:

dml=# \h create table

Command: CREATE TABLE

Description: define a new table

Syntax:

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE table_name ( [
    { column_name data_type [ DEFAULT default_expr ] [ column_constraint [ ... ] ]
    | table_constraint
    | LIKE parent_table [ { INCLUDING | EXCLUDING } { DEFAULTS | CONSTRAINTS | INDEXES } ] ... }
    [, ... ]
] )
[ INHERITS ( parent_table [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace ]
```

where column_constraint is:

```
[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  UNIQUE index_parameters |
  PRIMARY KEY index_parameters |
  CHECK ( expression ) |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
  [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

and table_constraint is:

```
[ CONSTRAINT constraint_name ]
{ UNIQUE ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters |
  CHECK ( expression ) |
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn [, ... ] ) ]
  [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

index_parameters in UNIQUE and PRIMARY KEY constraints are:

```
[ WITH ( storage_parameter [= value] [, ... ] ) ]  
[ USING INDEX TABLESPACE tablespace ]
```

Alguns exemplos de criação de tabelas

```
create database dba_projeto;  
\c dba_projeto
```

```
create table clientes (  
    cpf varchar(11) primary key,  
    nome varchar(45) not null,  
    telefone varchar(11),  
    email varchar(45),  
    data_nasc date not null  
);  
create table produtos (  
    codigo serial primary key,  
    descricao varchar(40) not null,  
    quantidade int2 not null,  
    data_compra date not null  
);
```

```
create table pedidos (  
    codigo serial primary key,  
    cpf_cliente varchar(11),  
    codigo_produto int4,  
    quantidade int2,  
    data date,  
    valor decimal(12,2)  
);
```

Agora criando a tabela pedidos com algumas foreign keys:

```
create table pedidos (  
    codigo serial primary key,  
    cpf_cliente varchar(11),  
    codigo_produto int4,  
    quantidade int2,  
    data date,  
    valor decimal(12,2),  
    CONSTRAINT pedidos_cli_fk FOREIGN KEY (cpf_cliente) REFERENCES clientes (cpf),  
    CONSTRAINT pedidos_prod_fk FOREIGN KEY (codigo_produto) REFERENCES produtos  
    (codigo)  
);
```

3.4) Entendendo o comando Alter Table

dml=# \h alter table

Command: ALTER TABLE

Description: change the definition of a table

Syntax:

ALTER TABLE [ONLY] name [*]

action [, ...]

ALTER TABLE [ONLY] name [*]

RENAME [COLUMN] column TO new_column

ALTER TABLE name

RENAME TO new_name

ALTER TABLE name

SET SCHEMA new_schema

where action is one of:

ADD [COLUMN] column type [column_constraint [...]]

DROP [COLUMN] column [RESTRICT | CASCADE]

ALTER [COLUMN] column TYPE type [USING expression]

ALTER [COLUMN] column SET DEFAULT expression

ALTER [COLUMN] column DROP DEFAULT

ALTER [COLUMN] column { SET | DROP } NOT NULL

ALTER [COLUMN] column SET STATISTICS integer

ALTER [COLUMN] column SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }

ADD table_constraint

DROP CONSTRAINT constraint_name [RESTRICT | CASCADE]

DISABLE TRIGGER [trigger_name | ALL | USER]

ENABLE TRIGGER [trigger_name | ALL | USER]

ENABLE REPLICA TRIGGER trigger_name

ENABLE ALWAYS TRIGGER trigger_name

DISABLE RULE rewrite_rule_name

ENABLE RULE rewrite_rule_name

ENABLE REPLICA RULE rewrite_rule_name

ENABLE ALWAYS RULE rewrite_rule_name

CLUSTER ON index_name

SET WITHOUT CLUSTER

SET WITHOUT OIDS

SET (storage_parameter = value [, ...])

RESET (storage_parameter [, ...])

INHERIT parent_table

NO INHERIT parent_table

OWNER TO new_owner

SET TABLESPACE new_tablespace

3.5) Alterando tabelas e colunas

Adicionar campo, remover campo, adicionar constraint, remover constraint, alterar valor default, renomear campo, renomear tabela, alterar tipo de dado de campo (>=8.0).

Adicionar Um Campo

```
ALTER TABLE tabela ADD COLUMN campo tipo;  
ALTER TABLE produtos ADD COLUMN descricao text;
```

Remover Campo

```
ALTER TABLE tabela DROP COLUMN campo;  
ALTER TABLE produtos DROP COLUMN descricao;  
ALTER TABLE produtos DROP COLUMN descricao CASCADE; -- Cuidado com CASCADE
```

Adicionar Constraint

```
ALTER TABLE tabela ADD CONSTRAINT nome;  
ALTER TABLE produtos ADD COLUMN descricao text CHECK (descricao <> "");  
ALTER TABLE produtos ADD CHECK (nome <> "");  
ALTER TABLE produtos ADD CONSTRAINT unique_cod_prod UNIQUE (cod_prod);  
ALTER TABLE produtos ADD FOREIGN KEY (cod_produtos) REFERENCES grupo_produtos;  
ALTER TABLE produtos ADD CONSTRAINT vendas_fk FOREIGN KEY (cod_produtos)  
REFERENCES produtos (codigo);
```

Remover Constraint

```
ALTER TABLE tabela DROP CONSTRAINT nome;  
ALTER TABLE produtos DROP CONSTRAINT produtos_pk;  
ALTERAR VALOR DEFAULT DE CAMPO:
```

Mudar Tipo de Dados de Campo (Só >=8.0):

```
ALTER TABLE tabela ALTER COLUMN campo TYPE tipo;  
ALTER TABLE produtos ALTER COLUMN preco TYPE numeric(10,2);  
ALTER TABLE produtos ALTER COLUMN data TYPE DATE USING CAST (data AS DATE);
```

Mudar Nome De Campo

```
ALTER TABLE tabela RENAME COLUMN campo_atual TO campo_novo;  
ALTER TABLE produtos RENAME COLUMN cod_prod TO cod_produto;
```

Setar/Remover Valor Default de Campo

```
ALTER TABLE tabela ALTER COLUMN campo SET DEFAULT valor;  
ALTER TABLE produtos ALTER COLUMN cod_prod SET DEFAULT 0;  
ALTER TABLE produtos ALTER COLUMN preco SET DEFAULT 7.77;  
ALTER TABLE tabela ALTER COLUMN campo DROP DEFAULT;  
ALTER TABLE produtos ALTER COLUMN preco DROP DEFAULT;
```

Adicionar/Remover NOT NULL

```
ALTER TABLE produtos ALTER COLUMN cod_prod SET NOT NULL;  
ALTER TABLE produtos ALTER COLUMN cod_prod DROP NOT NULL;
```

Renomear Tabela

```
ALTER TABLE tabela RENAME TO nomenovo;  
ALTER TABLE produtos RENAME TO equipamentos;
```

Adicionar Constraint (Restrição)

```
ALTER TABLE produtos ADD CONSTRAINT produtos_pk PRIMARY KEY (codigo);  
ALTER TABLE vendas ADD CONSTRAINT vendas_fk FOREIGN KEY (codigo) REFERENCES  
produtos(codigo_produto);  
ALTER TABLE vendas ADD CONSTRAINT vendas_fk FOREIGN KEY (codigo) REFERENCES  
produtos; -- Neste caso usa a chave primária da tabela produtos
```

Remover Constraint (Restrição)

```
ALTER TABLE produtos DROP CONSTRAINT produtos_pk;  
ALTER TABLE vendas DROP CONSTRAINT vendas_fk;
```

3.6) Comentários em objetos

Para comentar objetos no PostgreSQL usamos o comando COMMENT, que não é compatível com o SQL, mas pertence ao PostgreSQL.

dml=# \h comment

Command: COMMENT

Description: define or change the comment of an object

Syntax:

COMMENT ON

{

TABLE object_name |

COLUMN table_name.column_name |

AGGREGATE agg_name (agg_type [, ...]) |

CAST (sourcetype AS targettype) |

CONSTRAINT constraint_name ON table_name |

```

CONVERSION object_name |
DATABASE object_name |
DOMAIN object_name |
FUNCTION func_name ( [ [ argmode ] [ argname ] argtype [, ...] ] ) |
INDEX object_name |
LARGE OBJECT large_object_oid |
OPERATOR op (leftoperand_type, rightoperand_type) |
OPERATOR CLASS object_name USING index_method |
OPERATOR FAMILY object_name USING index_method |
[ PROCEDURAL ] LANGUAGE object_name |
ROLE object_name |
RULE rule_name ON table_name |
SCHEMA object_name |
SEQUENCE object_name |
TABLESPACE object_name |
TEXT SEARCH CONFIGURATION object_name |
TEXT SEARCH DICTIONARY object_name |
TEXT SEARCH PARSER object_name |
TEXT SEARCH TEMPLATE object_name |
TRIGGER trigger_name ON table_name |
TYPE object_name |
VIEW object_name
} IS 'text'

```

Resumindo:

COMMENT ON TIPO nomeobjeto IS 'Comentario';

Exemplo:

COMMENT ON TABLE clientes IS 'Comentario da Tabela Clientes';

No PGAdmin, ao selecionar a tabela vemos o comentário sobre a mesma.

3.7) Eliminando tabelas

Para eliminar tabelas usamos o comando SQL '**drop**'. Vejamos sua sintaxe:

dml=# \h drop table

Command: DROP TABLE

Description: remove a table

Syntax:

```
DROP TABLE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Bem simples e sem muitas opções.

IF EXISTS – Não dispara erro, caso a tabela não exista. Útil para scripts.

CASCADE – força a exclusão, excluindo automaticamente os objetos que dependem desta tabela, como views.

RESTRICT – Opção padrão, que se recusa a remover a tabela caso exista algum objeto dependente da tabela.

Exemplos:

DROP TABLE clientes;

DROP TABLE IF EXISTS clientes;
DROP TABLE clientes CASCADE;

4) Trabalhando com Conjuntos de Dados

Utilizando a União, Intersecção e Subtração de conjuntos de dados

Combinação de consultas

Pode-se combinar os resultados de duas consultas utilizando as operações de conjunto união, interseção e diferença [1] [2] [3] [4] [5] . A sintaxe é

```
consulta1 UNION [ALL] consulta2  
consulta1 INTERSECT [ALL] consulta2  
consulta1 EXCEPT [ALL] consulta2
```

onde consulta1 e consulta2 são consultas que podem utilizar qualquer uma das funcionalidades mostradas até aqui. As operações de conjuntos também podem ser aninhadas ou encadeadas. Por exemplo:

```
consulta1 UNION consulta2 UNION consulta3
```

significa, na verdade,
(consulta1 UNION consulta2) UNION consulta3

Efetivamente, UNION anexa o resultado da consulta2 ao resultado da consulta1 (embora não haja garantia que esta seja a ordem que as linhas realmente retornam). Além disso, são eliminadas do resultado as linhas duplicadas, do mesmo modo que no DISTINCT, a não ser que seja utilizado UNION ALL.

INTERSECT retorna todas as linhas presentes tanto no resultado da consulta1 quanto no resultado da consulta2. As linhas duplicadas são eliminadas, a não ser que seja utilizado INTERSECT ALL. EXCEPT retorna todas as linhas presentes no resultado da consulta1, mas que não estão presentes no resultado da consulta2 (às vezes isto é chamado de diferença entre duas consultas). Novamente, as linhas duplicadas são eliminadas a não ser que seja utilizado EXCEPT ALL.

Para ser possível calcular a união, a interseção, ou a diferença entre duas consultas, as duas consultas devem ser "compatíveis para união", significando que ambas devem retornar o mesmo número de colunas, e que as colunas correspondentes devem possuir tipos de dado compatíveis, conforme descrito na Seção 10.5.

Nota: O exemplo abaixo foi escrito pelo tradutor, não fazendo parte do manual original.

Exemplo. Linhas diferentes em duas tabelas com definições idênticas

Este exemplo mostra a utilização de EXCEPT e UNION para descobrir as linhas diferentes de duas tabelas semelhantes.

```
CREATE TEMPORARY TABLE a (c1 text, c2 text, c3 text);  
INSERT INTO a VALUES ('x', 'x', 'x');
```



```
INSERT INTO a VALUES ('x', 'x', 'y'); -- nas duas tabelas
INSERT INTO a VALUES ('x', 'y', 'x');
```

```
CREATE TEMPORARY TABLE b (c1 text, c2 text, c3 text);
INSERT INTO b VALUES ('x', 'x', 'y'); -- nas duas tabelas
INSERT INTO b VALUES ('x', 'x', 'y'); -- nas duas tabelas
INSERT INTO b VALUES ('x', 'y', 'y');
INSERT INTO b VALUES ('y', 'y', 'y');
INSERT INTO b VALUES ('y', 'y', 'y');
```

-- No comando abaixo só um par ('x', 'x', 'y') é removido do resultado
 -- Este comando executa no DB2 8.1 sem alterações.

```
(SELECT 'a-b' AS dif, a.* FROM a EXCEPT ALL SELECT 'a-b', b.* FROM b)
UNION ALL
(SELECT 'b-a', b.* FROM b EXCEPT ALL SELECT 'b-a', a.* FROM a);
```

```
dif | c1 | c2 | c3
-----+-----+-----+-----
a-b | x | x | x
a-b | x | y | x
b-a | x | x | y
b-a | x | y | y
b-a | y | y | y
b-a | y | y | y
(6 linhas)
```

-- No comando abaixo são removidas todas as linhas ('x', 'x', 'y'),
 -- e só é mostrada uma linha ('y', 'y', 'y') no resultado
 -- Este comando executa no DB2 8.1 sem alterações.
 -- Este comando executa no Oracle 10g trocando EXCEPT por MINUS.

```
(SELECT 'a-b' AS dif, a.* FROM a EXCEPT SELECT 'a-b', b.* FROM b)
UNION
(SELECT 'b-a', b.* FROM b EXCEPT SELECT 'b-a', a.* FROM a);
```

```
dif | c1 | c2 | c3
-----+-----+-----+-----
a-b | x | x | x
a-b | x | y | x
b-a | x | y | y
b-a | y | y | y
(4 linhas)
```

Notas

- [1] Dados dois conjuntos A e B: chama-se diferença entre A e B o conjunto formado pelos elementos de A que não pertencem a B; chama-se interseção de A com B o conjunto formado pelos elementos comuns ao conjunto A e ao conjunto B; chama-se união de A com B o conjunto formado pelos elementos que pertencem a A ou B. Edwaldo Bianchini e Herval

Paccola - Matemática - Operações com conjuntos. (N. do T.)

- [2] SQL Server — UNION combina os resultados de duas ou mais consultas em um único conjunto de resultados que inclui todas as linhas que pertencem à união das consultas. A operação UNION é diferente de utilizar junções que combinam colunas de duas tabelas. As regras básicas para combinar conjuntos de resultados de duas consultas utilizando UNION são as seguintes: a) O número e a ordem das colunas devem ser os mesmos em todas as consultas; b) Os tipos de dado devem ser compatíveis. UNION ALL inclui as linhas duplicadas. SQL Server 2005 Books Online — UNION (Transact-SQL) (N. do T.)
- [3] SQL Server — EXCEPT e INTERSECT retornam valores distintos comparando os resultados de duas consultas. EXCEPT retorna todos os valores distintos da consulta à esquerda que não se encontram na consulta à direita. INTERSECT retorna todos os valores distintos retornados pelas consultas à esquerda e a direita do operando INTERSECT. SQL Server 2005 Books Online — EXCEPT and INTERSECT (Transact-SQL) (N. do T.)
- [4] Oracle — Os operadores de conjunto combinam os resultados de duas consultas componentes em um único resultado. As consultas que contém operadores de conjunto são chamadas de consultas compostas. Os operadores de conjunto disponíveis são: UNION, UNION ALL, INTERSECT e MINUS (equivalente ao EXCEPT). Oracle® Database SQL Reference 10g Release 1 (10.1) Part Number B10759-01. (N. do T.)
- [5] DB2 — Os operadores de conjunto UNION, EXCEPT e INTERSECT correspondem aos operadores relacionais união, diferença e interseção. DB2 Version 9 for Linux, UNIX, and Windows (N. do T.)

Detalhes em: <http://pgdocptbr.sourceforge.net/pg80/queries-union.html>

Artigo do Juliano : <http://imasters.uol.com.br/artigo/954:>

POSTGRESQL Interagindo com banco de dados

Agora que você já tem o banco de dados PostgreSQL instalado e rodando, e já se identificou com alguma ferramenta para manipulação das bases de dados, vamos começar a interagir com o banco de dados. A intenção não é ensinar SQL, mas sim, mostrar como verificar no PostgreSQL determinadas funcionalidades existentes em outros bancos de dados, bem como algumas de suas particularidades.

Primeiro, criaremos 3 tabelas:

```
CREATE TABLE cliente (  
  cliente_id SERIAL NOT NULL,  
  desde    DATE NULL,  
  nome     VARCHAR(60) NULL,  
  CONSTRAINT XPKcliente  
    PRIMARY KEY (cliente_id)  
);
```

```
CREATE TABLE venda (  
  venda_id SERIAL NOT NULL,
```

```
cliente_id INT4 NOT NULL,  
data    DATE NULL,  
valor    NUMERIC(15,2) NULL,  
produto  VARCHAR(30) NULL,  
CONSTRAINT XPKvenda  
    PRIMARY KEY (venda_id),  
CONSTRAINT cliente_vendas  
    FOREIGN KEY (cliente_id)  
        REFERENCES cliente  
);  
  
CREATE INDEX XIF1venda ON venda  
(  
cliente_id  
);  
  
CREATE TABLE troca (  
troca_id SERIAL NOT NULL,  
cliente_id INT4 NOT NULL,  
data    DATE NULL,  
produto  VARCHAR(30) NULL,  
troca    VARCHAR(30) NULL,  
CONSTRAINT XPKtroca  
    PRIMARY KEY (troca_id),  
CONSTRAINT cliente_trocas  
    FOREIGN KEY (cliente_id)  
        REFERENCES cliente  
);  
  
CREATE INDEX XIF1troca ON troca  
(  
cliente_id  
);
```

Em seguida, iremos popular as tabelas com alguns dados:

```
INSERT INTO cliente (desde,nome)  
VALUES ('2002-01-12','Paulo Santos Macedo');  
INSERT INTO cliente (desde,nome)  
VALUES ('2001-07-21','Márcia Barbosa');  
INSERT INTO cliente (desde,nome)  
VALUES ('2000-02-27','Anderson Marques');  
INSERT INTO cliente (desde,nome)  
VALUES ('2003-01-12','Daniela Freitas');  
INSERT INTO cliente (desde,nome)  
VALUES ('2003-01-15','Ana Júlia Cabral');
```

```

INSERT INTO venda (cliente_id, data, valor, produto)
VALUES (1,'2002-12-23',16,'Relógio');
INSERT INTO venda (cliente_id, data, valor, produto)
VALUES (3,'2002-12-23',110,'Mala Viagem');
INSERT INTO venda (cliente_id, data, valor, produto)
VALUES (1,'2002-12-21',10,'Saca-rolha');
INSERT INTO venda (cliente_id, data, valor, produto)
VALUES (4,'2002-12-20',32,'Fichário');
INSERT INTO venda (cliente_id, data, valor, produto)
VALUES (2,'2002-12-23',28,'Despertador');
INSERT INTO venda (cliente_id, data, valor, produto)
VALUES (3,'2002-12-23',43,'Mochila');
INSERT INTO venda (cliente_id, data, valor, produto)
VALUES (2,'2002-12-21',22,'Rádio');
INSERT INTO venda (cliente_id, data, valor, produto)
VALUES (4,'2002-12-20',12,'Lapiseira');
INSERT INTO troca (cliente_id, data, produto, troca)
VALUES (1,'2003-02-12','Relógio','Relógio');
INSERT INTO troca (cliente_id, data, produto, troca)
VALUES (3,'2003-02-13','Mala Viagem','Maleta Executivo');
INSERT INTO troca (cliente_id, data, produto, troca)
VALUES (1,'2003-02-08','Saca-rolha','Garrafa Térmica');
INSERT INTO troca (cliente_id, data, produto, troca)
VALUES (4,'2003-02-09','Fichário','Fichário');

```

Agora sim, vamos começar.

COMBINANDO CONSULTAS

Um problema encontrado quando escrevemos consultas em SQL é que, em determinados casos, estas consultas devem ser combinadas para obter o resultado desejado, pois, através de uma consulta única e direta, talvez não seja possível obtê-los. Combinar consultas significa que mais de uma instrução SELECT estará sendo usada na consulta. O resultado desta combinação se dará através das seguintes palavras-chave:

UNION	utilizada para adicionar (unir) os resultados das instruções SELECT apresentadas na consulta
INTERSECT	retorna somente os dados comuns resultantes das instruções SELECT apresentadas na consulta
EXCEPT	mostra todos os dados que não estão incluídos na segunda instrução SELECT apresentada na consulta

Vamos aos exemplos:

Pelos dados iniciais de exemplo, vimos que existem 4 clientes que adquiriram produtos para o natal do ano passado e, alguns deles, tiveram que fazer a troca de alguns produtos por um motivo qualquer.

Queremos saber então, quais clientes NÃO precisaram fazer nenhuma troca:

```
SELECT cliente.nome FROM venda JOIN cliente ON cliente.cliente_id = venda.cliente_id
EXCEPT
SELECT cliente.nome FROM troca JOIN cliente ON cliente.cliente_id = troca.cliente_id;
cliente_id;
nome
-----
Márcia Barbosa
(1 row)
```

...e quais PRECISARAM fazer alguma troca:

```
SELECT cliente.nome FROM venda JOIN cliente ON cliente.cliente_id = venda.cliente_id
INTERSECT
SELECT cliente.nome FROM troca JOIN cliente ON cliente.cliente_id = troca.cliente_id;
cliente_id;
nome
-----
Anderson Marques
Daniela Freitas
Paulo Santos Macedo
(3 rows)
```

Agora, queremos saber quais produtos foram movimentados, ou seja, tanto faz se foram vendidos ou trocados:

```
SELECT venda.produto FROM venda
UNION
SELECT troca.troca FROM troca;
produto
-----
Despertador
Fichário
Garrafa Térmica
Lapiseira
Mala Viagem
Maleta Executivo
Mochila
Rádio
Relógio
Saca-rolha
```

(10 rows)

Observações:

A arquitetura da base de dados influi diretamente sobre como serão criadas as consultas, ou seja, quais tabelas contém certos dados e, qual o relacionamento entre eles;

Obviamente, existem muitas formas de obter o mesmo resultado, as maneiras apresentadas aqui, são algumas delas;

Podemos perceber que para a execução correta das combinações, usamos somente colunas semelhantes de cada SELECT;

Artigo continuando o anterior, do Juliano Ignácio no iMasters - http://imasters.uol.com.br/artigo/966/postgresql/union_ou_union_all/

UNION ou UNION ALL

Na coluna anterior vimos o uso de UNION, no entanto, nem sempre desejamos o comportamento padrão apresentado. Quando o UNION é executado, os dados referentes às SELECTs envolvidas são ordenados, eliminando a duplicação de registros. Porém, se você deseja unir as consultas de forma a aparecer TODOS os registros, use UNION ALL.

Tomando como exemplo as tabelas criadas na coluna anterior

SELECT venda.produto FROM venda UNION SELECT troca.troca FROM troca;	SELECT venda.produto FROM venda UNION ALL SELECT troca.troca FROM troca;
produto ----- Despertador Fichário Garrafa Térmica Lapiseira Mala Viagem Maleta Executivo Mochila Relógio Rádio Saca-rolha (10 rows)	produto ----- Relógio Mala Viagem Saca-rolha Fichário Despertador Mochila Rádio Lapiseira Relógio Maleta Executivo Garrafa Térmica Fichário (12 rows)

SUPRIMINDO AS DUPLICIDADES

Para que valores duplicados não sejam mostrados no resultado de uma consulta (sem usar UNION,

como vimos anteriormente), usamos DISTINCT. Veja que, o valor duplicado se refere ao registro como um todo, e não somente a uma coluna. Para que possamos entender, vamos inserir algumas linhas em nossa tabela de vendas criada anteriormente

```
INSERT INTO venda (cliente_id, data, valor, produto)
VALUES (3,'2003-02-12',16,'Relógio');
INSERT INTO venda (cliente_id, data, valor, produto)
VALUES (1,'2003-02-15',110,'Mala Viagem');
INSERT INTO venda (cliente_id, data, valor, produto)
VALUES (3,'2003-02-16',10,'Saca-rolha');
INSERT INTO venda (cliente_id, data, valor, produto)
VALUES (1,'2003-02-20',32,'Fichário');
```

Após inserir as vendas acima, gostaria de saber quais são os produtos que estamos vendendo desde o início dos lançamentos, como iremos observar, a primeira consulta irá mostrar produtos repetidos, o que não é necessário e, às vezes, atrapalha. Na segunda consulta, a cláusula DISTINCT esconde a duplicidade dos registros e coloca-os na ordem da primeira coluna (neste caso só temos uma mesmo).

SELECT produto FROM venda;	SELECT DISTINCT produto FROM venda;
produto ----- Relógio Mala Viagem Saca-rolha Fichário despertador Mochila Rádio Lapiseira Relógio Mala Viagem Saca-rolha Fichário (12 rows)	produto ----- despertador Fichário Lapiseira Mala Viagem Mochila Relógio Rádio Saca-rolha (8 rows)

VARIÁVEIS MÁGICAS

O PostgreSQL possui 4 variáveis 'mágicas' que guardam informações sobre o usuário corrente e data e hora atuais, facilitando, talvez, a implementação de rotinas de auditoria.

CURRENT_DATE	SELECT
CURRENT_TIME	CURRENT_DATE;

CURRENT_TIMESTAMP	date ----- 2003-02-24
CURRENT_USER	(1 row)
Para saber a quanto tempo foram vendidos os produtos, podemos executar a seguinte consulta:	SELECT produto, (CURRENT_DATE - data) AS dias FROM venda ORDER BY produto; produto dias -----+----- Despertador 63 Fichário 4 Fichário 66 Lapiseira 66 Mala Viagem 63 Mala Viagem 9 Mochila 63 Relógio 63 Relógio 12 Rádio 65 Saca-rolha 8 Saca-rolha 65 (12 rows)

Veja o script aula4_TeoriaDosConjutos.sql com mais exercícios.

Tutorial online: http://www.w3schools.com/sql/sql_union.asp

5) Consultando dados em múltiplas tabelas

- 5.1) Utilizando Apelidos para as tabelas
- 5.2) Cruzando dados entre tabelas distintas
- 5.3) Entendendo os Tipos de Join disponíveis
- 5.4) Trabalhando com CROSS JOIN
- 5.5) Trabalhando com INNER e OUTER JOINS
- 5.6) Trabalhando com NATURAL JOIN

Cláusula JOIN

A cláusula JOIN é empregada para permitir que um mesmo select recupere informações de mais de uma fonte de dados (tabelas, views, etc.). Em geral, as tabelas referenciadas possuem algum tipo de relacionamento entre elas, através de um ou mais campos que definam a ligação entre uma tabela e a outra (integridade referencial).

Há duas maneiras de implementar um join:

- A primeira é chamada de **non-ANSI** ou estilo **theta**, que utiliza a cláusula WHERE para efetuar a junção de tabelas;
- A segunda é chamada de **ANSI Join**, e é baseada no uso da cláusula JOIN propriamente dita.

Simple ligação

Um exemplo de JOIN em estilo *ANSI*:

```
SELECT p.uname, p.nome, a.qtde  
from PESSOAS p  
CROSS JOIN ACESSOS a;
```

Um exemplo de JOIN em estilo *theta*:

```
SELECT p.uname, p.nome, a.qtde  
from PESSOAS p, ACESSOS a;
```

Note que na chamada ANSI utilizamos CROSS JOIN, que é a sintaxe utilizada para recuperar todos os registros das tabelas ligadas, formando um produto cartesiano. É basicamente um INNER JOIN (citado adiante) sem condições.

Tipos de junções

Inner Joins

Somente as linhas/registros que satisfaçam a ligação determinada pelo JOIN serão recuperados pelo select, sendo assim, os registros que **não** se enquadram no relacionamento definido pelo join **não serão recuperados**.

Um exemplo de INNER JOIN em estilo ANSI:

```
SELECT p.uname,  
p.nome,  
a.qtde  
from PESSOAS p  
INNER JOIN ACESSOS a on p.uname=a.pessoa order by p.uname;
```

O mesmo JOIN em estilo theta:

```
SELECT p.uname,  
p.nome,  
a.qtde  
from PESSOAS p, ACESSOS a WHERE p.uname = a.pessoa order by p.uname;
```

Left Joins

Através do uso do **LEFT**, todos os registros na tabela à esquerda da query serão listados, independente de terem ou não registros relacionados na tabela à direita. Nesse caso, as colunas relacionadas com a tabela da direita voltam nulos (NULL).

Um exemplo de uso LEFT JOIN:

```
SELECT p.uname,  
p.nome,  
a.pessoa,  
a.qtde  
from PESSOAS p  
LEFT JOIN ACESSOS a on p.uname=a.pessoa order by p.uname;
```

No exemplo acima, todos os registros da tabela PESSOAS serão listados, independente de terem ou não registros associados na tabela ACESSOS. Caso não existam registros associados na tabela ACESSOS, os campos *a.pessoa* e *a.qtde* retornarão NULL.

Right joins

É o inverso do Left Join, ou seja, todos os registros da tabela à direita serão listados, independente de terem ou não registros relacionados na tabela à esquerda.

Um exemplo de uso RIGHT JOIN:

```
SELECT p.uname,  
p.nome,  
a.pessoa,  
a.qtde  
from pessoas p  
RIGHT JOIN acessos a on p.uname=a.pessoas order by p.uname;
```

Ou seja, todos os registros da tabela ACESSOS serão listados, e caso não haja correspondentes na tabela PESSOAS, a query devolve NULL para os campos *p.uname* e *p.nome*.

Matematicamente um Join provém a operação fundamental em álgebra relacional.

Tipos de junção**Junção cruzada**

```
T1 CROSS JOIN T2
```

Para cada combinação de linhas de T1 e T2, a tabela derivada contém uma linha formada por todas as colunas de T1 seguidas por todas as colunas de T2. Se as tabelas possuírem N e M linhas, respectivamente, a tabela juntada terá $N * M$ linhas.

FROM T1 CROSS JOIN T2 equivale a FROM T1, T2.

As palavras INNER e OUTER são opcionais em todas as formas. INNER é o padrão; LEFT, RIGHT e FULL implicam em junção externa.

A *condição de junção* é especificada na cláusula ON ou USING, ou implicitamente pela palavra NATURAL. A condição de junção determina quais linhas das duas tabelas de origem são consideradas "correspondentes", conforme explicado detalhadamente abaixo.

Os tipos possíveis de junção qualificada são:

INNER JOIN

Para cada linha L1 de T1, a tabela juntada possui uma linha para cada linha de T2 que satisfaz a condição de junção com L1.

LEFT OUTER JOIN

Primeiro, é realizada uma junção interna. Depois, para cada linha de T1 que não satisfaz a condição de junção com nenhuma linha de T2, é adicionada uma linha juntada com valores nulos nas colunas de T2. Portanto, a tabela juntada possui, incondicionalmente, no mínimo uma linha para cada linha de T1.

RIGHT OUTER JOIN

Primeiro, é realizada uma junção interna. Depois, para cada linha de T2 que não satisfaz a condição de junção com nenhuma linha de T1, é adicionada uma linha juntada com valores nulos nas colunas de T1. É o oposto da junção esquerda: a tabela resultante possui, incondicionalmente, uma linha para cada linha de T2.

FULL OUTER JOIN

Primeiro, é realizada uma junção interna. Depois, para cada linha de T1 que não satisfaz a condição de junção com nenhuma linha de T2, é adicionada uma linha juntada com valores nulos nas colunas de T2. Também, para cada linha de T2 que não satisfaz a condição de junção com nenhuma linha de T1, é adicionada uma linha juntada com valores nulos nas colunas de T1.

Tipos de junção no PostgreSQL, no SQL Server, no Oracle e no DB2

Tipo de junção	PostgreSQL 8.0.0	SQL Server 2000	Oracle 10g	DB2 8.1
INNER JOIN ON	sim	sim	sim	sim
LEFT OUTER JOIN ON	sim	sim	sim	sim
RIGHT OUTER JOIN ON	sim	sim	sim	sim
FULL OUTER JOIN ON	sim	sim	sim	sim

Tipo de junção	PostgreSQL 8.0.0	SQL Server 2000	Oracle 10g	DB2 8.1
INNER JOIN USING	sim	não	sim	não
CROSS JOIN	sim	sim	sim	não
NATURAL JOIN	sim	não	sim	não

Cláusula JOIN

A cláusula JOIN é empregada para permitir que um mesmo select recupere informações de mais de uma fonte de dados (tabelas, views, etc.). Em geral, as tabelas referenciadas possuem algum tipo de relacionamento entre elas, através de um ou mais campos que definam a ligação entre uma tabela e a outra (integridade referencial).

Há duas maneiras de implementar um join:

- A primeira é chamada de **non-ANSI** ou estilo **theta**, que utiliza a cláusula WHERE para efetuar a junção de tabelas;
- A segunda é chamada de **ANSI Join**, e é baseada no uso da cláusula JOIN propriamente dita.

Simples ligação

Um exemplo de JOIN em estilo ANSI:

```
SELECT p.uname, p.nome, a.qtde
from PESSOAS p
CROSS JOIN ACESSOS a;
```

Um exemplo de JOIN em estilo *theta*:

```
SELECT p.uname, p.nome, a.qtde
from PESSOAS p, ACESSOS a;
```

Note que na chamada ANSI utilizamos CROSS JOIN, que é a sintaxe utilizada para recuperar todos os registros das tabelas ligadas, formando um produto cartesiano. É basicamente um INNER JOIN (citado adiante) sem condições.

Tipos de junções

Inner Joins

Somente as linhas/registros que satisfaçam a ligação determinada pelo JOIN serão recuperados pelo select, sendo assim, os registros que **não** se enquadram no relacionamento definido pelo join **não serão recuperados**.

Um exemplo de INNER JOIN em estilo ANSI:

```
SELECT p.uname,
p.nome,
a.qtde
from PESSOAS p
INNER JOIN ACESSOS a on p.uname=a.pessoa order by p.uname;
```

O mesmo JOIN em estilo theta:

```
SELECT p.uname,
p.nome,
```

```
a.qtde
from PESSOAS p, ACESSOS a WHERE p.uname = a.pessoa order by p.uname;
```

Left Joins

Através do uso do **LEFT**, todos os registros na tabela à esquerda da query serão listados, independente de terem ou não registros relacionados na tabela à direita. Nesse caso, as colunas relacionadas com a tabela da direita voltam nulos (NULL).

Um exemplo de uso LEFT JOIN:

```
SELECT p.uname,
p.nome,
a.pessoa,
a.qtde
from PESSOAS p
LEFT JOIN ACESSOS a on p.uname=a.pessoa order by p.uname;
```

No exemplo acima, todos os registros da tabela PESSOAS serão listados, independente de terem ou não registros associados na tabela ACESSOS. Caso não existam registros associados na tabela ACESSOS, os campos *a.pessoa* e *a.qtde* retornarão NULL.

Right joins

É o inverso do Left Join, ou seja, todos os registros da tabela à direita serão listados, independente de terem ou não registros relacionados na tabela à esquerda.

Um exemplo de uso RIGHT JOIN:

```
SELECT p.uname,
p.nome,
a.pessoa,
a.qtde
from pessoas p
RIGHT JOIN acessos a on p.uname=a.pessoas order by p.uname;
```

Ou seja, todos os registros da tabela ACESSOS serão listados, e caso não haja correspondentes na tabela PESSOAS, a query devolve NULL para os campos *p.uname* e *p.nome*.

5.1) Utilizando Apelidos para as tabelas

Quando realizamos consultas em várias tabelas fica menor a consulta se adotarmos apelidos para as tabelas.

Por exemplo: vamos realizar uma consulta que envolve as tabelas: alunos e notas, então podemos usar os apelidos: *a* e *n*. Mas isso é algo opcional.

```
\c dml
create table alunos(codaluno int, nome varchar(45));
create table notas(codaluno int, nota1 numeric(4,2));

insert into alunos(codaluno, nome) values (1, 'João Pereira Brito');
insert into alunos(codaluno, nome) values (2, 'Roberto Pereira Brito');
insert into alunos(codaluno, nome) values (3, 'Manoel Pereira Brito');
insert into alunos(codaluno, nome) values (4, 'Pedro Pereira Brito');
insert into alunos(codaluno, nome) values (5, 'Francisco Pereira Brito');
```

```
insert into notas (codaluno, nota1) values (1, 7);
insert into notas (codaluno, nota1) values (2, 5);
insert into notas (codaluno, nota1) values (3, 8);
insert into notas (codaluno, nota1) values (4, 6);
insert into notas (codaluno, nota1) values (5, 9);
```

Quero trazer alunos e notas:

```
SELECT a.nome, n.nota1
FROM alunos a
INNER JOIN notas n ON a.codaluno = n.codaluno order by nota1;
```

5.2) Cruzando dados entre tabelas distintas

```
SELECT a.nome, n.nota1
FROM alunos a, notas n
WHERE a.codaluno = n.codaluno order by nota1;
```

As junções SQL são utilizadas quando precisamos selecionar dados de duas ou mais tabelas.

Existem as junções com estilo non-ANSI ou theta (junção com WHERE sem usar explicitamente a cláusula JOIN)

5.3) Entendendo os Tipos de Join disponíveis

As junções ANSI join (com JOIN explícito).

As junções ANSI podem ser de dois tipos, as INNER e as OUTER, que se subdividem em INNER, OUTER, LEFT e RIGHT.

A padrão é a INNER JOIN. INNER JOIN pode ser escrito com apenas JOIN.

Tipos de Junção Suportados pelo PostgreSQL:

INNER JOIN:

- NATURAL JOIN
- CROSS JOIN

OUTER JOIN

- LEFT JOIN
- RIGHT JOIN
- FULL JOIN

5.4) Trabalhando com CROSS JOIN

Chamado também de cartesiano Join ou produto. Um cross join retorna o produto cartesiano do conjunto de registros das duas tabelas da junção.

Se A e B são dois conjuntos então cross join será $A \times B$.

Cross Join Explícito:

```
SELECT *  
FROM funcionarios CROSS JOIN departamentos;
```

Cross Join Implícito:

```
SELECT *  
FROM funcionarios, departamentos;
```

Outros Exemplos:

```
SELECT p.maricula, p.senha, d.departamento FROM pessoal p CROSS JOIN departamento d;
```

INNER JOIN - Onde todos os registros que satisfazem à condição serão retornados.

Exemplo:

```
SELECT p.siape, p.nome, l.lotacao  
FROM pessoal p INNER JOIN lotacoes l  
ON p.siape = l.siape ORDER BY p.siape;
```

Exemplo no estilo theta (non-ANSI):

```
SELECT p.matricula, p.nome, d.departamento  
FROM pessoal p, departamento d  
WHERE p.matricula = d.matricula ORDER BY p.matricula;
```

OUTER JOIN que se divide em LEFT OUTER JOIN e RIGHT OUTER JOIN

LEFT OUTER JOIN ou simplesmente LEFT JOIN - Somente os registros da tabela da esquerda (left) serão retornados, tendo ou não registros relacionados na tabela da direita.

Primeiro, é realizada uma junção interna. Depois, para cada linha de T1 que não satisfaz a condição de junção com nenhuma linha de T2, é adicionada uma linha juntada com valores nulos nas colunas de T2. Portanto, a tabela juntada possui, incondicionalmente, no mínimo uma linha para cada linha de T1.

A tabela à esquerda do operador de junção exibirá cada um dos seus registros, enquanto que a da direita exibirá somente seus registros que tenham correspondentes aos da tabela da esquerda. Para os registros da direita que não tenham correspondentes na esquerda serão colocados valores NULL.

Exemplo (voltar todos somente de pessoal):

```
SELECT p.matricula, p.nome, d.departamentos  
FROM pessoal p LEFT JOIN departamentos d  
ON p.siape = d.matricula ORDER BY p.matricula ;
```

Veja que pessoal fica à esquerda em “FROM pessoal p LEFT JOIN departamentos d”.

RIGHT OUTER JOIN

Inverso do LEFT, este retorna todos os registros somente da tabela da direita (right). Primeiro, é realizada uma junção interna. Depois, para cada linha de T2 que não satisfaz a condição de junção com nenhuma linha de T1, é adicionada uma linha juntada com valores nulos nas colunas de T1. É o oposto da junção esquerda: a tabela resultante possui, incondicionalmente, uma linha para cada linha de T2.

Exemplo (retornar somente os registros de lotacoes):

```
SELECT p.matricula, p.nome, d.departamentos  
FROM pessoal p RIGHT JOIN departamentos d  
ON p.siape = d.matricula ORDER BY p.nome;
```

FULL OUTER JOIN

Primeiro, é realizada uma junção interna. Depois, para cada linha de T1 que não satisfaz a condição de junção com nenhuma linha de T2, é adicionada uma linha juntada com valores nulos nas colunas de T2. Também, para cada linha de T2 que não satisfaz a condição de junção com nenhuma linha de T1, é adicionada uma linha juntada com valores nulos nas colunas de T1.

E também as:

5.5) Trabalhando com INNER e OUTER JOINS

INNER JOIN

Um exemplo de um inner join !

Vamos supor a seguinte estrutura de tabelas:

Temos as tabelas alunos, notas e frequencias

Alunos Notas Frequencias

CodAluno CodNotas CodFrequencia
Nome CodAluno CodAluno
Endereco Nota1 Freq1
Fone Nota2 Freq2

Para você selecionar vamos supor:

O nome do aluno com a nota 1 e frequencia 1; o SELECT seria assim:

```
SELECT A.Nome, N.Nota1, F.Freq1  
FROM Alunos A  
INNER JOIN Notas N ON A.CodAluno = N.CodAluno  
INNER JOIN Frequencias F ON A.CodAluno = F.CodAluno
```

Isso buscaria de TODOS os Alunos sem excessão, o Nome, Nota1 e Freq1.

Agora se vc quisesse trazer de um determinado aluno, bastaria você acrescentar a seguinte linha:
WHERE A.CodAluno = ????

5.6) Trabalhando com NATURAL JOIN

É uma especialização do Equi-Join e NATURAL é uma forma abreviada de USING. Comparam-se ambas as tabelas do join e o resultado conterá somente uma coluna de cada par de colunas de mesmo nome.

Exemplo:

Tendo como base as duas tabelas:

Employee

LastName	DepartmentID
Rafferty	31
Jones	33
Steinberg	33
Robinson	34
Smith	34
Jasper	36

Department**DepartmentID DepartmentName**

31	Sales
33	Engineering
34	Clerical
35	Marketing

```
SELECT *  
FROM employee NATURAL JOIN department;
```

Somente um campo DepartmentID aparece na tabela resultante.

DepartmentID Employee.LastName Department.DepartmentName

34	Smith	Clerical
33	Jones	Engineering
34	Robinson	Clerical
33	Steinberg	Engineering
31	Rafferty	Sales

Mais informações em:

http://www.w3schools.com/sql/sql_join.asp

[http://en.wikipedia.org/wiki/Join_\(SQL\)](http://en.wikipedia.org/wiki/Join_(SQL))

<http://www.postgresql.org/docs/8.2/static/tutorial-join.html>

<http://www.postgresql.org/docs/current/static/queries-table-expressions.html>

<http://pgdocptbr.sourceforge.net/pg80/tutorial-join.html>

<http://pgdocptbr.sourceforge.net/pg80/queries-table-expressions.html>

Consultas com JOINS – Thiago Caserta

Obs.: Faça os devidos ajustes, já que este artigo foi feito para o SQL Server.

Quando comecei na área de TI, sempre me deparava com situações no SQL em que precisava pegar campos de outras tabelas não vinculadas diretamente com a que estamos manipulando, e sentia certa dificuldade em amarrar as tabelas pelas suas respectivas chaves, fossem elas código, id, usuário, etc.

Bem, algo que é de extrema ajuda nesses casos são os nossos amigos JOINS (**LEFT JOIN**, **RIGHT JOIN**, **INNER JOIN** e **FULL JOIN**).

Hoje percebo que muitos daqueles que estão começando na área de TI sentem o mesmo.

Junções, tradução de JOINS, são utilizadas em duas cláusulas específicas: **FROM** e **WHERE**, eu particularmente prefiro usar na cláusula **FROM**, por questões de desempenho e organização.

Neste artigo vamos analisar algumas consultas possíveis com essas poderosas cláusulas. Então: Let's JOIN!

A princípio vamos entender o primeiro **JOIN** mencionado, o **LEFT JOIN**.

Como podemos observar, e a própria sintaxe indica, essa cláusula trabalha com os dados da tabela "Esquada" como sendo os dados principais, ou seja, de acordo com o exemplo abaixo, o **LEFT JOIN** mostrará o que esta na Tabela1 (esquerda), podendo trabalhar também com qualquer outro dado da Tabela2 com a mesma chave encontrada na Tabela1.

Os dados principais que estaremos trabalhando serão os da Tabela1, já que esta, como já mencionado, é a nossa tabela " Esquerda ".

```
SELECT Tab1.* FROM Tabela1 Tab1  
LEFT JOIN Tabela2 Tab2 ON Tab1.Cod = Tab2.Cod
```

Podemos fazer a mesma amarração junto à cláusula **WHERE** assim como no exemplo abaixo onde estamos pegando os mesmos valores do exemplo acima.

```
SELECT Tab1.* FROM Tabela1 Tab1, Tabela2 Tab2  
WHERE Tab1.Cod *= Tab2.Cod
```

Nesse exemplo os sinais " *= " indicam a condição **LEFT JOIN**.

Partiremos agora para o irmão mais próximo do **LEFT JOIN**, o **RIGHT JOIN**.

O **RIGHT JOIN** retorna o que estiver na Tabela1 e Tabela2 com a mesma chave, e sendo o inverso do **LEFT JOIN** a tabela principal se torna a tabela da " Direita ", ou seja a Tabela2.

```
SELECT Tab2.* FROM @Tabela1 Tab1  
RIGHT JOIN @Tabela2 T2 ON Tab1.Cod = Tab2.Cod
```

Do mesmo modo que podemos utilizar o **LEFT JOIN** na cláusula **WHERE** podemos fazer assim também com o **RIGHT JOIN**.

```
SELECT Tab2.* FROM @Tabela1 Tab1, @Tabela2 Tab2  
WHERE T1.Cod =* T2.Cod
```

Observe que o sinal no **RIGHT JOIN** é diferente do **LEFT JOIN**, de "Asterisco ="mudamos para"= asterisco ".

Alteramos o asterisco da esquerda para a direita, o que se torna uma ajuda para não confundirmos as cláusulas.

LEFT JOIN - Asterisco à esquerda;

RIGHT JOIN - Asterisco à direita.

Desse ponto partiremos para os dois últimos **JOINS**, o **INNER JOIN** e o **FULL JOIN**.

O **INNER JOIN** nos retorna apenas o que esta na Tabela1 e Tabela2 com a mesma chave.

Exemplo:

```
SELECT * FROM @Tabela1 T1
INNER JOIN @Tabela2 T2 ON T1.Cod = T2.Cod
```

Assim como o **LEFT JOIN** e o **RIGHT JOIN**, podemos da mesma forma fazer essa amarração junto à cláusula **WHERE**, como segue o exemplo:

```
SELECT * from @Tabela T1, @Tabela2 T2
WHERE T1.Cod = T2.Cod
```

Nesse exemplo o sinal " = " indica a função **INNER JOIN**.

Já O **FULL JOIN** retorna o que estiver na Tabela1 e Tabela2 levando em conta o seu significado **FULL**, ou seja, completo. Portanto o **FULL JOIN** retorna tudo o que há nas Tabelas selecionadas

```
SELECT * FROM @Tabela1 T1
FULL JOIN @Tabela2 T2 ON T1.Cod = T2.Cod
```

Bem, essas são as 4 cláusulas mais utilizadas para fazermos amarrações entre tabelas ou pegarmos valores relacionados. Com certeza é de grande ajuda para todos os que fazem uso de banco de dados independentemente de qual seja.

Em anexo está um [exemplo](#) de todas as cláusulas tratadas aqui, porém com banco e tabelas reais.

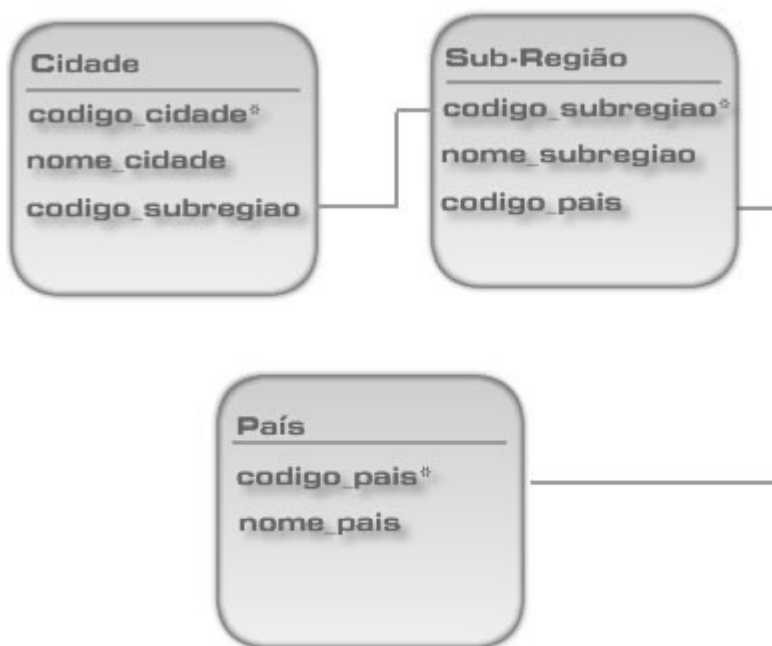
Este exemplo foi baseado no **Microsoft SQL Server**. As sintaxes das consultas não irão mudar, porém para àqueles que forem executar os exemplos em outros bancos terão de modificar os scripts de criação das tabelas.

Espero que esta matéria tenha sido de grande ajuda para àqueles que estão começando ou sentem dificuldades nesse assunto.

Junção entre tabelas no PostgreSQL - Daniel Oslei

A compreensão da real utilidade da junção de tabelas no estudo de banco de dados, e de que forma isto é feito, é um obstáculo para muitos estudantes. A dúvida mais constante a cerca do assunto é com o comando SQL conhecido como JOIN. Já recebi vários e-mails contendo dúvidas relacionadas a utilização correta dos JOINS. Por isso, o objetivo de hoje é esclarecer com uma sequência de exemplos os tipos de junções de tabelas possíveis no PostgreSQL.

Para os nossos exemplos utilizaremos uma estrutura de três tabelas simples com alguns dados inseridos. O diagrama abaixo representa o relacionamento entre as tabelas:



Vamos partir para o povoamento das tabelas, em que serão inseridos alguns poucos dados, apenas para a efetuação de nossas consultas:

Tabela cidade		
codigo	cidade	subregiao
1	Curitiba	1
2	Sao Paulo	2

3	Guarulhos	2
4	Buenos Aires	4
5	La Plata	4
6	Cordoba	5
7	Los Angeles	6
8	San Francisco	6
9	Orlando	7
10	Miami	7
11	Siena	8
12	Florenca	8
13	Milao	9
14	Yokohama	Null

Tabela subregiao

codigo	subregiao	pais
1	Parana	1
2	Sao Paulo	1
3	Rio Grande do Sul	1

4	Buenos Aires	2
5	Cordoba	2
6	California	3
7	Florida	3
8	Toscana	4
9	Lombardia	4
10	Aquitania	5
11	Borgonha	5
12	Calabria	5
13	Massachussetts	3
14	Chiapas	Null

Tabela País

codigo	pais
1	Brasil
2	Argentina
3	Estados Unidos
4	Italia

5	Franca
6	Noruega

Script SQL para criação das tabelas

Tabela cidade

```
CREATE TABLE "public"."cidade" (  
  "codigo_cidade" SERIAL,  
  "nome_cidade" VARCHAR(50),  
  "codigo_subregiao" INTEGER,  
  CONSTRAINT "cidade_pkey" PRIMARY  
  KEY("codigo_cidade")  
) WITH OIDS;
```

Tabela subregiao

```
CREATE TABLE "public"."subregiao" (  
  "codigo_subregiao" SERIAL,  
  "nome_subregiao" VARCHAR(50),  
  "codigo_pais" INTEGER,  
  CONSTRAINT "subregiao_pkey" PRIMARY  
  KEY("codigo_subregiao")  
) WITH OIDS;
```

Tabela país

```
CREATE TABLE "public"."pais" (  
  "codigo_pais" SERIAL,  
  "nome_pais" VARCHAR(50),  
  CONSTRAINT "pais_pkey" PRIMARY  
  KEY("codigo_pais")  
) WITH OIDS;
```

Inserção de dados

```
INSERT INTO pais (nome_pais) VALUES ('Brasil');  
INSERT INTO pais (nome_pais) VALUES ('Argentina');  
INSERT INTO pais (nome_pais) VALUES ('Estados
```



```
Unidos');
INSERT INTO pais (nome_pais) VALUES ('Italia');
INSERT INTO pais (nome_pais) VALUES ('Franca');
INSERT INTO pais (nome_pais) VALUES ('Noruega');

INSERT INTO subregiao ( nome_subregiao, codigo_pais)
VALUES ( 'Parana', 1);
INSERT INTO subregiao ( nome_subregiao, codigo_pais)
VALUES ( 'Sao Paulo', 1);
INSERT INTO subregiao ( nome_subregiao, codigo_pais)
VALUES ( 'Rio Grande do Sul', 1);
INSERT INTO subregiao ( nome_subregiao, codigo_pais)
VALUES ( 'Buenos Aires', 2);
INSERT INTO subregiao ( nome_subregiao, codigo_pais)
VALUES ( 'Cordoba', 2);
INSERT INTO subregiao ( nome_subregiao, codigo_pais)
VALUES ( 'California', 3);
INSERT INTO subregiao ( nome_subregiao, codigo_pais)
VALUES ( 'Florida', 3);
INSERT INTO subregiao ( nome_subregiao, codigo_pais)
VALUES ( 'Toscana', 4);
INSERT INTO subregiao ( nome_subregiao, codigo_pais)
VALUES ( 'Lombardia', 4);
INSERT INTO subregiao ( nome_subregiao, codigo_pais)
VALUES ( 'Aquitania', 5);
INSERT INTO subregiao ( nome_subregiao, codigo_pais)
VALUES ( 'Borgonha', 5);
INSERT INTO subregiao ( nome_subregiao, codigo_pais)
VALUES ( 'Calabria', 5);
INSERT INTO subregiao ( nome_subregiao, codigo_pais)
VALUES ( 'Massachussetts', 3);
INSERT INTO subregiao ( nome_subregiao, codigo_pais)
VALUES ( 'Chiapas', NULL);

INSERT INTO cidade (nome_cidade, codigo_subregiao)
VALUES ('Curitiba', 1);
INSERT INTO cidade (nome_cidade, codigo_subregiao)
VALUES ('Sao Paulo', 2);
INSERT INTO cidade (nome_cidade, codigo_subregiao)
VALUES ('Guarulhos', 2);
INSERT INTO cidade (nome_cidade, codigo_subregiao)
VALUES ('Buenos Aires', 4);
INSERT INTO cidade (nome_cidade, codigo_subregiao)
VALUES ('La Plata', 4);
INSERT INTO cidade (nome_cidade, codigo_subregiao)
VALUES ('Cordoba', 5);
INSERT INTO cidade (nome_cidade, codigo_subregiao)
```

```
VALUES ('Los Angeles', 6);
INSERT INTO cidade (nome_cidade, codigo_subregiao)
VALUES ('San Francisco', 6);
INSERT INTO cidade (nome_cidade, codigo_subregiao)
VALUES ('Orlando', 7);
INSERT INTO cidade (nome_cidade, codigo_subregiao)
VALUES ('Miami', 7);
INSERT INTO cidade (nome_cidade, codigo_subregiao)
VALUES ('Siena', 8);
INSERT INTO cidade (nome_cidade, codigo_subregiao)
VALUES ('Florenca', 8);
INSERT INTO cidade (nome_cidade, codigo_subregiao)
VALUES ('Milao', 9);
INSERT INTO cidade (nome_cidade, codigo_subregiao)
VALUES ('Yokohama', NULL);
```

A junção de tabelas ocasiona uma tabela derivada de outras duas tabelas (reais ou derivadas), de acordo com as regras do tipo de junção. No PostgreSQL as junções são classificadas como sendo qualificadas ou cruzadas.

Junções cruzadas

```
SELECT * FROM Tabela1 CROSS JOIN Tabela2
```

Cada linha de Tabela1 irá combinar-se com todas as linhas de Tabelas2. Para cada combinação de linhas de Tabela1 e Tabela2, a tabela derivada conterá uma linha com todas as colunas de Tabela1 seguidas por todas as colunas de Tabela2. O número de linhas retornadas por esta consulta sempre será o número de linhas de Tabela1 multiplicado pelo número de linha de Tabela2. Por exemplo, se Tabela1 possuir 20 linhas e Tabela2 possuir 10 linhas, será retornado 200 linhas. A consulta `SELECT * FROM cidade CROSS JOIN subregiao` de nosso exemplo retornará 196 linhas.

É óbvio que destas 196 linhas retornadas a maioria pode ser considerada inútil, portanto, devemos selecionar os nossos dados através de condições para nossa consulta. Essas condições são adicionadas através de cláusula `WHERE`.

```
SELECT * FROM cidade CROSS JOIN subregiao WHERE cidade.subregiao = subregiao.codigo
```

Como é perceptível, o uso de `CROSS JOIN` permite a junção de apenas duas tabelas. No entanto, nosso exemplo precisa juntar três tabelas, para isso, teremos que primeiro unir duas tabelas, para que o resultado desta junção seja utilizado com a terceira tabela.

```
SELECT * FROM cidade CROSS JOIN (subregiao CROSS JOIN pais).
```

Utilizar `SELECT * FROM cidade CROSS JOIN subregiao` equivale a utilizar `SELECT * FROM cidade, subregiao`, tanto uma como outra retornará as mesmas 196 linhas e utilizar `SELECT * FROM cidade CROSS JOIN (subregiao CROSS JOIN pais)` equivale a `SELECT * FROM cidade, subregiao, pais`, ambas retornarão as mesmas 1176 linhas.

Junções Qualificadas

As junções qualificadas trazem um pouquinho mais de complexidade e são divididas em junções internas e externas. Na utilização de junção qualificada, se não for especificado como junção interna

ou externa, por padrão o PostgreSQL considera como sendo interna.

Junções internas

A utilização da cláusula INNER é o que caracteriza o comando para uma junção interna, porém, ele não é obrigatório. Pode parecer à primeira vista que as junções internas se equiparam com as junções cruzadas vistas anteriormente, até por que as duas consultas a seguir são equivalentes:

```
SELECT * FROM cidade CROSS JOIN subregiao
```

```
SELECT * FROM cidade INNER JOIN subregiao ON TRUE
```

Mas nas junções internas é sempre obrigatória a especificação de condição de junção, ou seja, quais linhas de uma tabela têm alguma ligação com a linha de outra tabela. Para isso podemos utilizar uma das cláusulas ON ou USING ou utilizar a palavra NATURAL no nosso comando.

A cláusula ON é o mais comumente utilizado por se assemelhar com a cláusula WHERE, ou seja, um par de linhas de Tabela1 e Tabela2 são correspondentes, se a expressão da cláusula ON produz um resultado verdade (*true*) para este par de linhas.

```
SELECT * FROM cidade INNER JOIN subregiao ON  
cidade.codigo_subregiao = subregiao.codigo_subregiao
```

codigo_cidade	nome_cidade	codigo_subregiao	codigo_subregiao_1	nome_subregiao	codigo_pais
1	Curitiba	1	1	Parana	1
2	Sao Paulo	2	2	Sao Paulo	1
3	Guarulhos	2	2	Sao Paulo	1
4	Buenos Aires	4	4	Buenos Aires	2
5	La Plata	4	4	Buenos Aires	2
6	Cordoba	5	5	Cordoba	2
8	San Francisco	6	6	California	3
7	Los Angeles	6	6	California	3

9	Orlando	7	7	Florida	3
10	Miami	7	7	Florida	3
11	Siena	8	8	Toscana	4
12	Florenca	8	8	Toscana	4
13	Milao	9	9	Lombardia	4

A cláusula USING traz alguma semelhança com o ON, por também retornar um valor verdadeiro ou falso para aquele conjunto de linhas, no entanto, ele é uma forma mais rápida e abreviada de criação da consulta. Passando um nome de coluna, a execução desta consulta irá procurar nas tabelas a coluna especificada e comparar as duas. Por exemplo, `t1 INNER JOIN t2 USING (a, b, c)` equivale a `t1 INNER JOIN t2 ON (t1.a = t2.a AND t1.b = t2.b AND t1.c = t2.c)`. Portanto, a consulta anterior equivale à consulta abaixo:

```
SELECT * FROM subregiao INNER JOIN cidade USING
(codigo_subregiao)
```

Para facilitar mais, existe a utilização de NATURAL, que nada mais é abreviação de USING. Com NATURAL, a consulta encontrará todas as colunas que tem nomes iguais nas duas tabelas e fará a comparação de igualdade. O exemplo de USING acima equivale ao seguinte:

```
SELECT * FROM subregiao NATURAL INNER JOIN cidade
```

Mas cuidado com a utilização de NATURAL, pois, ele vai comparar todas as colunas com nomes iguais, o que pode trazer resultados inesperados quando houver duas colunas com o mesmo nome e estas não tenham nenhuma relação.

Junções Externas

Para representar uma junção externa utiliza-se a cláusula OUTER, no entanto, ela não é obrigatória. O que caracteriza realmente as junções externas são as cláusulas LEFT, RIGHT e FULL. As cláusulas ON, USING e NATURAL valem da mesma forma nas junções internas e externas.

LEFT OUTER JOIN

Primeiro, uma junção interna é realizada. Depois, para cada linha de T1 que não satisfaz a condição de junção com nenhuma linha de T2, uma linha juntada é adicionada com valores nulos nas colunas de T2. Portanto, a tabela juntada possui, incondicionalmente, no mínimo uma linha para cada linha de T1.

```
SELECT * FROM subregiao LEFT OUTER JOIN cidade USING (codigo_subregiao)
```

codigo_subregiao	nome_subregiao	codigo_pais	codigo_cidade	nome_cidade
------------------	----------------	-------------	---------------	-------------

	1	Parana	1	1	Curitiba
	2	Sao Paulo	1	2	Sao Paulo
	2	Sao Paulo	1	3	Guarulhos
	3	Rio G. do Sul	1	Null	Null
	4	Buenos Aires	2	4	Buenos Aires
	4	Buenos Aires	2	5	La Plata
	5	Cordoba	2	6	Cordoba
	6	California	3	7	Los Angeles
	6	California	3	8	San Francisco
	7	Florida	3	9	Orlando
	7	Florida	3	10	Miami
	8	Toscana	4	11	Siena
	8	Toscana	4	12	Florenca
	9	Lombardia	4	13	Milao
	10	Aquitania	5	Null	Null
	11	Borgonha	5	Null	Null
	12	Calabria	5	Null	Null

13	Massachussetts	3	Null	Null
----	----------------	---	------	------

Reparem nas linhas destacadas acima. As sub-regiões Rio Grande do Sul, Aquitania, Borgonha, Calabria e Massachussetts não possuem nenhuma cidade registrada. Em uma consulta normal eles seriam ignorados. Com o uso de LEFT todos as linhas das tabelas da esquerda que não possuem correspondentes na tabela da direita são acrescentadas no resultado da consulta.

RIGHT OUTER JOIN

Primeiro, uma junção interna é realizada. Depois, para cada linha de T2 que não satisfaz a condição de junção com nenhuma linha de T1, uma linha juntada é adicionada com valores nulos nas colunas de T1. É o oposto da junção esquerda: a tabela resultante possui, incondicionalmente, uma linha para cada linha de T2.

SELECT * FROM subregiao RIGHT OUTER JOIN pais USING (codigo_pais)

codigo_pais	codigo_subregiao	nome_subregiao	nome_cidade
1	2	Sao Paulo	Brasil
1	3	Rio Grande do Sul	Brasil
1	1	Parana	Brasil
2	4	Buenos Aires	Argentina
2	5	Cordoba	Argentina
3	13	Massachussetts	Estados Unidos
3	6	California	Estados Unidos
3	7	Florida	Estados Unidos
4	9	Lombardia	Italia

4	8	Toscana	Italia
5	10	Aquitania	Franca
5	11	Borgonha	Franca
5	12	Calabria	Franca
6	Null	Null	Noruega

Basicamente, a diferença entre RIGHT e LEFT está na escolha da tabela em que os elementos que não possuem correspondentes serão escolhidos para ser acrescentados no resultado da consulta. Neste exemplo, Noruega não tem nenhuma sub-região cadastrada, mas mesmo assim ele entra no resultado final.

Continuaremos na próxima matéria, publicada ainda hoje, iniciando com o FULL OUTER JOIN.

Junção entre tabelas no PostgreSQL - Parte 02

Continuaremos falando sobre a compreensão da real utilidade da junção de tabelas no estudo de banco de dados. Para acessar a primeira parte da matéria, publicada hoje também, acesse o link <http://www.imasters.com.br/artigo.php?cn=2867&cc=23>

FULL OUTER JOIN

Primeiro, uma junção interna é realizada. Depois, para cada linha de T1 que não satisfaz a condição de junção com nenhuma linha de T2, uma linha juntada é adicionada com valores nulos nas colunas de T2. Também, para cada linha de T2 que não satisfaz a condição de junção com nenhuma linha de T1, uma linha juntada com valores nulos nas colunas de T1 é adicionada.

`SELECT * FROM subregiao FULL OUTER JOIN cidade USING (codigo_subregiao)`

codigo_subregiao	nome_subregiao	codigo_pais	codigo_cidade	nome_cidade
1	Parana	1	1	Curitiba
2	Sao Paulo	1	2	Sao Paulo
2	Sao Paulo	1	3	Guarulhos
3	Rio G. do Sul	1	Null	Null
4	Buenos Aires	2	4	Buenos Aires
4	Buenos Aires	2	5	La Plata
5	Cordoba	2	6	Cordoba
6	California	3	8	San Francisco
6	California	3	7	Los Angeles
7	Florida	3	9	Orlando

7	Florida	3	10	Miami
8	Toscana	4	11	Siena
8	Toscana	4	12	Florenca
9	Lombardia	4	13	Milao
10	Aquitania	5	Null	Null
11	Borgonha	5	Null	Null
12	Calabria	5	Null	Null
13	Massachussetts	3	Null	Null
Null	Null	Null	14	Yokohama

O uso de FULL não é nada mais que a utilização de RIGHT e LEFT juntos. Neste exemplo, foram acrescentados 5 sub-regiões que não possuem nenhum correspondente na tabela de cidade e nesta consulta apareceu a cidade de Yokohama que não possui uma sub-região.

Para buscarmos todos os dados de nossas tabelas utilizando JOIN podemos usar o seguinte comando:

```
SELECT * FROM cidade FULL JOIN (subregiao FULL JOIN pais USING (codigo_pais)) USING (codigo_subregiao)
```

ou

```
SELECT * FROM cidade FULL JOIN subregiao FULL JOIN pais USING (codigo_pais) USING (codigo_subregiao)
```

codigo_subregiao	codigo_cidade	nome_cidade	codigo_pais	nome_subregiao	nome_pais
1	1	Curitiba	1	Parana	Brasil
2	2	Sao Paulo	1	Sao Paulo	Brasil
2	3	Guarulhos	1	Sao Paulo	Brasil

3	Null	Null	1	Rio G. do Sul	Brasil
4	4	Buenos Aires	2	Buenos Aires	Argentina
4	5	La Plata	2	Buenos Aires	Argentina
5	6	Cordoba	2	Cordoba	Argentina
6	8	San Francisco	3	California	Estados Unidos
6	7	Los Angeles	3	California	Estados Unidos
7	9	Orlando	3	Florida	Estados Unidos
7	10	Miami	3	Florida	Estados Unidos
8	11	Siena	4	Toscana	Itália
8	12	Florenca	4	Toscana	Itália
9	13	Milao	4	Lombardia	Itália
Null	14	Yokohama	Null	Null	Null
10	Null	Null	5	Aquitania	Franca
11	Null	Null	5	Borgonha	Franca
12	Null	Null	5	Calabria	Franca
13	Null	Null	3	Massachussets	Estados Unidos

14	Null	Null	Null	Chiapas	Null
Null	Null	Null	6	Null	Noruega

Atenção nas condições da consulta

Quando se usa junção externa deve-se ter muito cuidado com as condições utilizadas na consulta, pois, lembre-se que nestas consultas mesmo que a condição não satisfaça uma linha em comparação com linhas da outra tabela, elas serão retornadas acompanhadas de valores nulos. Vejam um exemplo:

Se acaso quiser saber quais são as cidades registradas como sendo da região de Toscana, as consultas abaixo podem não ser as mais apropriadas:

`SELECT cidade.descricao, subregiao.descricao FROM cidade LEFT OUTER JOIN subregiao ON cidade.codigo_subregiao = subregiao.codigo_subregiao AND subregiao.descricao = "Toscana"`

descricao	descricao_1
Curitiba	Null
Sao Paulo	Null
Guarulhos	Null
Buenos Aires	Null
La Plata	Null
Cordoba	Null
San Francisco	Null
Los Angeles	Null
Orlando	Null
Miami	Null
Siena	Toscana

Florenca	Toscana
Milao	Null
Yokohama	Null

SELECT cidade.descricao, subregiao.descricao FROM cidade RIGHT OUTER JOIN subregiao ON cidade.codigo_subregiao = subregiao.codigo_subregiao AND subregiao.descricao = "Toscana"

descricao	descricao_1
Null	Parana
Null	Sao Paulo
Null	Rio Grande do Sul
Null	Buenos Aires
Null	Cordoba
Null	California
Null	Florida
Siena	Toscana
Florenca	Toscana
Null	Lombardia
Null	Aquitania
Null	Borgonha

Null	Calabria
Null	Massachussetts
Null	Chiapas

O mais correto para esta consulta é utilizar

```
SELECT cidade.descricao, subregiao.descricao FROM cidade INNER JOIN subregiao USING (codigo_subregiao) WHERE subregiao.descricao = "Toscana"
```

ou

```
SELECT cidade.descricao, subregiao.descricao FROM cidade INNER JOIN subregiao ON cidade.codigo_subregiao = subregiao.codigo_subregiao AND subregiao.descricao = "Toscana"
```

que retornarão o mesmo resultado:

nome_cidade	nome_subregiao
Siena	Toscana
Florenca	Toscana

A utilização de JOINS pode parecer complicada, no entanto, ele existe para tornar mais fácil a elaboração das consultas. Espero que tenham compreendido e qualquer dúvida que aparecer pode entrar em contato comigo pelo meu [e-mail](mailto:ribafs@ribafs.net). Até a próxima semana.

EXEMPLOS DE JOINS SOFISTICADOS

```
SELECT event.name, comment.comment
FROM event, comment
WHERE event.id=comment.event_id;
```

```
SELECT event.name, comment.comment
FROM event
INNER JOIN comment
ON event.id=comment.event_id;
```

```
SELECT event.name, comment.comment
FROM event
LEFT JOIN comment ON
```

```
event.id=comment.event_id;
```

```
SELECT event.name, comment.comment  
FROM event  
RIGHT JOIN comment  
ON event.id=comment.event_id;
```

```
SELECT event.name, comment.comment  
FROM comment  
RIGHT JOIN event  
ON event.id=comment.event_id;
```

Join

Non-equijoin – função de unir tabelas sem campos em comun.

```
select a.nome, b.codigo  
from cd a, cd, cat b  
where a.preco between b.menor_preco and b.maior_preco;
```

União Regular (inner join ou equi-join)

São os join que tem a cláusula WHERE unindo a PK com a FK das tabelas afetadas.

```
select cd.cod, gravadora.nome  
from cd, gravadora  
where cd.cod_grav = gravadora.cod_grav;
```

Sintaxe alternativa (quando a PK e a FK têm o mesmo nome):

```
select cd.cod, gravadora.nome  
from cd natural join gravadora;
```

Apelidos em Tabelas

```
select a.codigo, b.nome  
from cd a, gravadora b  
where a.codigo = b.codigo;
```

Unindo mais de duas Tabelas

```
select a.nome, b.numero, c.nome
```

```
from cd a, faixa b, musica c
where a.codigo in(1,2)
and a.codigo = b.codigo
and b.codigo = c.codigo;
```

Outer Join no PostgreSQL (com SQL padrão):

```
SELECT *
FROM t1 LEFT OUTER JOIN t2 ON (t1.col = t2.col);
```

ou

```
SELECT *
FROM t1 LEFT OUTER JOIN t2 USING (col);
```

Mais detalhes em:

http://imasters.uol.com.br/artigo/6374/bancodedados/consultas_com_joins/imprimir/

<http://www.imasters.com.br/artigo.php?cn=2867&cc=23>

http://imasters.uol.com.br/artigo/2870/postgresql/juncao_entre_tabelas_no_postgresql_-_parte_02/

6) Utilizando Operadores

- 6.1) Introdução aos operadores
- 6.2) Entendendo os Operadores de texto
- 6.3) Entendendo as Expressões regulares
- 6.4) Entendendo os Operadores matemáticos
- 6.5) Entendendo a importância da Conversão de tipos

6.1) Introdução aos operadores

Um operador é algo que você alimenta com um ou mais valores e que devolve outro valor.

Operador é quem liga duas constantes ou variáveis. Temos operadores matemáticos, de string, de data, de atribuição, lógicos, de array, etc.

Seguem alguns exemplos.

Exemplo - Resolução do tipo em operador de exponenciação

Existe apenas um operador de exponenciação definido no catálogo, e recebe argumentos do tipo double precision. O rastreador atribui o tipo inicial integer aos os dois argumentos desta expressão de consulta:

```
=> SELECT 2 ^ 3 AS "Expressão";
```

```
expressao
-----
      8
(1 linha)
```

Portanto, o analisador faz uma conversão de tipo nos dois operandos e a consulta fica equivalente a

```
=> SELECT CAST(2 AS double precision) ^ CAST(3 AS double precision) AS "exp";
```

Exemplo - Resolução do tipo em operador de concatenação de cadeia de caracteres

Uma sintaxe estilo cadeia de caracteres é utilizada para trabalhar com tipos cadeias de caracteres, assim como para trabalhar com tipos de extensão complexa. Cadeias de caracteres de tipo não especificado se correspondem com praticamente todos os operadores candidatos.

Um exemplo com um argumento não especificado:

```
=> SELECT text 'abc' || 'def' AS "texto e desconhecido";
```

```
texto e desconhecido
-----
abcdef
(1 linha)
```

Neste caso o analisador procura pela existência de algum operador recebendo o tipo text nos dois argumentos. Uma vez que existe, assume que o segundo argumento deve ser interpretado como sendo do tipo text.

Concatenação de tipos não especificados:

```
=> SELECT 'abc' || 'def' AS "não especificado";
```

```
não especificado
-----
abcdef
(1 linha)
```

Neste caso não existe nenhuma pista inicial do tipo a ser usado, porque não foi especificado nenhum tipo na consulta. Portanto, o analisador procura todos os operadores candidatos, e descobre que existem candidatos aceitando tanto cadeia de caracteres quanto cadeia de bits como entrada. Como a categoria cadeia de caracteres é a preferida quando está disponível, esta categoria é selecionada e, depois, é usado o tipo preferido para cadeia de caracteres, text, como o tipo específico para solucionar os literais de tipo desconhecido.

Exemplo - Resolução do tipo em operador de valor absoluto e negação

O catálogo de operadores do PostgreSQL possui várias entradas para o **operador de prefixo @**, todas **implementando operações de valor absoluto** para vários tipos de dado numéricos. Uma destas entradas é para o tipo float8, que é o tipo preferido da categoria numérica. Portanto, o PostgreSQL usa esta entrada quando na presença de uma entrada não numérica:

```
=> SELECT @ '-4.5' AS "abs";
```

```
abs
-----
4.5
(1 linha)
```

Aqui o sistema realiza uma conversão implícita de text para float8 antes de aplicar o operador escolhido. Pode ser verificado que foi utilizado float8, e não algum outro tipo, escrevendo-se:

```
=> SELECT @ '-4.5e500' AS "abs";
```

```
ERRO: "-4.5e500" está fora da faixa para o tipo double precision
```

Por outro lado, o **operador de prefixo ~ (negação bit-a-bit)** é definido apenas para tipos de dado inteiros, e não para float8. Portanto, se tentarmos algo semelhante usando ~, resulta em:

```
=> SELECT ~ '20' AS "negação";
```

```
ERRO: operador não é único: ~ "unknown"
```

```
DICA: Não foi possível escolher um operador candidato melhor.  
Pode ser necessário adicionar uma conversão de tipo explícita.
```

Isto acontece porque o sistema não pode decidir qual dos vários **operadores ~ (negativo)** possíveis deve ser o preferido. Pode ser dada uma ajuda usando uma conversão explícita:

```
=> SELECT ~ CAST('20' AS int8) AS "negação";
```

```
   negação  
-----  
          -21  
(1 linha)
```

Mais detalhes em: <http://pgdocptbr.sourceforge.net/pg80/typeconv-oper.html>

Precedência dos operadores (decrecente)

Operador/Elemento	Associatividade	Descrição
.	esquerda	separador de nome de tabela/coluna
::	esquerda	conversão de tipo estilo PostgreSQL
[]	esquerda	seleção de elemento de matriz
-	direita	menos unário
^	esquerda	exponenciação
* / %	esquerda	multiplicação, divisão, módulo
+ -	esquerda	adição, subtração
IS		IS TRUE, IS FALSE, IS UNKNOWN, IS NULL
ISNULL		teste de nulo
NOTNULL		teste de não nulo
(qualquer outro)	esquerda	os demais operadores nativos e os definidos pelo usuário
IN		membro de um conjunto
BETWEEN		contido em um intervalo
OVERLAPS		sobreposição de intervalo de tempo


```

      maiusculas
-----
À AÇÃO SEQUÊNCIA

=> CREATE FUNCTION minusculas(text) RETURNS text AS '
'>   SELECT translate( lower($1),
'>       text ' 'ÁÉÍÓÚÀÈÌÒÙÃÕÂÊÎÔÛÄËÏÖÇ' ',
'>       text ' 'áéíóúàèìòùãõâêîôöäëïöç' ')
'> ' LANGUAGE SQL STRICT;

=> SELECT minusculas('À AÇÃO SEQUÊNCIA');
      minusculas
-----
à ação sequência

```

6.3) Entendendo as Expressões regulares

Correspondência com padrão (Expressões regulares)

O PostgreSQL disponibiliza três abordagens distintas para correspondência com padrão:

o operador LIKE tradicional do SQL;

o operador mais recente SIMILAR TO (adicionado ao SQL:1999);

e as expressões regulares no estilo POSIX.

Além disso, também está disponível a função de correspondência com padrão substring, que utiliza expressões regulares tanto no estilo SIMILAR TO quanto no estilo POSIX.

Dica: Havendo necessidade de correspondência com padrão acima destas, deve ser considerado o desenvolvimento de uma função definida pelo usuário em Perl ou Tcl.

LIKE

```

cadeia_de_caracteres LIKE padrão [ESCAPE caractere_de_escape]
cadeia_de_caracteres NOT LIKE padrão [ESCAPE caractere_de_escape]

```

Cada padrão define um conjunto de cadeias de caracteres. A expressão LIKE retorna verdade se a cadeia_de_caracteres estiver contida no conjunto de cadeias de caracteres representado pelo padrão; como esperado, a expressão NOT LIKE retorna falso quando LIKE retorna verdade, e vice-versa, e a expressão equivalente é NOT (cadeia_de_caracteres LIKE padrão).

Quando o padrão não contém os caracteres percentagem ou sublinhado, o padrão representa apenas a própria cadeia de caracteres; neste caso LIKE atua como o operador igual. No padrão o caractere sublinhado (_) representa (corresponde a) qualquer um único caractere; o caractere percentagem (%) corresponde a qualquer cadeia com zero ou mais caracteres.

Alguns exemplos:

```

'abc' LIKE 'abc'      verdade
'abc' LIKE 'a%'      verdade
'abc' LIKE '_b_'      verdade
'abc' LIKE 'c'        falso

```

Pode ser utilizada a palavra chave ILIKE no lugar de LIKE para fazer a correspondência não diferenciar letras maiúsculas de minúsculas, conforme o idioma ativo. [1] Isto não faz parte do padrão SQL, sendo uma extensão do PostgreSQL.

O operador ~ equiva ao LIKE, enquanto ~* corresponde ao ILIKE.

Também existem os operadores !~ e !~*, representando o NOT LIKE e o NOT ILIKE respectivamente. **Todos estes operadores são específicos do PostgreSQL.**

Expressões regulares do SIMILAR TO

```
cadeia_de_caracteres SIMILAR TO padrão [ESCAPE caractere_de_escape]  
cadeia_de_caracteres NOT SIMILAR TO padrão [ESCAPE caractere_de_escape]
```

O operador SIMILAR TO retorna verdade ou falso conforme o padrão corresponda ou não à cadeia de caracteres fornecida. Este operador é **muito semelhante ao LIKE, exceto por interpretar o padrão utilizando a definição de expressão regular do padrão SQL**. As expressões regulares do padrão SQL são um cruzamento curioso entre a notação do LIKE e a notação habitual das expressões regulares.

Da mesma forma que o LIKE, o operador SIMILAR TO somente é bem-sucedido quando o padrão corresponde a toda cadeia de caracteres; é diferente do praticado habitualmente nas expressões regulares, onde o padrão pode corresponder a qualquer parte da cadeia de caracteres.

Também como o LIKE, o operador SIMILAR TO utiliza _ e % como caracteres curinga, representando qualquer um único caractere e qualquer cadeia de caracteres, respectivamente (são comparáveis ao . e ao .* das expressões regulares POSIX).

Além destas funcionalidades tomadas emprestada do LIKE, o **SIMILAR TO suporta os seguintes metacaracteres para correspondência com padrão pegos emprestado das expressões regulares POSIX:**

- | representa alternância (uma das duas alternativas).
- * representa a repetição do item anterior zero ou mais vezes.
- + representa a repetição do item anterior uma ou mais vezes.
- Os parênteses () podem ser utilizados para agrupar itens em um único item lógico.
- A expressão de colchetes [...] especifica uma classe de caracteres, do mesmo modo que na expressão regular POSIX.

Deve ser observado que as repetições limitadas (? e {...}) não estão disponíveis, embora existam no POSIX. Além disso, **o ponto (.) não é um metacaractere**.

Da mesma forma que no LIKE, a contrabarra desativa o significado especial de qualquer um dos metacaracteres; ou pode ser especificado um caractere de escape diferente por meio da cláusula ESCAPE.

Alguns exemplos:

```
'abc' SIMILAR TO 'abc'      verdade
'abc' SIMILAR TO 'a'        falso
'abc' SIMILAR TO '%(b|d)%'  verdade
'abc' SIMILAR TO '(b|c)%'   falso
```

A função substring com três parâmetros, substring(cadeia_de_caracteres FROM padrão FOR caractere_de_escape), permite extrair a parte da cadeia de caracteres que corresponde ao padrão da expressão regular SQL:1999. Assim como em SIMILAR TO, o padrão especificado deve corresponder a toda a cadeia de caracteres, senão a função falha e retorna nulo. Para indicar a parte do padrão que deve ser retornada em caso de sucesso, o padrão deve conter duas ocorrências do caractere de escape seguidas por aspas ("). É retornado o texto correspondente à parte do padrão entre estas marcas.

Alguns exemplos:

```
substring('foobar' FROM '%"o_b#"%' FOR '#')    oob
substring('foobar' FROM '#%"o_b#"%' FOR '#')    NULL
```

Expressões regulares POSIX

A tabela abaixo mostra os operadores disponíveis para correspondência com padrão utilizando as expressões regulares POSIX.

Operadores de correspondência para expressões regulares

Operador	Descrição	Exemplo
~	Corresponde à expressão regular, diferenciando maiúsculas e minúsculas	'thomas' ~ '.*thomas.*'
~*	Corresponde à expressão regular, não diferenciando maiúsculas e minúsculas	'thomas' ~* '.*Thomas.*'
!~	Não corresponde à expressão regular, diferenciando maiúsculas e minúsculas	'thomas' !~ '.*Thomas.*'
!~*	Não corresponde à expressão regular, não diferenciando maiúsculas e minúsculas	'thomas' !~* '.*vadim.*'

As expressões regulares POSIX fornecem uma forma mais poderosa para correspondência com padrão que os operadores LIKE e SIMILAR TO. Muitas ferramentas do Unix, como egrep, sed e awk, utilizam uma linguagem para correspondência com padrão semelhante à descrita aqui.

Uma expressão regular é uma sequência de caracteres contendo uma definição abreviada de um

conjunto de cadeias de caracteres (um *conjunto regular*). Uma cadeia de caracteres é dita correspondendo a uma expressão regular se for membro do conjunto regular descrito pela expressão regular. Assim como no LIKE, os caracteres do padrão correspondem exatamente aos caracteres da cadeia de caracteres, a não ser quando forem caracteres especiais da linguagem da expressão regular — porém, as expressões regulares utilizam caracteres especiais diferentes dos utilizados pelo LIKE. Diferentemente dos padrões do LIKE, uma expressão regular pode corresponder a qualquer parte da cadeia de caracteres, a não ser que a expressão regular seja explicitamente ancorada ao início ou ao final da cadeia de caracteres.

Alguns exemplos:

```
'abc' ~ 'abc'      verdade
'abc' ~ '^a'       verdade
'abc' ~ '(b|d)'    verdade
'abc' ~ '^ (b|c) ' falso
```

A função substring com dois parâmetros, substring(cadeia_de_caracteres FROM padrão), permite extrair a parte da cadeia de caracteres que corresponde ao padrão da expressão regular POSIX. A função retorna nulo quando não há correspondência, senão retorna a parte do texto que corresponde ao padrão. Entretanto, quando o padrão contém parênteses, é retornada a parte do texto correspondendo à primeira subexpressão entre parênteses (aquela cujo abre parênteses vem primeiro). Podem ser colocados parênteses envolvendo toda a expressão, se for desejado utilizar parênteses em seu interior sem disparar esta exceção. Se for necessária a presença de parênteses no padrão antes da subexpressão a ser extraída, veja os parênteses não-capturantes descritos abaixo.

Alguns exemplos:

```
substring('foobar' from 'o.b')      oob
substring('foobar' from 'o(.)b')    o
```

As expressões regulares do PostgreSQL são implementadas utilizando um pacote escrito por [Henry Spencer](#). Grande parte da descrição das expressões regulares abaixo foi copiada textualmente desta parte de seu manual.

Abaixo está mostrado o script usado para criar e carregar a tabela:

```
\!chcp 1252
CREATE TABLE textos(texto VARCHAR(40));
INSERT INTO textos VALUES ('www.apache.org');
INSERT INTO textos VALUES ('pgdocptbr.sourceforge.net');
INSERT INTO textos VALUES ('WWW.PHP.NET');
INSERT INTO textos VALUES ('www-130.ibm.com');
INSERT INTO textos VALUES ('Julia Margaret Cameron');
INSERT INTO textos VALUES ('Sor Juana Inés de la Cruz');
INSERT INTO textos VALUES ('Inês Pedrosa');
INSERT INTO textos VALUES ('Amy Semple McPherson');
INSERT INTO textos VALUES ('Mary McCarthy');
INSERT INTO textos VALUES ('Isabella Andreine');
INSERT INTO textos VALUES ('Jeanne Marie Bouvier de la Motte Guyon');
INSERT INTO textos VALUES ('Maria Tinteretto');
INSERT INTO textos VALUES ('');
INSERT INTO textos VALUES (' '||chr(9)||chr(10)||chr(11)||chr(12)||chr(13));
INSERT INTO textos VALUES ('192.168.0.15');
INSERT INTO textos VALUES ('pgsql-bugs-owner@postgresql.org');
INSERT INTO textos VALUES ('00:08:54:15:E5:FB');
```

A seguir estão mostradas as consultas efetuadas juntamente com seus resultados:

- I. Selecionar textos contendo um ou mais caracteres de "a" até "z", seguidos por um ponto, seguido por um ou mais caracteres de "a" até "z", seguidos por um ponto, seguido por um ou mais caracteres de "a" até "z".

- PostgreSQL 8.0.0

```
=> SELECT texto FROM textos WHERE texto SIMILAR TO '([a-z]+) \. ([a-z]+) \. ([a-z]+) ' ;
-- ou
=> SELECT texto FROM textos WHERE texto ~ '^([a-z]+) \. ([a-z]+) \. ([a-z]+) $' ;
-- ou
=> SELECT texto FROM textos WHERE texto ~ '^([a-z]+) \. ([a-z]+) \. ([a-z]+) $' ;
```

```

          texto
-----
www.apache.org
pgdocptbr.sourceforge.net
```

- Oracle 10g

```
SQL> SELECT texto FROM textos WHERE REGEXP_LIKE(texto,
'^([a-z]+) \. ([a-z]+) \. ([a-z]+) $' );
-- ou
SQL> SELECT texto FROM textos WHERE REGEXP_LIKE(texto,
'^([a-z]+) \. ([a-z]+) \. ([a-z]+) $' );
```

```

      TEXTO
-----
www.apache.org
pgdocptbr.sourceforge.net
WWW.PHP.NET
```

6.4) Entendendo os Operadores matemáticos

São fornecidos operadores matemáticos para muitos tipos de dados do PostgreSQL. Para os tipos sem as convenções matemáticas habituais para todas as permutações possíveis (por exemplo, os tipos de data e hora), o comportamento real é descrito nas próximas seções.

Operadores matemáticos

Operador	Descrição	Exemplo	Resultado
+	adição	2 + 3	5
-	subtração	2 - 3	-1

Operador	Descrição	Exemplo	Resultado
*	multiplicação	2 * 3	6
/	divisão (divisão inteira trunca o resultado)	4 / 2	2
%	módulo (resto)	5 % 4	1
^	exponenciação	2.0 ^ 3.0	8
/	raiz quadrada	/ 25.0	5
/	raiz cúbica	/ 27.0	3
!	fatorial	5 !	120
!!	fatorial (operador de prefixo)	!! 5	120
@	valor absoluto	@ -5.0	5
&	AND bit a bit	91 & 15	11
	OR bit a bit	32 3	35
#	XOR bit a bit	17 # 5	20
~	NOT bit a bit	~1	-2
<<	deslocamento à esquerda bit a bit	1 << 4	16
>>	deslocamento à direita bit a bit	8 >> 2	2

Os operadores bit a bit trabalham somente em tipos de dado inteiros, enquanto os demais estão disponíveis para todos os tipos de dado numéricos.

A tabela abaixo mostra as funções matemáticas disponíveis. Nesta tabela "dp" significa double precision. Muitas destas funções são fornecidas em várias formas, com diferentes tipos de dado dos argumentos. Exceto onde estiver indicado, todas as formas das funções retornam o mesmo tipo de dado de seu argumento. As funções que trabalham com dados do tipo double precision são, em sua maioria, implementadas usando a biblioteca C do sistema hospedeiro; a precisão e o comportamento em casos limites podem, portanto, variar dependendo do sistema hospedeiro.

Detalhes em: <http://pgdocptbr.sourceforge.net/pg80/functions-math.html>
<http://www.postgresql.org/docs/8.3/interactive/functions-math.html>

6.5) Entendendo a importância da Conversão de tipos

Conversão de tipo

Os comandos SQL podem, intencionalmente ou não, usar tipos de dado diferentes na mesma expressão. O PostgreSQL possui muitas funcionalidades para processar expressões com mistura de tipos.

Em muitos casos não há necessidade do usuário compreender os detalhes do mecanismo de conversão de tipo. Entretanto, as conversões implícitas feitas pelo PostgreSQL podem afetar o resultado do comando. Quando for necessário, os resultados podem ser personalizados utilizando

uma conversão de tipo *explícita*.

Este capítulo apresenta os mecanismos e as convenções de conversão de tipo de dado do PostgreSQL.

Visão geral

A linguagem SQL é uma linguagem fortemente tipada, ou seja, todo item de dado possui um tipo de dado associado que determina seu comportamento e a utilização permitida. O PostgreSQL possui um sistema de tipo de dado extensível, muito mais geral e flexível do que o de outras implementações do SQL. Por isso, a maior parte do comportamento de conversão de tipo de dado do PostgreSQL é governado por regras gerais, em vez de heurísticas [1] *ad hoc* [2], permitindo, assim, expressões com tipos diferentes terem significado mesmo com tipos definidos pelo usuário.

O rastreador/analizador (*scanner/parser*) do PostgreSQL divide os elementos léxicos em apenas cinco categorias fundamentais: inteiros, números não inteiros, cadeias de caracteres, identificadores e palavras chave. As constantes dos tipos não numéricos são, em sua maioria, classificadas inicialmente como cadeias de caracteres. A definição da linguagem SQL permite especificar o nome do tipo juntamente com a cadeia de caracteres, e este mecanismo pode ser utilizado no PostgreSQL para colocar o analisador no caminho correto. Por exemplo, a consulta

```
=> SELECT text 'Origem' AS "local", point '(0,0)' AS "valor";
```

```
local  | valor
-----+-----
Origem | (0,0)
(1 linha)
```

possui duas constantes literais, dos tipos text e point. Se não for especificado um tipo para o literal cadeia de caracteres, então será atribuído inicialmente o tipo guardador de lugar unknown (desconhecido), a ser determinado posteriormente nos estágios descritos abaixo.

Existem quatro construções SQL fundamentais que requerem regras de conversão de tipo distintas no analisador do PostgreSQL:

Chamadas de função

Grande parte do sistema de tipo do PostgreSQL é construído em torno de um amplo conjunto de funções. As funções podem possuir um ou mais argumentos. Como o PostgreSQL permite a sobrecarga de funções, o nome da função, por si só, não identifica unicamente a função a ser chamada; o analisador deve selecionar a função correta baseando-se nos tipos de dado dos argumentos fornecidos.

Operadores

O PostgreSQL permite expressões com operadores unários (um só argumento) de prefixo e de sufixo, assim como operadores binários (dois argumentos). Assim como as funções, os operadores podem ser sobrecarregados e, portanto, existe o mesmo problema para selecionar o operador correto.

Armazenamento do valor

Os comandos SQL INSERT e UPDATE colocam os resultados das expressões em tabelas. As expressões nestes comandos devem corresponder aos tipos de dado das colunas de destino, ou talvez serem convertidas para estes tipos de dado.

Construções UNION, CASE e ARRAY

Como os resultados de todas as cláusulas SELECT de uma declaração envolvendo união devem aparecer em um único conjunto de colunas, deve ser feita a correspondência entre os tipos de dado dos resultados de todas as cláusulas SELECT e a conversão em um conjunto uniforme. Do mesmo modo, os resultados das expressões da construção CASE devem ser todos convertidos em um tipo de dado comum, para que a expressão CASE tenha, como um todo, um tipo de dado de saída conhecido. O mesmo se aplica às construções ARRAY.

Os catálogos do sistema armazenam informações sobre que conversões entre tipos de dado, chamadas de *casts*, são válidas, e como realizar estas conversões. Novas conversões podem ser adicionadas pelo usuário através do comando CREATE CAST (Geralmente isto é feito junto com a definição de novos tipos de dado. O conjunto de conversões entre os tipos nativos foi cuidadosamente elaborado, sendo melhor não alterá-lo).

É fornecida no analisador uma heurística adicional para permitir estimar melhor o comportamento apropriado para os tipos do padrão SQL. Existem diversas *categorias de tipo* básicas definidas: boolean, numeric, string, bitstring, datetime, timespan, geometric, network e a definida pelo usuário. Cada categoria, com exceção da definida pelo usuário, possui um ou mais *tipo preferido*, selecionado preferencialmente quando há ambigüidade. Na categoria definida pelo usuário, cada tipo é o seu próprio tipo preferido. As expressões ambíguas (àquelas com várias soluções de análise candidatas) geralmente podem, portanto, serem resolvidas quando existem vários tipos nativos possíveis, mas geram erro quando existem várias escolhas para tipos definidos pelo usuário.

Todas as regras de conversão de tipo foram projetadas com vários princípios em mente:

- As conversões implícitas nunca devem produzir resultados surpreendentes ou imprevisíveis.
- Tipos definidos pelo usuário, para os quais o analisador não possui nenhum conhecimento *a priori*, devem estar "acima" na hierarquia de tipo. Nas expressões com tipos mistos, os tipos nativos devem sempre ser convertidos no tipo definido pelo usuário (obviamente, apenas se a conversão for necessária).
- Tipos definidos pelo usuário não se relacionam. Atualmente o PostgreSQL não dispõe de informações sobre o relacionamento entre tipos, além das heurísticas codificadas para os tipos nativos e relacionamentos implícitos baseado nas funções e conversões disponíveis.
- Não deve haver nenhum trabalho extra do analisador ou do executor se o comando não

necessitar de conversão de tipo implícita, ou seja, se o comando estiver bem formulado e os tipos já se correspondem, então o comando deve prosseguir sem despendar tempo adicional no analisador, e sem introduzir chamadas de conversão implícita desnecessárias no comando.

Além disso, se o comando geralmente requer uma conversão implícita para a função, e se o usuário definir uma nova função com tipos corretos para os argumentos, então o analisador deve usar esta nova função, não fazendo mais a conversão implícita utilizando a função antiga.

Notas

- [1] *heurística* — Rubrica: informática. — método de investigação baseado na aproximação progressiva de um dado problema. Dicionário Eletrônico Houaiss da língua portuguesa 1.0 (N. do T.)
- [2] *ad hoc* — destinado a essa finalidade; feito exclusivamente para explicar o fenômeno que descreve e que não serve para outros casos, não dando margem a qualquer generalização (diz-se de regra, argumento, definição etc.) — Dicionário Eletrônico Houaiss da língua portuguesa 1.0 (N. do T.)

Detalhes em: <http://pgdocptbr.sourceforge.net/pg80/typeconv.html>

Funções

A função específica a ser utilizada em uma chamada de função é determinada de acordo com os seguintes passos.

Resolução do tipo em função

1. Selecionar no catálogo do sistema [pg_proc](#) as funções a serem consideradas. Se for utilizado um nome de função não qualificado, as funções consideradas serão aquelas com nome e número de argumentos corretos, visíveis no caminho de procura corrente (consulte a [Seção 5.8.3](#)). Se for fornecido um nome de função qualificado, somente serão consideradas as funções no esquema especificado.
 - a. Se forem encontradas no caminho de procura várias funções com argumentos do mesmo tipo, somente será considerada àquela que aparece primeiro no caminho. Mas as funções com argumentos de tipos diferentes serão consideradas em pé de igualdade, não importando a posição no caminho de procura.
2. Verificar se alguma função aceita exatamente os mesmos tipos de dado dos argumentos de entrada. Caso exista (só pode haver uma correspondência exata no conjunto de funções consideradas), esta é usada. Os casos envolvendo o tipo unknown nunca encontram correspondência nesta etapa.
3. Se não for encontrada nenhuma correspondência exata, verificar se a chamada de função parece ser uma solicitação trivial de conversão de tipo. Isto acontece quando a chamada de

função possui apenas um argumento, e o nome da função é o mesmo nome (interno) de algum tipo de dado. Além disso, o argumento da função deve ser um literal de tipo desconhecido, ou um tipo binariamente compatível com o tipo de dado do nome da função. Quando estas condições são satisfeitas, o argumento da função é convertido no tipo de dado do nome da função sem uma chamada real de função.

4. Procurar pela melhor correspondência.

- a. Desprezar as funções candidatas para as quais os tipos da entrada não correspondem, e nem podem ser convertidos (utilizando uma conversão implícita) para corresponder. Para esta finalidade é assumido que os literais do tipo unknown podem ser convertidos em qualquer tipo. Se permanecer apenas uma função candidata, então esta é usada; senão continuar na próxima etapa.
- b. Examinar todas as funções candidatas, e manter aquelas com mais correspondências exatas com os tipos da entrada (Para esta finalidade os domínios são considerados idênticos aos seus tipos base). Manter todas as funções candidatas se nenhuma possuir alguma correspondência exata. Se permanecer apenas uma função candidata, então esta é usada; senão continuar na próxima etapa.
- c. Examinar todas as funções candidatas, e manter aquelas que aceitam os tipos preferidos (da categoria de tipo do tipo de dado de entrada) em mais posições onde a conversão de tipo será necessária. Manter todas as candidatas se nenhuma aceitar o tipo preferido. Se permanecer apenas uma função candidata, esta será usada; senão continuar na próxima etapa.
- d. Se algum dos argumentos de entrada for do tipo "unknown", verificar as categorias de tipo aceitas nesta posição do argumento pelas funções candidatas remanescentes. Em cada posição, selecionar a categoria string se qualquer uma das candidatas aceitar esta categoria (este favorecimento em relação à cadeia de caracteres é apropriado, porque um literal de tipo desconhecido se parece com uma cadeia de caracteres). Senão, se todas as candidatas remanescentes aceitam a mesma categoria de tipo, selecionar esta categoria; senão falhar, porque a escolha correta não pode ser deduzida sem informações adicionais. Rejeitar agora as funções candidatas que não aceitam a categoria de tipo selecionada; além disso, se alguma função candidata aceitar o tipo preferido em uma dada posição do argumento, rejeitar as candidatas que aceitam tipos não preferidos para este argumento.
- e. Se permanecer apenas uma função candidata, este será usada; Se não permanecer nenhuma função candidata, ou se permanecer mais de uma candidata, então falhar.

Deve ser observado que as regras da "melhor correspondência" são idênticas para a resolução do tipo em operador e função. Seguem alguns exemplos.

Exemplo - Resolução do tipo do argumento em função de arredondamento

Existe apenas uma função round com dois argumentos (O primeiro é numeric e o segundo é integer). Portanto, a consulta abaixo converte automaticamente o primeiro argumento do tipo integer para numeric:

```
=> SELECT round(4, 4);
```

```
round
-----
4.0000
(1 linha)
```

Na verdade esta consulta é convertida pelo analisador em

```
=> SELECT round(CAST (4 AS numeric), 4);
```

Uma vez que inicialmente é atribuído o tipo numeric às constantes numéricas com ponto decimal, a consulta abaixo não necessita de conversão de tipo podendo, portanto, ser ligeiramente mais eficiente:

```
=> SELECT round(4.0, 4);
```

Exemplo - Resolução do tipo em função de subcadeia de caracteres

Existem diversas funções substr, uma das quais aceita os tipos text e integer. Se esta função for chamada com uma constante cadeia de caracteres de tipo não especificado, o sistema escolherá a função candidata que aceita o argumento da categoria preferida para string (que é o tipo text).

```
=> SELECT substr('1234', 3);
```

```
substr
-----
      34
(1 linha)
```

Se a cadeia de caracteres for declarada como sendo do tipo varchar, o que pode ser o caso se vier de uma tabela, então o analisador tenta converter para torná-la do tipo text:

```
=> SELECT substr(varchar '1234', 3);
```

```
substr
-----
      34
(1 linha)
```

Esta consulta é transformada pelo analisador para se tornar efetivamente:

```
=> SELECT substr(CAST (varchar '1234' AS text), 3);
```

Nota: O analisador descobre no catálogo [pg_cast](#) que os tipos text e varchar são binariamente compatíveis, significando que um pode ser passado para uma função que aceita o outro sem realizar qualquer conversão física. Portanto, neste caso, não é realmente inserida nenhuma chamada de conversão de tipo explícita.

E, se a função for chamada com um argumento do tipo integer, o analisador tentará convertê-lo em text:

```
=> SELECT substr(1234, 3);
```

```
substr
-----
      34
(1 linha)
```

Na verdade é executado como:

```
=> SELECT substr(CAST (1234 AS text), 3);
```

Esta transformação automática pode ser feita, porque existe uma conversão implícita de integer para text que pode ser chamada.

Armazenamento de valor

Os valores a serem inseridos na tabela são convertidos no tipo de dado da coluna de destino de acordo com as seguintes etapas.

Conversão de tipo para armazenamento de valor

1. Verificar a correspondência exata com o destino.
2. Senão, tentar converter a expressão no tipo de dado de destino. Isto será bem-sucedido se houver uma conversão registrada entre os dois tipos. Se a expressão for um literal de tipo desconhecido, o conteúdo do literal cadeia de caracteres será enviado para a rotina de conversão de entrada do tipo de destino.
3. Verificar se existe uma conversão de tamanho para o tipo de destino. Uma conversão de tamanho é uma conversão do tipo para o próprio tipo. Se for encontrada alguma no catálogo [pg_cast](#) aplicá-la à expressão antes de armazenar na coluna de destino. A função que implementa este tipo de conversão sempre aceita um parâmetro adicional do tipo integer, que recebe o comprimento declarado da coluna de destino (na verdade, seu valor atttypmod; a interpretação de atttypmod varia entre tipos de dado diferentes). A função de conversão é responsável por aplicar toda semântica dependente do comprimento, tal como verificação do tamanho ou truncamento.

Exemplo - Conversão de tipo no armazenamento de *character*

Para uma coluna de destino declarada como character(20), a seguinte declaração garante que o valor

armazenado terá o tamanho correto:

```
=> CREATE TABLE vv (v character(20));
=> INSERT INTO vv SELECT 'abc' || 'def';
=> SELECT v, length(v) FROM vv;
```

v	length
abcdef	20

(1 linha)

O que acontece realmente aqui, é que os dois literais desconhecidos são resolvidos como text por padrão, permitindo que o operador || seja resolvido como concatenação de text. Depois, o resultado text do operador é convertido em bpchar ("caractere completado com brancos", ou "*blank-padded char*", que é o nome interno do tipo de dado character) para corresponder com o tipo da coluna de destino (Uma vez que os tipos text e bpchar são binariamente compatíveis, esta conversão não insere nenhuma chamada real de função). Por fim, a função de tamanho bpchar(bpchar, integer) é encontrada no catálogo do sistema, e aplicada ao resultado do operador e comprimento da coluna armazenada. Esta função específica do tipo realiza a verificação do comprimento requerido, e adiciona espaços para completar.

Construções UNION, CASE e ARRAY

As construções UNION do SQL precisam unir tipos, que podem não ser semelhantes, para que se tornem um único conjunto de resultados. O algoritmo de resolução é aplicado separadamente a cada coluna de saída da consulta união. As construções INTERSECT e EXCEPT resolvem tipos não semelhantes do mesmo modo que UNION. As construções CASE e ARRAY utilizam um algoritmo idêntico para fazer a correspondência das expressões componentes e selecionar o tipo de dado do resultado.

Resolução do tipo em UNION, CASE e ARRAY

1. Se todas as entradas forem do tipo unknown, é resolvido como sendo do tipo text (o tipo preferido da categoria cadeia de caracteres). Senão, ignorar as entradas unknown ao escolher o tipo do resultado.
2. Se as entradas não-desconhecidas não forem todas da mesma categoria de tipo, falhar.
3. Escolher o primeiro tipo de entrada não-desconhecido que for o tipo preferido nesta categoria, ou que permita todas as entradas não-desconhecidas serem convertidas implicitamente no mesmo.
4. Converter todas as entradas no tipo selecionado.

Seguem alguns exemplos.

Exemplo - Resolução do tipo com tipos subespecificados em uma união

```
=> SELECT text 'a' AS "texto" UNION SELECT 'b';
```

texto
a


```
b
(2 linhas)
```

Neste caso, o literal de tipo desconhecido 'b' é resolvido como sendo do tipo text.

Exemplo - Resolução do tipo em uma união simples

```
=> SELECT 1.2 AS "numérico" UNION SELECT 1;
```

```
numérico
-----
      1
     1.2
(2 linhas)
```

O literal 1.2 é do tipo numeric, e o valor inteiro 1 pode ser convertido implicitamente em numeric, portanto este tipo é utilizado.

Exemplo - Resolução do tipo em uma união transposta

```
=> SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);
```

```
real
-----
      1
     2.2
(2 linhas)
```

Neste caso, como o tipo real não pode ser convertido implicitamente em integer, mas integer pode ser implicitamente convertido em real, o tipo do resultado da união é resolvido como real.

7) Reuso de Código: Utilizando Funções

- 7.1) Introdução às Funções
- 7.2) Utilizando Funções matemáticas
- 7.3) Utilizando Funções de data e hora
- 7.4) Utilizando Funções de Texto (Strings)
- 7.5) Utilizando Funções de Conversão de Tipos (Casts)
- 7.6) Utilizando Outras Funções
- 7.7) Entendendo as Funções de Agregação

7.1) Introdução às Funções

O PostgreSQL fornece um grande número de funções e operadores para os tipos de dado nativos. Os usuários também podem definir suas próprias funções e operadores, conforme descrito na [Parte V](#). Os comandos `\df` e `\do` do `psql` podem ser utilizados para mostrar a lista de todas as funções e operadores disponíveis, respectivamente.

Havendo preocupação quanto à portabilidade, deve-se ter em mente que a maioria das funções e operadores descritos neste capítulo, com exceção dos operadores mais triviais de aritmética e de comparação, além de algumas funções indicadas explicitamente, não são especificadas pelo padrão SQL. Algumas das funcionalidades estendidas estão presentes em outros sistemas gerenciadores de banco de dados SQL e, em muitos casos, estas funcionalidades são compatíveis e consistentes entre as várias implementações.

Detalhes em: <http://pgdocptbr.sourceforge.net/pg80/functions.html>

7.2) Utilizando Funções matemáticas

Operadores Matemáticos

`+`, `-`, `*`, `/`, `%` (somar, subtrair, multiplicar, dividir, módulo (resto de divisão de inteiros)),
`^`(potência),
`!`(fatorial),
`@`(valor absoluto)
`| /` - raiz quadrada (`| / 25.0 = 5`)
`|| /` - raiz cúbica (`|| / 27.0 = 3`)

Algumas funções Matemáticas

`ABS(x)` - valor absoluto de `x`
`CEIL(numeric)` - arredonda para o próximo inteiro superior
`DEGREES(valor)` - converte valor de radianos para graus
`FLOOR(numeric)` - arredonda para o próximo inteiro inferior
`MOD(x,y)` - resto da divisão de `x` por `y`
`PI()` - constante PI (3,1415...)

POWER(x,y) - x elevado a y
RADIANS(valor) - converte valor de graus para radianos
RANDOM() - valor aleatório entre 0 e 1
ROUND(numeric) - arredonda para o inteiro mais próximo
ROUND(v, d) - arredonda v com d casas decimais
SIGN(numeric) - retorna o sinal da entrada, como -1 ou +1
SQRT(X) - Raiz quadrada de X
TRUNC (numeric) - trunca para o nenhuma casa decimal
TRUNC (v numeric, s int) - trunca para s casas decimais

Operadores Lógicos:

AND, OR e NOT. TRUE, FALSE e NULL

Operadores de Comparação:

<, >, <=, >=, =, <> ou !=
a BETWEEN x AND y
a NOT BETWEEN x AND y
expressão IS NULL
expressão IS NOT NULL
expressão IS TRUE
expressão IS NOT TRUE
expressão IS FALSE
expressão IS NOT FALSE
expressão IS UNKNOWN
expressão IS NOT UNKNOWN

GREATEST - Retorna o maior valor de uma lista - SELECT GREATEST(1,4,6,8,2); - - 8

LEAST - Retorna o menor valor de uma lista.

Todos os valores da lista devem ser do mesmo tipo e nulos são ignorados.

Obs.: Ambas as funções acima não pertencem ao SQL standard, mas são uma extensão do PostgreSQL.

7.3) Utilizando Funções de data e hora

Operações com datas:

timestamp '2001-09-28 01:00' + interval '23 hours' -> timestamp '2001-09-29 00:00'
date '2001-09-28' + interval '1 hour' -> timestamp '2001-09-28 01:00'
date '01/01/2006' - date '31/01/2006'
time '01:00' + interval '3 hours'time -> '04:00'
interval '2 hours' - time '05:00' -> time '03:00:00'

Função age (retorna Interval) - Diferença entre datas

age(timestamp)interval (Subtrai de hoje)
age(timestamp '1957-06-13') -> 43 years 8 mons 3 days
age(timestamp, timestamp)interval Subtrai os argumentos
age('2001-04-10', timestamp '1957-06-13') -> 43 years 9 mons 27 days

Função extract (retorna double)

Extraí parte da data: ano, mês, dia, hora, minuto, segundo.
select extract(year from age('2001-04-10', timestamp '1957-06-13'))
select extract(month from age('2001-04-10', timestamp '1957-06-13'))
select extract(day from age('2001-04-10', timestamp '1957-06-13'))

Data e Hora atuais (retornam data ou hora)

SELECT CURRENT_DATE;
SELECT CURRENT_TIME;
SELECT CURRENT_TIME(0);
SELECT CURRENT_TIMESTAMP;
SELECT CURRENT_TIMESTAMP(0);

Somar dias e horas a uma data:

SELECT CAST('06/04/2006' AS DATE) + INTERVAL '27 DAYS' AS Data;

Função now (retorna timestamp with zone)

now() - Data e hora corrente (timestamp with zone);
Não usar em campos somente timestamp.

Função date_part (retorna double)

SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
Resultado: 16 (day é uma string, diferente de extract)

Obtendo o dia da data atual:

SELECT DATE_PART('DAY', CURRENT_TIMESTAMP) AS dia;

Obtendo o mês da data atual:

SELECT DATE_PART('MONTH', CURRENT_TIMESTAMP) AS mes;

Obtendo o ano da data atual:

SELECT DATE_PART('YEAR', CURRENT_TIMESTAMP) AS ano;

Função date_trunc (retorna timestamp)

SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
Retorna 2001-02-16 00:00:00

Convertendo (CAST)

select to_date('1983-07-18', 'YYYY-MM-DD')
select to_date('19830718', 'YYYYMMDD')

Função timeofday (retorna texto)

select timeofday() -> Fri Feb 24 10:07:32.000126 2006 BRT

Interval

interval [(p)]

to_char(interval '15h 2m 12s', 'HH24:MI:SS')

date '2001-09-28' + interval '1 hour'

interval '1 day' + interval '1 hour'

interval '1 day' - interval '1 hour'

900 * interval '1 second'

Interval trabalha com as unidades: second, minute, hour, day, week, month, year, decade, century, millenium ou abreviaturas ou plurais destas unidades.

Se informado sem unidades '13 10:38:14' será devidamente interpretado '13 days 10 hours 38 minutes 14 seconds'.

CURRENT_DATE - INTERVAL '1' day;

TO_TIMESTAMP('2006-01-05 17:56:03', 'YYYY-MM-DD HH24:MI:SS')

Operações com Datas

- A Diferença entre dois Timestamps é sempre um Interval

TIMESTAMP '2001-12-31' – TIMESTAMP '2001-12-11' = INTERVAL '19 days'

- Adicionar ou subtrair um Interval com um Timestamp produzirá um Timestamp

TIMESTAMP '2001-12-11' + INTERVAL '19 days' = TIMESTAMP '2001-12-30'

- Adicionar ou suntrair dois Interval acarreta em outro Interval:

INTERVAL '1 month' + INTERVAL '1 month 3 days' = INTERVAL '2 months 3 days'

Dica: Para a tradução podemos usar algumas funções que trabalham com string para varrer o resultado traduzindo.

- Não podemos efetuar operações de Adição, Multiplicação ou Divisão com dois Timestamps:

Causará Erro.

- A diferença entre dois valores tipo Date é um Integer representando o número de dias:

DATE '2001-12-30' – DATE '2001-12-11' = INTEGER 19

- Adicionando ou subtraindo Integer para um Date produzirá um Date:

DATE '2001-12-13' + INTEGER 7 = DATE '2001-12-20'

- Não podemos efetuar operações de Adição, Multiplicação ou Divisão com dois Dates:
Causará Erro.

Como testar estes exemplos no PostgreSQL?

```
SELECT DATE '2001-12-30' – DATE '2001-12-11';
```

7.4) Utilizando Funções de Texto (Strings)

Concatenação de Strings - dois || (pipes)

```
SELECT 'ae' || 'io' || 'u' AS vogais;    --vogais ----- aeiou  
SELECT CHR(67)||CHR(65)||CHR(84) AS "Dog"; -- Dog  CAT
```

Quantidade de Caracteres de String

char_length - retorna o número de caracteres

```
SELECT CHAR_LENGTH('Evolução'); - -Retorna 8
```

Ou SELECT LENGTH('Database'); - - Retorna 8

Converter para minúsculas

```
SELECT LOWER('UNIFORM');
```

Converter para maiúsculas

```
SELECT UPPER('universidade');
```

Posição de caractere

```
SELECT POSITION('@' IN 'ribafs@gmail.com'); -- Retorna 7
```

Ou SELECT STRPOS('Ribamar' , 'mar'); - - Retorna 5

Substring

SUBSTRING(string [FROM inteiro] [FOR inteiro])

```
SELECT SUBSTRING ('Ribamar FS' FROM 9 FOR 10); - - Retorna FS
```

SUBSTRING(string FROM padrão);

```
SELECT SUBSTRING ('PostgreSQL' FROM '.....'); - - Retorna Postgre
```

```
SELECT SUBSTRING ('PostgreSQL' FROM '...$'); - -Retorna SQL
```

Primeiros e últimos ...\$

Ou

SUBSTR ('string', inicio, quantidade);

```
SELECT SUBSTR ('Ribamar', 4, 3); - - Retorna mar
```

Substituir todos os caracteres semelhantes

```
SELECT TRANSLATE(string, velho, novo);
```

```
SELECT TRANSLATE('Brasil', 'il', 'ão'); - - Retorna Brasília
```

```
SELECT TRANSLATE('Brasileiro', 'eiro', 'eira');
```

Remover Espaços de Strings

```
SELECT TRIM(' SQL - PADRÃO ');
```

Calcular MD5 de String

```
SELECT MD5('ribafs'); - - Retorna 53cd5b2af18063bea8ddc804b21341d1
```

Repetir uma string n vezes

```
SELECT REPEAT('SQL-', 3); - - Retorna SQL-SQL-SQL-
```

Sobrescrever substring em string

```
SELECT REPLACE ('Postgresql', 'sql', 'SQL'); - - Retorna PostgreSQL
```

Dividir Cadeia de Caracteres com Delimitador

```
SELECT SPLIT_PART( 'PostgreSQL', 'gre', 2); - -Retorna SQL
```

```
SELECT SPLIT_PART( 'PostgreSQL', 'gre', 1); - -Retorna Post
```

<-----gre----->

Iniciais Maiúsculas

```
INITCAP(text) - INITCAP ('olá mundo') - - Olá Mundo
```

Remover Espaços em Branco

TRIM ([leading | trailing | both] [characters] from string)- remove caracteres da direita e da esquerda. trim (both 'b' from 'babacatebbbb'); - - abacate

RTRIM (string text, chars text) - Remove os caracteres chars da direita (default é espaço)
rtrim('removarrrr', 'r') - - remova

LTRIM - (string text, chars text) - Remove os caracteres chars da esquerda
ltrim('abssssremova', 'abs') - - remova

Detalhes no item 9.4 do Manual:

<http://pgdocptbr.sourceforge.net/pg80/functions-string.html>

Like e %

```
SELECT * FROM FRIENDS WHERE LASTNAME LIKE 'M%';
```

O ILIKE é case INsensitive e o LIKE case sensitive.

~~ equivale ao LIKE

~~* equivale ao ILIKE

!~~ equivale ao NOT LIKE

!~~* equivale ao NOT ILIKE

... LIKE '[4-6]_6%' -- Pegar o primeiro sendo de 4 a 6,
-- o segundo qualquer dígito,
-- o terceiro sendo 6 e os demais quaisquer

% similar a *

_ similar a ? (de arquivos no DOS)

Correspondência com um Padrão

O PostgreSQL disponibiliza três abordagens distintas para correspondência com padrão: o operador LIKE tradicional do SQL; o operador mais recente SIMILAR TO (adicionado ao SQL:1999); e as expressões regulares no estilo POSIX. Além disso, também está disponível a função de correspondência com padrão substring, que utiliza expressões regulares tanto no estilo SIMILAR TO quanto no estilo POSIX.

```
SELECT substring('XY1234Z', 'Y*([0-9]{1,3})'); -- Resultado: 123
```

```
SELECT substring('XY1234Z', 'Y*?([0-9]{1,3})'); -- Resultado: 1
```

SIMILAR TO

O operador SIMILAR TO retorna verdade ou falso conforme o padrão corresponda ou não à cadeia de caracteres fornecida. Este operador é muito semelhante ao LIKE, exceto por interpretar o padrão utilizando a definição de expressão regular do padrão SQL.

```
'abc' SIMILAR TO 'abc'    verdade
```

```
'abc' SIMILAR TO 'a' falso
```

```
'abc' SIMILAR TO '%(b|d)%' verdade
```

```
'abc' SIMILAR TO '(b|c)%' falso
```

```
SELECT 'abc' SIMILAR TO '%(b|d)%';    -- Procura b ou d em 'abc' e no caso retorna
TRUE
REGEXP
```

```
SELECT 'abc' ~ '.*ab.*';
```

~ distingue a de A

~* não distingue a de A

!~ distingue expressões distingue a de A

!~* distingue expressões não distingue a de A

```
'abc' ~ 'abc' -- TRUE
```

```
'abc' ~ '^a' -- TRUE
```

```
'abc' ~ '(b|j)' -- TRUE
```

```
'abc' ~ '^ (b|c)' -- FALSE
```

7.5) Utilizando Funções de Conversão de Tipos (Casts)

Conversão Explícita de Tipos (CAST)

CAST (expressão AS tipo) AS apelido; -- Sintaxe SQL ANSI

Outra forma:

Tipo (expressão);

Exemplo:

```
SELECT DATE '10/05/2002' - DATE '10/05/2001'; -- Retorna a quantidade de dias
          - -entre as duas datas
```

Para este tipo de conversão devemos:

Usar float8 ao invés de double precision;

Usar entre aspas alguns tipos como interval, time e timestamp

Obs.: aplicações portáteis devem evitar esta forma de conversão e em seu lugar usar o CAST explicitamente.

A função CAST() é utilizada para converter explicitamente tipos de dados em outros.

```
SELECT CAST(2 AS double precision) ^ CAST(3 AS double precision) AS "exp";
```

```
SELECT ~ CAST('20' AS int8) AS "negativo"; - Retorna -21
```

```
SELECT round(CAST (4 AS numeric), 4); - Retorna 4.0000
```

```
SELECT substr(CAST (1234 AS text), 3);
```

```
SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);
```

7.6) Utilizando Outras Funções

Tipos Geométricos:

```
CREATE TABLE geometricos(ponto POINT, segmento LSEG, retangulo BOX, poligono
POLYGON, circulo CIRCLE);
```

ponto (0,0),

segmento de (0,0) até (0,1),

retângulo (base inferior (0,0) até (1,0) e base superior (0,1) até (1,1)) e

círculo com centro em (1,1) e raio 1.

```
INSERT INTO geometricos VALUES ('(0,0)', '((0,0),(0,1))', '((0,0),(0,1))', '((0,0),(0,1),(1,1),
(1,0))', '((1,1),1)');
```

Tipos de Dados para Rede:

Para tratar especificamente de redes o PostgreSQL tem os tipos de dados cidr, inet e macaddr.

cidr – para redes IPV4 e IPV6

inet – para redes e hosts IPV4 e IPV6

macaddr – endereços MAC de placas de rede

Assim como tipos data, tipos de rede devem ser preferidos ao invés de usar tipos texto para guardar IPs, Máscaras ou endereços MAC.

Veja um exemplo em Índices Parciais e a documentação oficial para mais detalhes.

Formatação de Tipos de Dados

TO_CHAR - Esta função deve ser evitada, pois está prevista sua descontinuação.

TO_DATE

date TO_DATE(text, text); Recebe dois parâmetros text e retorna date.

Um dos parâmetros é a data e o outro o formato.

SELECT TO_DATE('29032006','DDMMYYYY'); - Retorna 2006-03-29

TO_TIMESTAMP

tmt TO_TIMESTAMP(text,text) - Recebe dois text e retorna timestamp with zone

SELECT TO_TIMESTAMP('29032006 14:23:05','DDMMYYYY HH:MI:SS'); - Retorna 2006-03-29 14:23:05+00

TO_NUMBER

numeric TO_NUMBER(text,text)

SELECT TO_NUMBER('12,454.8-', '99G999D9S'); Retorna -12454.8

SELECT TO_NUMBER('12,454.8-', '99G999D9'); Retorna 12454.8

SELECT TO_NUMBER('12,454.8-', '99999D9'); Retorna 12454

Detalhes no item 9.8 do manual oficial.

Funções Diversas

SELECT CURRENT_DATABASE();

SELECT CURRENT_SCHEMA();

SELECT CURRENT_SCHEMA(boolean);

SELECT CURRENT_USER;

SELECT SESSION_USER;

SELECT VERSION();

SELECT CURRENT_SETTING('DATESTYLE');

SELECT HAS_TABLE_PRIVILEGE('usuario','tabela','privilegio');

SELECT HAS_TABLE_PRIVILEGE('postgres','nulos','insert'); - - Retorna: t

SELECT HAS_DATABASE_PRIVILEGE('postgres','testes','create'); - - Retorna: t

SELECT HAS_SCHEMA_PRIVILEGE('postgres','public','create'); - - Retorna: t

SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);

Arrays

```
SELECT ARRAY[1,1,2,2,3,3]::INT[] = ARRAY[1,2,3];
SELECT ARRAY[1,2,3] = ARRAY[1,2,8];
SELECT ARRAY[1,3,5] || ARRAY[2,4,6];
SELECT 0 || ARRAY[2,4,6];
```

Array de char com 48 posições e cada uma com 2:
campo char(2) [48]

Funções Geométricas

```
area(objeto) - - area(box '((0,0), (1,1))');
center(objeto) - - center(box '((0,0), (1,2))');
diameter(circulo double) - - diameter(circle '((0,0), 2.0)');
height(box) - - height(box '((0,0), (1,1))');
length(objeto) - - length(path '((-1,0), (1,0))');
radius(circle) - - radius(circle '((0,0), 2.0)');
width(box) - - width(box '((0,0), (1,1))');
```

Funções para Redes

Funções cidr e inet

```
host(inet) - - host('192.168.1.5/24') - - 192.168.1.5
masklen(inet) - - masklen('192.168.1.5/24') - - 24
netmask(inet) - - netmask('192.168.1.5/24') - - 255.255.255.0
network(inet) - - network('192.168.1.5/24') - - 192.168.1.0/24
```

Função macaddr

```
trunc(macaddr) - - trunc(macaddr '12:34:34:56:78:90:ab') - - 12:34:56:00:00:00
```

Funções de Informação do Sistema

```
current_database()
current_schema()
current_schemas(boolean)
current_user()
inet_client_addr()
inet_client_port()
inet_server_addr()
inet_server_port()
pg_postmaster_start_time()
version()
has_table_privilege(user, table, privilege) - dá privilégio ao user na tabela
```

has_table_privilege(table, privilege) - dá privilégio ao usuário atual na tabela
has_database_privilege(user, database, privilege) - dá privilégio ao user no banco
has_function_privilege(user, function, privilege) - dá privilégio ao user na função
has_language_privilege(user, language, privilege) - dá privilégio ao user na linguagem
has_schema_privilege(user, schema, privilege) - dá privilégio ao user no esquema
has_tablespace_privilege(user, tablespace, privilege) - dá privilégio ao user no tablespace
current_setting(nome) - valor atual da configuração
set_config(nome, novovalor, is_local) - seta parâmetro de retorna novo valor

pg_start_backup(label text)
pg_stop_backup()
pg_column_size(qualquer)
pg_tablespace_size(nome)
pg_database_size(nome)
pg_relation_size(nome)
pg_total_relation_size(nome)
pg_size_pretty(bigint)

pg_ls_dir(diretorio)
pg_read_file(arquivo text, offset bigint, tamanho bigint)
pg_stat_file(arquivo text)

7.7) Entendendo as Funções de Agregação (Agrupamento)

As funções de agrupamento são usadas para contar o número de registros de uma tabela.

avg(expressão)
count(*)
count(expressão)
max(expressão)
min(expressão)
stddev(expressão)
sum(expressão)
variance(expressão)

Onde expressão, pode ser "ALL expressão" ou "DISTINCT expressão".

count(distinct expressão)

As funções de Agrupamento (agregação) não podem ser utilizadas na cláusula WHERE. Devem ser utilizadas entre o SELECT e o FROM.

Dica: Num SELECT que usa uma função agregada, as demais colunas devem fazer parte da cláusula GROUP BY. Somente podem aparecer após o SELECT ou na cláusula HAVING. De uso proibido nas demais cláusulas.

Dica2: Ao contar os registros de uma tabela com a função COUNT(campo) e esse campo for nulo em alguns registros, estes registros não serão computados, por isso cuidado com os nulos também nas funções de agregação. Somente o count(*) conta os nulos.

A cláusula HAVING normalmente vem precedida de uma cláusula GROUP BY e obrigatoriamente contém funções de agregação.

ALERTA: Retornam somente os registros onde o campo pesquisado seja diferente de NULL.

NaN - Not a Number (Não é um número)

UPDATE tabela SET campo1 = 'NaN';

Exemplos:

SELECT MIN(campo) AS "Valor Mínimo" FROM tabela;

Caso tenha problema com esta consulta use:

SELECT campo FROM tabela ORDER BY campo ASC LIMIT 1; -- trará o menor

SELECT MAX(campo) AS "Valor Máximo" FROM tabela;

Caso tenha problema com esta consulta use:

SELECT campo FROM tabela ORDER BY campo DESC LIMIT 1; -- trará o maior

8) Entendendo e Utilizando sub-consultas

8) Expressões de subconsulta

Esta seção descreve as expressões de subconsulta em conformidade com o padrão SQL disponíveis no PostgreSQL. Todas as formas das expressões documentadas nesta seção retornam resultados booleanos (verdade/falso).

8.1. EXISTS

`EXISTS (subconsulta)`

O argumento do EXISTS é uma declaração SELECT arbitrária, ou uma *subconsulta*. A subconsulta é processada para determinar se retorna alguma linha. Se retornar pelo menos uma linha, o resultado de EXISTS é "verdade"; se a subconsulta não retornar nenhuma linha, o resultado de EXISTS é "falso".

A subconsulta pode referenciar variáveis da consulta que a envolve, que atuam como constantes durante a execução da subconsulta.

A subconsulta geralmente só é processada até ser determinado se retorna pelo menos uma linha, e não até o fim. Não é recomendável escrever uma subconsulta que tenha efeitos colaterais (tal como chamar uma função de sequência); pode ser difícil prever se o efeito colateral ocorrerá ou não.

Como o resultado depende apenas de alguma linha ser retornada, e não do conteúdo da linha, normalmente não há interesse na saída da subconsulta. Uma convenção de codificação habitual é escrever todos os testes de EXISTS na forma EXISTS(SELECT 1 WHERE ...). Entretanto, existem exceções para esta regra, como as subconsultas que utilizam INTERSECT.

Este exemplo simples é como uma junção interna em col2, mas produz no máximo uma linha de saída para cada linha de tab1, mesmo havendo várias linhas correspondentes em tab2:

```
SELECT col1 FROM tab1
WHERE EXISTS(SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

Exemplo. Utilização das cláusulas CASE e EXISTS juntas

Neste exemplo a tabela frutas é consultada para verificar se o alimento é uma fruta ou não. Caso o alimento conste da tabela frutas é uma fruta, caso não conste não é uma fruta. Abaixo está mostrado o *script* utilizado para criar e carregar as tabelas e executar a consulta. [1]

```
CREATE TEMPORARY TABLE frutas (id SERIAL PRIMARY KEY, nome TEXT);
INSERT INTO frutas VALUES (DEFAULT, 'banana');
INSERT INTO frutas VALUES (DEFAULT, 'maçã');
CREATE TEMPORARY TABLE alimentos (id SERIAL PRIMARY KEY, nome TEXT);
INSERT INTO alimentos VALUES (DEFAULT, 'maçã');
INSERT INTO alimentos VALUES (DEFAULT, 'espinafre');
SELECT nome, CASE WHEN EXISTS (SELECT nome FROM frutas WHERE nome=a.nome)
                  THEN 'sim'
                  ELSE 'não'
                END AS fruta
FROM alimentos a;
```

Abaixo está mostrado o resultado da execução do *script*.

nome	fruta
maçã	sim
espinafre	não

8.2. IN

expressão IN (subconsulta)

O lado direito é uma subconsulta entre parênteses, que deve retornar exatamente uma coluna. A expressão à esquerda é processada e comparada com cada linha do resultado da subconsulta. O resultado do IN é "verdade" se for encontrada uma linha igual na subconsulta. O resultado é "falso" se não for encontrada nenhuma linha igual (incluindo o caso especial onde a subconsulta não retorna nenhuma linha).

Deve ser observado que, se o resultado da expressão à esquerda for nulo, ou se não houver nenhum valor igual à direita e uma das linhas à direita tiver o valor nulo, o resultado da construção IN será nulo, e não falso. Isto está de acordo com as regras normais do SQL para combinações booleanas de valores nulos.

Da mesma forma que no EXISTS, não é razoável assumir que a subconsulta será processada até o fim.

construtor_de_linha IN (subconsulta)

O lado esquerdo desta forma do IN é um construtor de linha, conforme descrito na [Seção 4.2.11](#). O lado direito é uma subconsulta entre parênteses, que deve retornar exatamente tantas colunas quantas forem as expressões na linha do lado esquerdo. As expressões do lado esquerdo são processadas e comparadas, por toda a largura, com cada linha do resultado da subconsulta. O resultado do IN é "verdade" se for encontrada uma linha igual na subconsulta. O resultado é "falso" se não for encontrada nenhuma linha igual (incluindo o caso especial onde a subconsulta não retorna nenhuma linha).

Da maneira usual, os valores nulos nas linhas são combinados de acordo com as regras normais para expressões booleana do SQL. As linhas são consideradas iguais se todos os seus membros correspondentes forem não-nulos e iguais; as linhas são diferentes se algum membro correspondente for não-nulo e diferente; senão o resultado da comparação é desconhecido (nulo). Se o resultado de todas as linhas for diferente ou nulo, com pelo menos um nulo, o resultado do IN será nulo.

Exemplo. Utilização das cláusulas CASE e IN juntas

Este exemplo é idêntico ao [Exemplo 9-16](#), só que utiliza a cláusula IN para executar a consulta, conforme mostrado abaixo. [\[2\]](#)

```
SELECT nome, CASE WHEN nome IN (SELECT nome FROM frutas)
                  THEN 'sim'
                  ELSE 'não'
                END AS fruta
FROM alimentos;
```

Abaixo está mostrado o resultado da execução do script.

nome		fruta
maçã		sim
espinafre		não

8.3. NOT IN

`expressão NOT IN (subconsulta)`

O lado direito é uma subconsulta entre parênteses, que deve retornar exatamente uma coluna. A expressão à esquerda é processada e comparada com cada linha do resultado da subconsulta. O resultado de NOT IN é "verdade" se somente forem encontradas linhas diferentes na subconsulta (incluindo o caso especial onde a subconsulta não retorna nenhuma linha). O resultado é "falso" se for encontrada alguma linha igual.

Deve ser observado que se o resultado da expressão à esquerda for nulo, ou se não houver nenhum valor igual à direita e uma das linhas à direita tiver o valor nulo, o resultado da construção NOT IN será nulo, e não verdade. Isto está de acordo com as regras normais do SQL para combinações booleanas de valores nulos.

Da mesma forma que no EXISTS, não é razoável assumir que a subconsulta será processada até o fim.

`construtor_de_linha NOT IN (subconsulta)`

O lado esquerdo desta forma do NOT IN é um construtor de linha, conforme descrito na [Seção 4.2.11](#). O lado direito é uma subconsulta entre parênteses, que deve retornar exatamente tantas colunas quantas forem as expressões na linha do lado esquerdo. As expressões do lado esquerdo são processadas e comparadas, por toda a largura, com cada linha do resultado da subconsulta. O resultado do NOT IN é "verdade" se somente forem encontradas linhas diferentes na subconsulta (incluindo o caso especial onde a subconsulta não retorna nenhuma linha). O resultado é "falso" se for encontrada alguma linha igual.

Da maneira usual, os valores nulos nas linhas são combinados de acordo com as regras normais para expressões booleana do SQL. As linhas são consideradas iguais se todos os seus membros correspondentes forem não-nulos e iguais; as linhas são diferentes se algum membro correspondente for não-nulo e diferente; senão o resultado da comparação é desconhecido (nulo). Se o resultado de todas as linhas for diferente ou nulo, com pelo menos um nulo, o resultado do NOT IN será nulo.

8.4. ANY/SOME

`expressão operador ANY (subconsulta)`
`expressão operador SOME (subconsulta)`

O lado direito é uma subconsulta entre parênteses, que deve retornar exatamente uma coluna. A expressão à esquerda é processada e comparada com cada linha do resultado da subconsulta usando o operador especificado, devendo produzir um resultado booleano. O resultado do ANY é "verdade" se for obtido algum resultado verdade. O resultado é "falso" se nenhum resultado verdade for encontrado (incluindo o caso especial onde a subconsulta não retorna nenhuma linha). [\[3\]](#)

SOME é sinônimo de ANY. IN equivale a = ANY.

Deve ser observado que se não houver nenhuma comparação bem sucedida, e pelo menos uma linha da direita gerar nulo como resultado do operador, o resultado da construção ANY será nulo, e não falso. Isto está de acordo com as regras normais do SQL para combinações booleanas de valores nulos.

Do mesmo modo que no EXISTS, não é razoável supor que a subconsulta será processada até o fim.

```
construtor_de_linha operador ANY (subconsulta)
construtor_de_linha operador SOME (subconsulta)
```

O lado esquerdo desta forma do ANY é um construtor de linha, conforme descrito na [Seção 4.2.11](#). O lado direito é uma subconsulta entre parênteses, que deve retornar exatamente tantas colunas quantas forem as expressões existentes na linha do lado esquerdo. As expressões do lado esquerdo são processadas e comparadas, por toda a largura, com cada linha do resultado da subconsulta utilizando o operador especificado. Atualmente, somente são permitidos os operadores = e <> em construções ANY para toda a largura da linha. O resultado do ANY é "verdade" se for encontrada alguma linha igual ou diferente, respectivamente. O resultado será "falso" se não for encontrada nenhuma linha deste tipo (incluindo o caso especial onde a subconsulta não retorna nenhuma linha).

Da maneira usual, os valores nulos nas linhas são combinados de acordo com as regras normais para expressões booleana do SQL. As linhas são consideradas iguais se todos os seus membros correspondentes forem não-nulos e iguais; as linhas são diferentes se algum membro correspondente for não-nulo e diferente; senão o resultado da comparação é desconhecido (nulo). Se houver pelo menos um resultado de linha nulo, então o resultado de ANY não poderá ser falso; será verdade ou nulo.

Exemplo. Utilização das cláusulas CASE e ANY juntas

Este exemplo é idêntico ao [Exemplo 9-16](#), só que utiliza a cláusula ANY para executar a consulta, conforme mostrado abaixo. [\[4\]](#)

```
SELECT nome, CASE WHEN nome = ANY (SELECT nome FROM frutas)
                  THEN 'sim'
                  ELSE 'não'
                END AS fruta
FROM alimentos;
```

Abaixo está mostrado o resultado da execução do script.

nome	fruta
maçã	sim
espinafre	não

8.5. ALL

```
expressão operador ALL (subconsulta)
```

O lado direito é uma subconsulta entre parênteses, que deve retornar exatamente uma coluna. A expressão à esquerda é processada e comparada com cada linha do resultado da subconsulta usando o operador especificado, devendo produzir um resultado booleano. O resultado do ALL é "verdade" se o resultado de todas as linhas for verdade (incluindo o caso especial onde a subconsulta não

retorna nenhuma linha). O resultado é "falso" se for encontrado algum resultado falso.

NOT IN equivale a ∇ ALL.

Deve ser observado que se todas as comparações forem bem-sucedidas, mas pelo menos uma linha da direita gerar nulo como resultado do operador, o resultado da construção ALL será nulo, e não verdade. Isto está de acordo com as regras normais do SQL para combinações booleanas de valores nulos.

Do mesmo modo que no EXISTS, não é razoável supor que a subconsulta será processada até o fim.

`construtor_de_linha operador ALL (subconsulta)`

O lado esquerdo desta forma do ALL é um construtor de linha, conforme descrito na [Seção 4.2.11](#). O lado direito é uma subconsulta entre parênteses, que deve retornar exatamente tantas colunas quantas forem as expressões existentes na linha do lado esquerdo. As expressões do lado esquerdo são processadas e comparadas, por toda a largura, com cada linha do resultado da subconsulta utilizando o operador especificado. Atualmente, somente são permitidos os operadores = e ∇ em construções ALL para toda a largura da linha. O resultado do ALL é "verdade" se todas as linhas da subconsulta forem iguais ou diferentes, respectivamente (incluindo o caso especial onde a subconsulta não retorna nenhuma linha). O resultado será "falso" se não for encontrada nenhuma linha que seja igual ou diferente, respectivamente.

Da maneira usual, os valores nulos nas linhas são combinados de acordo com as regras normais para expressões booleana do SQL. As linhas são consideradas iguais se todos os seus membros correspondentes forem não-nulos e iguais; as linhas são diferentes se algum membro correspondente for não-nulo e diferente; senão o resultado da comparação é desconhecido (nulo). Se houver pelo menos um resultado de linha nulo, então o resultado de ALL não poderá ser verdade; será falso ou nulo.

8.6. Comparação de toda a linha

`construtor_de_linha operador (subconsulta)`

O lado esquerdo é um construtor de linha, conforme descrito na [Seção 4.2.11](#). O lado direito é uma subconsulta entre parênteses, que deve retornar exatamente tantas colunas quantas forem as expressões existentes na linha do lado esquerdo. Além disso, a subconsulta não pode retornar mais de uma linha (se não retornar nenhuma linha o resultado será considerado nulo). As expressões do lado esquerdo são processadas e comparadas, por toda a largura, com cada linha do resultado da subconsulta. Atualmente, somente são permitidos os operadores = e ∇ para comparação por toda a largura da linha. O resultado é "verdade" se as duas linhas forem iguais ou diferentes, respectivamente

Da maneira usual, os valores nulos nas linhas são combinados de acordo com as regras normais para expressões booleana do SQL. As linhas são consideradas iguais se todos os seus membros correspondentes forem não-nulos e iguais; as linhas são diferentes se algum membro correspondente for não-nulo e diferente; senão o resultado da comparação é desconhecido (nulo).

Notas

- [1] Exemplo escrito pelo tradutor, não fazendo parte do manual original.
- [2] Exemplo escrito pelo tradutor, não fazendo parte do manual original.
- [3] SQL Server 2000 — SOME | ANY comparam um valor escalar com o conjunto de valores de uma única coluna. Sintaxe: expressão_escalar { = | < > | ! = | > | > = | ! > | < | < = | ! < } { SOME | ANY } (subconsulta). A subconsulta possui o conjunto de resultados de uma coluna, e o mesmo tipo de dado da expressão escalar. SOME e ANY retornam verdade quando a comparação especificada é verdade para qualquer par (expressão_escalar, x), onde x é um valor do conjunto de uma única coluna. Senão retorna falso. SOME | ANY (N. do T.)
- [4] Exemplo escrito pelo tradutor, não fazendo parte do manual original.

Fonte: <http://pgdocptbr.sourceforge.net/pg80/functions-subquery.html>

Sub-consultas

Tipos de Sub-consultas:

- Interna (retorna somente um registro)
- Interna (retorna mais de um registro)
- Interna (retorna mais de um registro e mais de um campo)

O resultado do select mais interno serve de base para o select mais externo.

Este tipo tem um bom desempenho.

Uma tabela é consultada e com o retorno será pesquisada a outra.

Pesquisar os produtos com preco maior que o médio.

```
select produto, preco
  from produtos
 where preco > select avg(preco)
                from produtos;
```

Este exemplo tem baixo desempenho, pois executará o select interno para cada registro e o resultado passa para o select externo.

```
select codigo, produto, preco
  from produtos p
 where preco > select avg(preco_venda)
               from produtos
               where codigo = p.codigo;
```

Dica: não usar order by no select interno.

Somente devemos usar um único order by em toda a consulta e este no select externo.

```
select codigo, produto, preco from produtos
  where codigo = select codigo from produtos
                 where cod_venda = 2
                 and valor > select preco from produtos where cod_venda=2;
```

Sub-consulta na cláusula HAVING

```
select codigo min(preco)
  from produtos
 group by codigo
 having min(preco) >
        select preco from produtos where cod_ven = 6;
```

Sub-consultas com EXISTS

Só mostra o resultado se o select interno retornar um ou mais registros.

```
select cod_fornecedor, fornecedor from fornecedores
  where exists select * from produtos
  where produtos.cod_fornecedor = fornecedores.cod_fornecedor;
```

Caso o select interno retorne:

0 registro – false
>= 1 registro – true

Sub-consulta de Múltiplas linhas

O operador deve ser operador de grupo, como: IN, ANY ou ALL.

IN

Saber que produtos tem preço igual ou menor que o preço de cada fornecedor.

```
1o) select min(preco) from produtos
      group by codigo_fornecedor;
```

Supor retorno de: 3, 8, 5

```
2o) Com os valores do retorno anterior
select codigo, nome, preco
      from produtos
      where preco in(3,8,5);
```

Ou

```
select codigo, nome, preco
      from produtos
      where preco in
            select min(preco)
                  from produtos
                  group by codigo_fornecedor;
```

ANY

```
select codigo, nome, preco
      from produtos
      where preco < ANY
            select avg(preco)
                  from produtos
                  group by codigo_fornecedor;
```

Sub-consultas com múltiplos campos

Geralmente é lenta.

```
select codigo, nome, codigo_fornecedor, codigo_venda
      from produtos
      where codigo_fornecedor != codigo_venda IN
            select codigo_fornecedor, codigo_venda
                  from produtos
                  group by codigo_fornecedor;
```

9) Alterando dados nas tabelas

- 9.1) Adicionando dados com Insert
- 9.2) Adicionando dados com Select
- 9.3) Modificando dados com Update
- 9.4) Removendo dados com Delete
- 9.5) Removendo dados com Truncate

9.1) Adicionando dados com Insert

Inserção de dados

A tabela recém-criada não contém dados. A primeira ação a ser realizada para o banco de dados ter utilidade é inserir dados. Conceitualmente, os dados são inseridos uma linha de cada vez. É claro que é possível inserir mais de uma linha, mas não existe maneira de inserir menos de uma linha por vez. Mesmo que se conheça apenas o valor de algumas colunas, deve ser criada uma linha completa.

Para criar uma linha é utilizado o comando [INSERT](#). Este comando requer o nome da tabela, e um valor para cada coluna da tabela. Por exemplo, considere a tabela produtos do [Capítulo 5](#):

```
CREATE TABLE produtos (  
    cod_prod    integer,  
    nome        text,  
    preco       numeric  
);
```

Um exemplo de comando para inserir uma linha é:

```
INSERT INTO produtos VALUES (1, 'Queijo', 9.99);
```

Os valores dos dados são colocados na mesma ordem que as colunas se encontram na tabela, separados por vírgula. Geralmente os valores dos dados são literais (constantes), mas também são permitidas expressões escalares.

A sintaxe mostrada acima tem como desvantagem ser necessário conhecer a ordem das colunas da tabela. Para evitar isto, as colunas podem ser relacionadas explicitamente. Por exemplo, os dois comandos mostrados abaixo possuem o mesmo efeito do comando mostrado acima:

```
INSERT INTO produtos (cod_prod, nome, preco) VALUES (1, 'Queijo', 9.99);  
INSERT INTO produtos (nome, preco, cod_prod) VALUES ('Queijo', 9.99, 1);
```

Muitos usuários consideram boa prática escrever sempre os nomes das colunas.

Se não forem conhecidos os valores de todas as colunas, as colunas com valor desconhecido podem ser omitidas. Neste caso, estas colunas são preenchidas com seu respectivo valor padrão. Por exemplo:

```
INSERT INTO produtos (cod_prod, nome) VALUES (1, 'Queijo');  
INSERT INTO produtos VALUES (1, 'Queijo');
```

A segunda forma é uma extensão do PostgreSQL, que preenche as colunas a partir da esquerda com quantos valores forem fornecidos, e as demais com o valor padrão.

Para ficar mais claro, pode ser requisitado explicitamente o valor padrão da coluna individualmente, ou para toda a linha:

```
INSERT INTO produtos (cod_prod, nome, preco) VALUES (1, 'Queijo', DEFAULT);  
INSERT INTO produtos DEFAULT VALUES;
```

Dica: Para realizar "cargas volumosas", ou seja, inserir muitos dados, consulte o comando [COPY](#). Este comando não é tão flexível quanto o comando [INSERT](#), mas é mais eficiente.

Fonte: <http://pgdocptbr.sourceforge.net/pg80/dml.html>

9.2) Adicionando dados com Select

O exemplo abaixo insere algumas linhas na tabela filmes a partir da tabela temp_filmes com a mesma disposição de colunas da tabela filmes:

```
INSERT INTO filmes SELECT * FROM temp_filmes WHERE data_prod < '2004-05-07';
```

9.3) Modificando dados com Update

Atualização de dados

A modificação dos dados armazenados no banco de dados é referida como atualização. Pode ser atualizada uma linha, todas as linhas, ou um subconjunto das linhas da tabela. Uma coluna pode ser atualizada separadamente; as outras colunas não são afetadas.

Para realizar uma atualização são necessárias três informações:

1. O nome da tabela e da coluna a ser atualizada;
2. O novo valor para a coluna;
3. Quais linhas serão atualizadas.

Lembre-se que foi dito no [Capítulo 5](#) que o SQL, de uma maneira geral, não fornece um identificador único para as linhas. Portanto, não é necessariamente possível especificar diretamente a linha a ser atualizada. Em vez disso, devem ser especificadas as condições que a linha deve atender para ser atualizada. Somente havendo uma chave primária na tabela (não importando se foi declarada ou não), é possível endereçar uma linha específica com confiança, escolhendo uma condição correspondendo à chave primária. Ferramentas gráficas de acesso a banco de dados

dependem da chave primária para poderem atualizar as linhas individualmente.

Por exemplo, o comando mostrado abaixo atualiza todos os produtos com preço igual a 5, mudando estes preços para 10:

```
UPDATE produtos SET preco = 10 WHERE preco = 5;
```

Este comando pode atualizar nenhuma, uma, ou muitas linhas. Não é errado tentar uma atualização que não corresponda a nenhuma linha.

Vejamos este comando em detalhe: Primeiro aparece a palavra chave UPDATE seguida pelo nome da tabela. Como usual, o nome da tabela pode ser qualificado pelo esquema, senão é procurado no caminho. Depois aparece a palavra chave SET, seguida pelo nome da coluna, por um sinal de igual, e pelo novo valor da coluna. O novo valor da coluna pode ser qualquer expressão escalar, e não apenas uma constante. Por exemplo, se for desejado aumentar o preço de todos os produtos em 10% pode ser utilizado:

```
UPDATE produtos SET preco = preco * 1.10;
```

Como pode ser visto, a expressão para obter o novo valor pode fazer referência ao valor antigo. Também foi deixada de fora a cláusula WHERE. Quando esta cláusula é omitida, significa que todas as linhas da tabela serão atualizadas e, quando está presente, somente as linhas que atendem à condição desta cláusula serão atualizadas. Deve ser observado que o sinal de igual na cláusula SET é uma atribuição, enquanto o sinal de igual na cláusula WHERE é uma comparação, mas isto não cria uma ambigüidade. Obviamente, a condição da cláusula WHERE não é necessariamente um teste de igualdade, estão disponíveis vários outros operadores (consulte o [Capítulo 9](#)), mas a expressão deve produzir um resultado booleano.

Também pode ser atualizada mais de uma coluna pelo comando UPDATE, colocando mais de uma atribuição na cláusula SET. Por exemplo:

```
UPDATE minha_tabela SET a = 5, b = 3, c = 1 WHERE a > 0;
```

9.4) Removendo dados com Delete

Exclusão de dados

Até aqui foi mostrado como adicionar dados a tabelas, e como modificar estes dados. Está faltando mostrar como remover os dados que não são mais necessários. Assim como só é possível adicionar dados para toda uma linha, uma linha também só pode ser removida por inteiro da tabela. Na seção anterior foi explicado que o SQL não fornece uma maneira para endereçar diretamente uma determinada linha. Portanto, a remoção das linhas só pode ser feita especificando as condições que as linhas a serem removidas devem atender. Havendo uma chave primária na tabela, então é possível especificar exatamente a linha. Mas também pode ser removido um grupo de linhas atendendo a uma determinada condição, ou podem ser removidas todas as linhas da tabela de uma só vez.

É utilizado o comando [DELETE](#) para remover linhas; a sintaxe deste comando é muito semelhante a do comando [UPDATE](#). Por exemplo, para remover todas as linhas da tabela produtos possuindo

preço igual a 10:

```
DELETE FROM produtos WHERE preco = 10;
```

Se for escrito simplesmente

```
DELETE FROM produtos;
```

então todas as linhas da tabela serão excluídas! Dica de programador.

9.5) Removendo dados com Truncate

TRUNCATE -- esvazia a tabela

Sinopse

```
TRUNCATE [ TABLE ] nome
```

Descrição

O comando TRUNCATE remove rapidamente todas as linhas da tabela. Possui o mesmo efeito do comando DELETE não qualificado (sem WHERE), mas como na verdade não varre a tabela é mais rápido. É mais útil em tabelas grandes.

Parâmetros

nome

O nome (opcionalmente qualificado pelo esquema) da tabela a ser truncada.

Observações

O comando TRUNCATE não pode ser usado quando existe referência de chave estrangeira de outra tabela para a tabela. Neste caso a verificação da validade tornaria necessária a varredura da tabela, e o ponto central é não fazê-la.

O comando TRUNCATE não executa nenhum gatilho ON DELETE definido pelo usuário, porventura existente na tabela.

Exemplos

Truncar a tabela tbl_grande:

```
TRUNCATE TABLE tbl_grande;
```

Compatibilidade

Não existe o comando TRUNCATE no padrão SQL.

Manipulação de registros

INSERT insere um registro por vez.

Para inserir vários registros de cada vez usar:

```
INSERT INTO tabela (campo1, campo2) SELECT campo1, campo2 FROM tabela2;
```

```
INSERT INTO tabela SELECT * FROM tabela2;
```

Operadores Lógicos

... WHERE (c1=3 AND c2=5) OR c3=6;

- Não há limite na quantidade de operadores usados;
- Os operadores são avaliados da esquerda para a direita;
- É muito importante usar parênteses: legibilidade e garantem o que queremos. Sem eles pode ficar obscuro.

10) Filtrando dados nas tabelas

- 10.1) Filtrando dados com a cláusula where
- 10.2) Entendendo os operadores Like e Ilike
- 10.3) Utilizando o Operador Between
- 10.4) Utilizando o IN
- 10.5) Cuidados ao fazer Comparações com valores “NULL“
- 10.6) Utilizando a Cláusula Order By

10.1) Filtrando dados com a cláusula where

Rótulos de coluna

Podem ser atribuídos nomes para as entradas da lista de seleção para processamento posterior. Neste caso, "processamento posterior" é uma especificação opcional de classificação e o aplicativo cliente (por exemplo, os títulos das colunas para exibição). Por exemplo:

```
SELECT a AS valor, b + c AS soma FROM ...
```

Se nenhum nome de coluna de saída for especificado utilizando AS, o sistema atribui um nome padrão. Para referências a colunas simples, é o nome da coluna referenciada. Para chamadas de função, é o nome da função. Para expressões complexas o sistema gera um nome genérico.

Nota: Aqui, o nome dado à coluna de saída é diferente do nome dado na cláusula FROM (consulte a [Seção 7.2.1.2](#)). Na verdade, este processo permite mudar o nome da mesma coluna duas vezes, mas o nome escolhido na lista de seleção é o passado adiante.

DISTINCT

Após a lista de seleção ser processada, a tabela resultante pode opcionalmente estar sujeita à remoção das linhas duplicadas. A palavra chave DISTINCT deve ser escrita logo após o SELECT para especificar esta funcionalidade:

```
SELECT DISTINCT lista_de_seleção ...
```

(Em vez de DISTINCT pode ser utilizada a palavra ALL para especificar o comportamento padrão de manter todas as linhas)

Como é óbvio, duas linhas são consideradas distintas quando têm pelo menos uma coluna diferente. Os valores nulos são considerados iguais nesta comparação.

Como alternativa, uma expressão arbitrária pode determinar quais linhas devem ser consideradas distintas:

```
SELECT DISTINCT ON (expressão [, expressão ...]) lista_de_seleção ...
```

Neste caso, expressão é uma expressão de valor arbitrária avaliada para todas as linhas. Um conjunto de linhas para as quais todas as expressões são iguais são consideradas duplicadas, e somente a primeira linha do conjunto é mantida na saída. Deve ser observado que a "primeira linha" de um conjunto é imprevisível, a não ser que a consulta seja ordenada por um número suficiente de colunas para garantir a ordem única das linhas que chegam no filtro DISTINCT (o processamento de DISTINCT ON ocorre após a ordenação do ORDER BY).

A cláusula DISTINCT ON não faz parte do padrão SQL, sendo algumas vezes considerada um estilo ruim devido à natureza potencialmente indeterminada de seus resultados. Utilizando-se adequadamente GROUP BY e subconsultas no FROM esta construção pode ser evitada, mas geralmente é a alternativa mais fácil.

A cláusula WHERE

A sintaxe da [Cláusula WHERE](#) é

```
WHERE condição_de_pesquisa
```

onde a condição_de_pesquisa é qualquer expressão de valor (consulte a [Seção 4.2](#)) que retorne um valor do tipo boolean.

Após o processamento da cláusula FROM ter sido feito, cada linha da tabela virtual derivada é verificada com relação à condição de pesquisa. Se o resultado da condição for verdade, a linha é mantida na tabela de saída, senão (ou seja, se o resultado for falso ou nulo) a linha é desprezada. Normalmente a condição de pesquisa faz referência a pelo menos uma coluna da tabela gerada pela cláusula FROM; embora isto não seja requerido, se não for assim a cláusula WHERE não terá utilidade.

Nota: A condição de junção de uma junção interna pode ser escrita tanto na cláusula WHERE quanto na cláusula JOIN. Por exemplo, estas duas expressões de tabela são equivalentes:

```
FROM a, b WHERE a.id = b.id AND b.val > 5
```

e

```
FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5
```

ou talvez até mesmo

```
FROM a NATURAL JOIN b WHERE b.val > 5
```

Qual destas formas deve ser utilizada é principalmente uma questão de estilo. A sintaxe do JOIN na cláusula FROM provavelmente não é muito portátil para outros sistemas gerenciadores de banco de dados SQL. Para as junções externas não existe escolha em nenhum caso: devem ser feitas na cláusula FROM. A cláusula ON/USING da junção externa *não* é equivalente à condição WHERE, porque determina a adição de linhas (para as linhas de entrada sem correspondência) assim como a remoção de linhas do resultado final.

Abaixo estão mostrados alguns exemplos de cláusulas WHERE:

```
SELECT ... FROM fdt WHERE c1 > 5
```

```
SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)
```

```
SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)
```

```
SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)
```

```
SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)
AND 100
```

```
SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)
```

sendo que fdt é a tabela derivada da cláusula FROM. As linhas que não aderem à condição de pesquisa da cláusula WHERE são eliminadas de fdt. Deve ser observada a utilização de subconsultas escalares como expressões de valor. Assim como qualquer outra consulta, as subconsultas podem utilizar expressões de tabela complexas. Deve ser observado, também, como fdt é referenciada nas subconsultas. A qualificação de c1 como fdt.c1 somente será necessária se c1 também for o nome de uma coluna na tabela de entrada derivada da subconsulta. Entretanto, a qualificação do nome da coluna torna mais clara a consulta, mesmo quando não é necessária. Este exemplo mostra como o escopo do nome da coluna de uma consulta externa se estende às suas consultas internas.

As cláusulas GROUP BY e HAVING

Após passar pelo filtro WHERE, a tabela de entrada derivada pode estar sujeita ao agrupamento, utilizando a cláusula GROUP BY, e à eliminação de grupos de linhas, utilizando a cláusula HAVING.

```
SELECT lista_de_seleção
      FROM ...
      [WHERE ...]
      GROUP BY referência_a_coluna_de_agrupamento [,
referência_a_coluna_de_agrupamento]...
```

A [Cláusula GROUP BY](#) é utilizada para agrupar linhas da tabela que compartilham os mesmos valores em todas as colunas da lista. Em que ordem as colunas são listadas não faz diferença. O efeito é combinar cada conjunto de linhas que compartilham valores comuns em uma linha de grupo que representa todas as linhas do grupo. Isto é feito para eliminar redundância na saída, e/ou para calcular agregações aplicáveis a estes grupos. Por exemplo:

```
=> SELECT * FROM testel;
```

```
x | y
---+---
a | 3
c | 2
b | 5
a | 1
(4 linhas)
```

```
=> SELECT x FROM testel GROUP BY x;
```

```
x
---
a
b
c
(3 linhas)
```

Na segunda consulta não poderia ser escrito `SELECT * FROM teste1 GROUP BY x`, porque não existe um valor único da coluna `y` que poderia ser associado com cada grupo. As colunas agrupadas podem ser referenciadas na lista de seleção, desde que possuam um valor único em cada grupo.

De modo geral, se uma tabela for agrupada as colunas que não são usadas nos agrupamentos não podem ser referenciadas, exceto nas expressões de agregação. Um exemplo de expressão de agregação é:

```
=> SELECT x, sum(y) FROM teste1 GROUP BY x;
```

```
x | sum
---+-----
a |      4
b |      5
c |      2
(3 linhas)
```

Aqui `sum()` é a função de agregação que calcula um valor único para o grupo todo. Mais informações sobre as funções de agregação disponíveis podem ser encontradas na [Seção 9.15](#).

Dica: Um agrupamento sem expressão de agregação computa, efetivamente, o conjunto de valores distintas na coluna. Também poderia ser obtido por meio da cláusula `DISTINCT` (consulte a [Seção 7.3.3](#)).

Abaixo está mostrado um outro exemplo: cálculo do total das vendas de cada produto (e não o total das vendas de todos os produtos).

```
SELECT cod_prod, p.nome, (sum(v.unidades) * p.preco) AS vendas
FROM produtos p LEFT JOIN vendas v USING (cod_prod)
GROUP BY cod_prod, p.nome, p.preco;
```

Neste exemplo, as colunas `cod_prod`, `p.nome` e `p.preco` devem estar na cláusula `GROUP BY`, porque são referenciadas na lista de seleção da consulta (dependendo da forma exata como a tabela `produtos` for definida, as colunas `nome` e `preço` podem ser totalmente dependentes da coluna `cod_prod`, tornando os agrupamentos adicionais teoricamente desnecessários, mas isto ainda não está implementado). A coluna `v.unidades` não precisa estar na lista do `GROUP BY`, porque é usada apenas na expressão de agregação (`sum(...)`), que representa as vendas do produto. Para cada produto, a consulta retorna uma linha sumarizando todas as vendas do produto.

No SQL estrito, a cláusula `GROUP BY` somente pode agrupar pelas colunas da tabela de origem, mas o PostgreSQL estende esta funcionalidade para permitir o `GROUP BY` agrupar pelas colunas da lista de seleção. O agrupamento por expressões de valor, em vez de nomes simples de colunas, também é permitido.

Se uma tabela for agrupada utilizando a cláusula `GROUP BY`, mas houver interesse em alguns grupos apenas, pode ser utilizada a cláusula `HAVING`, de forma parecida com a cláusula `WHERE`, para eliminar grupos da tabela agrupada. A sintaxe é:

```
SELECT lista_de_seleção FROM ... [WHERE ...] GROUP BY ... HAVING
expressão_booleana
```

As expressões na cláusula HAVING podem fazer referência tanto a expressões agrupadas quanto a não agrupadas (as quais necessariamente envolvem uma função de agregação).

Exemplo:

```
=> SELECT x, sum(y) FROM teste1 GROUP BY x HAVING sum(y) > 3;
```

```
 x | sum
---+-----
 a |    4
 b |    5
(2 linhas)
```

```
=> SELECT x, sum(y) FROM teste1 GROUP BY x HAVING x < 'c';
```

```
 x | sum
---+-----
 a |    4
 b |    5
(2 linhas)
```

Agora vamos fazer um exemplo mais próximo da realidade:

```
SELECT cod_prod, p.nome, (sum(v.unidades) * (p.preco - p.custo)) AS lucro
FROM produtos p LEFT JOIN vendas v USING (cod_prod)
WHERE v.data > CURRENT_DATE - INTERVAL '4 weeks'
GROUP BY cod_prod, p.nome, p.preco, p.custo
HAVING sum(p.preco * v.unidades) > 5000;
```

No exemplo acima, a cláusula WHERE está selecionando linhas por uma coluna que não é agrupada (a expressão somente é verdadeira para as vendas feitas nas quatro últimas semanas, enquanto a cláusula HAVING restringe a saída aos grupos com um total de vendas brutas acima de 5000. Deve ser observado que as expressões de agregação não precisam ser necessariamente as mesmas em todas as partes da consulta.

Exemplo 7-1. Utilização de HAVING sem GROUP BY no SELECT

O exemplo abaixo mostra a utilização da cláusula HAVING sem a cláusula GROUP BY no comando SELECT. É criada a tabela produtos e são inseridas cinco linhas. Quando a cláusula HAVING exige a presença de mais de cinco linhas na tabela, a consulta não retorna nenhuma linha.

[\[4\]](#) [\[5\]](#)

```
=> create global temporary table produtos(codigo int, valor float);
=> insert into produtos values (1, 102);
=> insert into produtos values (2, 104);
=> insert into produtos values (3, 202);
=> insert into produtos values (4, 203);
=> insert into produtos values (5, 204);

=> select avg(valor) from produtos;
```

```
 avg
-----
 163
(1 linha)
```

```
=> select avg(valor) from produtos having count(*)>=5;
```

```
   avg
-----
   163
(1 linha)
```

```
=> select avg(valor) from produtos having count(*)=5;
```

```
   avg
-----
   163
(1 linha)
```

```
=> select avg(valor) from produtos having count(*)>5;
```

```
   avg
-----
(0 linhas)
```

10.2) Entendendo os operadores Like e Ilike

Correspondência com padrão

O PostgreSQL disponibiliza três abordagens distintas para correspondência com padrão: o operador LIKE tradicional do SQL; o operador mais recente SIMILAR TO (adicionado ao SQL:1999); e as expressões regulares no estilo POSIX. Além disso, também está disponível a função de correspondência com padrão substring, que utiliza expressões regulares tanto no estilo SIMILAR TO quanto no estilo POSIX.

Dica: Havendo necessidade de correspondência com padrão acima destas, deve ser considerado o desenvolvimento de uma função definida pelo usuário em Perl ou Tcl.

LIKE

```
cadeia_de_caracteres LIKE padrão [ESCAPE caractere_de_escape]
cadeia_de_caracteres NOT LIKE padrão [ESCAPE caractere_de_escape]
```

Cada padrão define um conjunto de cadeias de caracteres. A expressão LIKE retorna verdade se a cadeia_de_caracteres estiver contida no conjunto de cadeias de caracteres representado pelo padrão; como esperado, a expressão NOT LIKE retorna falso quando LIKE retorna verdade, e vice-versa, e a expressão equivalente é NOT (cadeia_de_caracteres LIKE padrão).

Quando o padrão não contém os caracteres percentagem ou sublinhado, o padrão representa apenas a própria cadeia de caracteres; neste caso LIKE atua como o operador igual. No padrão o caractere sublinhado (_) representa (corresponde a) qualquer um único caractere; o caractere percentagem (%) corresponde a qualquer cadeia com zero ou mais caracteres.

Alguns exemplos:

```
'abc' LIKE 'abc'      verdade
'abc' LIKE 'a%'       verdade
'abc' LIKE '_b_'      verdade
```



```
'abc' LIKE 'c'      falso
```

A correspondência com padrão LIKE sempre abrange toda a cadeia de caracteres. Para haver correspondência com o padrão em qualquer posição da cadeia de caracteres, o padrão deve começar e terminar pelo caractere percentagem.

Para corresponder ao próprio caractere sublinhado ou percentagem, sem corresponder a outros caracteres, estes caracteres devem ser precedidos pelo caractere de escape no padrão. O caractere de escape padrão é a contrabarra, mas pode ser definido um outro caractere através da cláusula ESCAPE. Para corresponder ao próprio caractere de escape, devem ser escritos dois caracteres de escape.

Deve ser observado que a contrabarra também possui significado especial nos literais cadeias de caracteres e, portanto, para escrever em uma constante um padrão contendo uma contrabarra devem ser escritas duas contrabarras no comando SQL. Assim sendo, para escrever um padrão que corresponda ao literal contrabarra é necessário escrever quatro contrabarras no comando, o que pode ser evitado escolhendo um caractere de escape diferente na cláusula ESCAPE; assim a contrabarra deixa de ser um caractere especial para o LIKE (Mas continua sendo especial para o analisador de literais cadeias de caracteres e, por isso, continuam sendo necessárias duas contrabarras).

Também é possível fazer com que nenhum caractere sirva de escape declarando ESCAPE ". Esta declaração tem como efeito desativar o mecanismo de escape, tornando impossível anular o significado especial dos caracteres sublinhado e percentagem no padrão.

Alguns exemplos: (N. do T.)

```
-- Neste exemplo a contrabarra única é consumida pelo analisador de literais
-- cadeias de caracteres, e não anula o significado especial do _
```

```
=> SELECT tablename FROM pg_tables WHERE tablename LIKE '%g\_o%' ESCAPE '\\';
```

```
tablename
-----
pg_group
pg_operator
pg_opclass
pg_largeobject
```

```
-- Neste exemplo somente uma das duas contrabarras é consumida pelo analisador
-- de literais cadeias de caracteres e, portanto, a segunda contrabarra anula
-- o significado especial do _
```

```
=> SELECT tablename FROM pg_tables WHERE tablename LIKE '%g\\\_o%' ESCAPE '\\';
```

```
tablename
-----
pg_operator
pg_opclass
```

```
-- No Oracle não são necessárias duas contrabarras, como mostrado abaixo:
```

```
SQL> SELECT view_name FROM all_views WHERE view_name LIKE 'ALL\\_%' ESCAPE '\\';
```

Pode ser utilizada a palavra chave ILIKE no lugar de LIKE para fazer a correspondência não diferenciar letras maiúsculas de minúsculas, conforme o idioma ativo. [1] Isto não faz parte do

padrão SQL, sendo uma extensão do PostgreSQL.

O operador `~~` equivale ao `LIKE`, enquanto `~~*` corresponde ao `ILIKE`. Também existem os operadores `!~~` e `!~~*`, representando o `NOT LIKE` e o `NOT ILIKE` respectivamente. Todos estes operadores são específicos do PostgreSQL.

Expressões regulares do SIMILAR TO

```
cadeia_de_caracteres SIMILAR TO padrão [ESCAPE caractere_de_escape]
cadeia_de_caracteres NOT SIMILAR TO padrão [ESCAPE caractere_de_escape]
```

O operador `SIMILAR TO` retorna verdade ou falso conforme o padrão corresponda ou não à cadeia de caracteres fornecida. Este operador é muito semelhante ao `LIKE`, exceto por interpretar o padrão utilizando a definição de expressão regular do padrão SQL. As expressões regulares do padrão SQL são um cruzamento curioso entre a notação do `LIKE` e a notação habitual das expressões regulares.

Da mesma forma que o `LIKE`, o operador `SIMILAR TO` somente é bem-sucedido quando o padrão corresponde a toda cadeia de caracteres; é diferente do praticado habitualmente nas expressões regulares, onde o padrão pode corresponder a qualquer parte da cadeia de caracteres. Também como o `LIKE`, o operador `SIMILAR TO` utiliza `_` e `%` como caracteres curinga, representando qualquer um único caractere e qualquer cadeia de caracteres, respectivamente (são comparáveis ao `.` e ao `.*` das expressões regulares POSIX).

Além destas funcionalidades pegadas emprestadas do `LIKE`, o `SIMILAR TO` suporta os seguintes metacaracteres para correspondência com padrão pegos emprestados das expressões regulares POSIX:

- `|` representa alternância (uma das duas alternativas).
- `*` representa a repetição do item anterior zero ou mais vezes.
- `+` representa a repetição do item anterior uma ou mais vezes.
- Os parênteses `()` podem ser utilizados para agrupar itens em um único item lógico.
- A expressão de colchetes `[...]` especifica uma classe de caracteres, do mesmo modo que na expressão regular POSIX.

Deve ser observado que as repetições limitadas (`?` e `{...}`) não estão disponíveis, embora existam no POSIX. Além disso, o ponto `.` não é um metacaractere.

Da mesma forma que no `LIKE`, a contrabarra desativa o significado especial de qualquer um dos metacaracteres; ou pode ser especificado um caractere de escape diferente por meio da cláusula `ESCAPE`.

Alguns exemplos:

```
'abc' SIMILAR TO 'abc'      verdade
'abc' SIMILAR TO 'a'        falso
'abc' SIMILAR TO '%(b|d)%'  verdade
'abc' SIMILAR TO '(b|c)%'   falso
```

A função `substring` com três parâmetros, `substring(cadeia_de_caracteres FROM padrão FOR caractere_de_escape)`, permite extrair a parte da cadeia de caracteres que corresponde ao padrão da expressão regular SQL:1999. Assim como em `SIMILAR TO`, o padrão especificado deve corresponder a toda a cadeia de caracteres, senão a função falha e retorna nulo. Para indicar a parte do padrão que deve ser retornada em caso de sucesso, o padrão deve conter duas ocorrências do

caractere de escape seguidas por aspas ("). É retornado o texto correspondente à parte do padrão entre estas marcas.

Alguns exemplos:

```
substring('foobar' FROM '%#"o_b#"%' FOR '#')    oob
substring('foobar' FROM '#_"o_b#"%' FOR '#')    NULL
```

10.3) Utilizando o Operador Between

Operadores de comparação

Estão disponíveis os operadores de comparação habituais, conforme mostrado na [Tabela 9-1](#).

Tabela 9-1. Operadores de comparação

Operador	Descrição
<	menor
>	maior
<=	menor ou igual
>=	maior ou igual
=	igual
<> ou !=	diferente

Nota: O operador != é convertido em <> no estágio de análise. Não é possível implementar os operadores != e <> realizando operações diferentes.

Os operadores de comparação estão disponíveis para todos os tipos de dado onde fazem sentido. Todos os operadores de comparação são operadores binários, que retornam valores do tipo boolean; expressões como $1 < 2 < 3$ não são válidas (porque não existe o operador < para comparar um valor booleano com 3).

Além dos operadores de comparação, está disponível a construção especial BETWEEN.

`a BETWEEN x AND y`

equivale a

`a >= x AND a <= y`

Analogamente,

`a NOT BETWEEN x AND y`

equivale a

`a < x OR a > y`

Não existe diferença entre as duas formas, além dos ciclos de CPU necessários para reescrever a primeira forma na segunda internamente.

Para verificar se um valor é nulo ou não, são usadas as construções

```
expressão IS NULL
expressão IS NOT NULL
```

ou às construções equivalentes, mas fora do padrão,

```
expressão ISNULL
expressão NOTNULL
```

Não deve ser escrito expressão = NULL, porque NULL não é "igual a" NULL (O valor nulo representa um valor desconhecido, e não se pode saber se dois valores desconhecidos são iguais). Este comportamento está de acordo com o padrão SQL.

Dica: Alguns aplicativos podem (incorretamente) esperar que expressão = NULL retorne verdade se o resultado da expressão for o valor nulo. É altamente recomendado que estes aplicativos sejam modificadas para ficarem em conformidade com o padrão SQL. Entretanto, se isto não puder ser feito, está disponível a variável de configuração [transform_null_equals](#). Quando [transform_null_equals](#) está ativado, o PostgreSQL converte as cláusulas x = NULL em x IS NULL. Este foi o comportamento padrão do PostgreSQL nas versões de 6.5 a 7.1.

O resultado dos operadores de comparação comuns é nulo (significando "desconhecido"), quando algum dos operandos é nulo. Outra forma de fazer comparação é com a construção IS DISTINCT FROM:

```
expressão IS DISTINCT FROM expressão
```

Para expressões não-nulas é o mesmo que o operador $\lt \gt$. Entretanto, quando as duas expressões são nulas retorna falso, e quando apenas uma expressão é nula retorna verdade. Portanto, atua efetivamente como se nulo fosse um valor de dado normal, em vez de "desconhecido".

Os valores booleanos também podem ser testados utilizando as construções

```
expressão IS TRUE
expressão IS NOT TRUE
expressão IS FALSE
expressão IS NOT FALSE
expressão IS UNKNOWN
expressão IS NOT UNKNOWN
```

Estas formas sempre retornam verdade ou falso, e nunca o valor nulo, mesmo quando o operando é nulo. A entrada nula é tratada como o valor lógico "desconhecido". Deve ser observado que IS UNKNOWN e IS NOT UNKNOWN são efetivamente o mesmo que IS NULL e IS NOT NULL, respectivamente, exceto que a expressão de entrada deve ser do tipo booleana.

10.4) Utilizando o IN

IN

expressão IN (subconsulta)

O lado direito é uma subconsulta entre parênteses, que deve retornar exatamente uma coluna. A expressão à esquerda é processada e comparada com cada linha do resultado da subconsulta. O resultado do IN é "verdade" se for encontrada uma linha igual na subconsulta. O resultado é "falso" se não for encontrada nenhuma linha igual (incluindo o caso especial onde a subconsulta não retorna nenhuma linha).

Deve ser observado que, se o resultado da expressão à esquerda for nulo, ou se não houver nenhum valor igual à direita e uma das linhas à direita tiver o valor nulo, o resultado da construção IN será nulo, e não falso. Isto está de acordo com as regras normais do SQL para combinações booleanas de valores nulos.

Da mesma forma que no EXISTS, não é razoável assumir que a subconsulta será processada até o fim.

construtor_de_linha IN (subconsulta)

O lado esquerdo desta forma do IN é um construtor de linha, conforme descrito na [Seção 4.2.11](#). O lado direito é uma subconsulta entre parênteses, que deve retornar exatamente tantas colunas quantas forem as expressões na linha do lado esquerdo. As expressões do lado esquerdo são processadas e comparadas, por toda a largura, com cada linha do resultado da subconsulta. O resultado do IN é "verdade" se for encontrada uma linha igual na subconsulta. O resultado é "falso" se não for encontrada nenhuma linha igual (incluindo o caso especial onde a subconsulta não retorna nenhuma linha).

Da maneira usual, os valores nulos nas linhas são combinados de acordo com as regras normais para expressões booleanas do SQL. As linhas são consideradas iguais se todos os seus membros correspondentes forem não-nulos e iguais; as linhas são diferentes se algum membro correspondente for não-nulo e diferente; senão o resultado da comparação é desconhecido (nulo). Se o resultado de todas as linhas for diferente ou nulo, com pelo menos um nulo, o resultado do IN será nulo.

Exemplo 9-17. Utilização das cláusulas CASE e IN juntas

Este exemplo é idêntico ao [Exemplo 9-16](#), só que utiliza a cláusula IN para executar a consulta, conforme mostrado abaixo. [\[2\]](#)

```
SELECT nome, CASE WHEN nome IN (SELECT nome FROM frutas)
                  THEN 'sim'
                  ELSE 'não'
                END AS fruta
FROM alimentos;
```

Abaixo está mostrado o resultado da execução do script.

nome	fruta
maçã	sim
espinafre	não

NOT IN

expressão NOT IN (subconsulta)

O lado direito é uma subconsulta entre parênteses, que deve retornar exatamente uma coluna. A expressão à esquerda é processada e comparada com cada linha do resultado da subconsulta. O resultado de NOT IN é "verdade" se somente forem encontradas linhas diferentes na subconsulta (incluindo o caso especial onde a subconsulta não retorna nenhuma linha). O resultado é "falso" se for encontrada alguma linha igual.

Deve ser observado que se o resultado da expressão à esquerda for nulo, ou se não houver nenhum valor igual à direita e uma das linhas à direita tiver o valor nulo, o resultado da construção NOT IN será nulo, e não verdade. Isto está de acordo com as regras normais do SQL para combinações booleanas de valores nulos.

Da mesma forma que no EXISTS, não é razoável assumir que a subconsulta será processada até o fim.

construtor_de_linha NOT IN (subconsulta)

O lado esquerdo desta forma do NOT IN é um construtor de linha, conforme descrito na [Seção 4.2.11](#). O lado direito é uma subconsulta entre parênteses, que deve retornar exatamente tantas colunas quantas forem as expressões na linha do lado esquerdo. As expressões do lado esquerdo são processadas e comparadas, por toda a largura, com cada linha do resultado da subconsulta. O resultado do NOT IN é "verdade" se somente forem encontradas linhas diferentes na subconsulta (incluindo o caso especial onde a subconsulta não retorna nenhuma linha). O resultado é "falso" se for encontrada alguma linha igual.

Da maneira usual, os valores nulos nas linhas são combinados de acordo com as regras normais para expressões booleana do SQL. As linhas são consideradas iguais se todos os seus membros correspondentes forem não-nulos e iguais; as linhas são diferentes se algum membro correspondente for não-nulo e diferente; senão o resultado da comparação é desconhecido (nulo). Se o resultado de todas as linhas for diferente ou nulo, com pelo menos um nulo, o resultado do NOT IN será nulo.

10.5) Cuidados ao fazer Comparações com valores “NULL”**OPERADOR NULL****NULLIF**

Função que atualiza campos com valor NULL.

update produtos

set codigo_fornecedor = NULLIF (codigo_fornecedor, 3);

Semelhante a:

```
if codigo_fornecedor = 3 then NULL
elseif codigo_fornecedor != 3 then codigo_fornecedor= codigo_fornecedor
end
```

COALESCE

Função tipo CASE que testa valores diferentes de NULL e retorna o primeiro não NULL.

```
select codigo, nome, codigo_fornecedor
       coalesce (codigo_fornecedor, codigo)
from produtos;
```

Similar a:

```
if exists (codigo_fornecedor) then retorne codigo_fornecedor
elseif not exists (codigo_fornecedor) then retorne codigo
end
```

Caso o primeiro for NULL procurará retornar o segundo. Se o segundo for NULL procurará retornar o terceiro até o último.

Em SQL NULL é para valores inexistentes. Regra geral: NULL se propaga, o que significa que com quem NULL se combina o resultado será um NULL.

NULL não zero, não é string vazia nem string de comprimento zero.

Um exemplo: num cadastro de alunos, para o aluno que ainda não se conhece a nota, não é correto usar zero para sua nota, mas sim NULL.

Não se pode efetuar cálculos de expressões onde um dos elementos é NULL.

COMPARANDO NULLs

NOT NULL com NULL -- Unknown

NULL com NULL -- Unknown

CONVERSÃO DE/PARA NULL

NULLIF() e COALESCE()

NULLIF(valor1, valor2)

NULLIF – Retorna NULL se, e somente se, valor1 e valor2 forem iguais, caso contrário retorna valor1.

Algo como:

```
if (valor1 == valor2){
then NULL
else valor1;
```

Retorna valor1 somente quando valor1 == valor2.

COALESCE – retorna o primeiro de seus argumentos que não for NULL. Só retorna NULL quando todos os seus argumentos forem NULL.

Uso: mudar o valor padrão cujo valor seja NULL.

```
create table nulos(nulo int, nulo2 int, nulo3 int);
```

```
insert into nulos values (1,null,null);
```

```
select coalesce(nulo, nulo2, nulo3) from nulos; - - Retorna 1, valor do campo nulo;
```

```
select coalesce(nulo2, nulo3) from nulos; - - Retorna NULL, pois ambos são NULL.
```

```
GREATEST - Retorna o maior valor de uma lista - SELECT GREATEST(1,4,6,8,2); - - 8
```

LEAST - Retorna o menor valor de uma lista.

Todos os valores da lista devem ser do mesmo tipo e nulos são ignorados.

Obs.: Ambas as funções acima não pertencem ao SQL standard, mas são uma extensão do PostgreSQL.

CONCATENANDO NULLs

A regra é: NULL se propaga. Qualquer que concatene com NULL gerará NULL.

STRING || NULL -- NULL

Usos:

- Como valor default para campos que futuramente receberão valor.
- Valor default para campos que poderão ser sempre inexistentes.

10.6) Utilizando a Cláusula Order By

ORDER BY - Ordena o resultado da consulta por um ou mais campos em ordem ascendente (ASC, default) ou descendente (DESC).

Exemplos:

```
ORDER BY cliente; -- pelo cliente e ascendente
```

```
ORDER BY cliente DESC; -- descendente
```

```
ORDER BY cliente, quantidade; -- pelo cliente e sub ordenado pela quantidade
```

```
ORDER BY cliente DESC, quant ASC;
```

No exemplo ordenando por dois campos:

```
SELECT * FROM pedidos ORDER BY cliente, quantidade;
```

A saída ficaria algo como:

Antônio – 1
Antônio – 2
João - 1
Pedro - 1
Pedro - 2

Combinação de NULL com FALSE, TRUE, NULL e NOT NULL:

select null and true - gera null

select null and false - gera false

select null and null - gera null

select null and not null - gera null

Referência: Livro SQL Curso Prático - Celso Henrique Poderoso de Oliveira
(Muito bom livro sobre a linguagem SQL)

NULO ou NÃO NULO, eis a questão...

NULL determina um estado e não um valor, por isso deve ser tratado de maneira especial. Quando queremos saber se um campo é nulo, a comparação a ser feita é "campo **is null**" e não "campo = null", sendo que essa última sempre retornará FALSO.

Do mesmo modo, para determinar se um campo não é nulo, usamos "campo **is not null**".

Você Sabia?

Os registros cujo campo é NULL são desprezados quando se utiliza uma função de agregação.

```
SELECT *  
FROM ACESSOS  
WHERE QTDE IS NULL;
```

Retorna todos os campos e todos os registros onde o campo QTDE é NULL.

Cláusula ORDER BY

Como o próprio nome sugere, essa cláusula ordena informações obtidas em uma query, de forma [ASC]endente (menor para o maior, e da letra A para Z), que é o padrão e pode ser omitida, ou de maneira [DESC]endente. O NULL, por ser um estado e não um valor, aparece antes de todos na ordenação. Alguns SGBDs permitem especificar se os nulls devem aparecer no início ou fim dos registros, como por exemplo o Firebird, com o uso das cláusulas **NULLS FIRST** e **NULLS LAST**.

11) Entendendo a execução das transações no PostgreSQL

Uso de Transações no PostgreSQL

Transação é cada execução de comando SQL que realiza leitura e ou escrita em bancos de dados. O PostgreSQL implementa transações resguardando as características ACID (Atomicidade, Consistência, Isolamento e Durabilidade).

Atomicidade – uma transação é totalmente executada ou totalmente revertida sem deixar efeitos no banco de dados.

Consistência – os resultados são coerentes com as operações realizadas.

Isolamento – a execução de uma transação não interfere nem sofre interferência das demais transações em execução.

Durabilidade – o resultado das transações deve ser persistido fisicamente no banco de dados.

No PostgreSQL todo comando que executamos é internamente executado em transações, que gerencia a manutenção das características ACID.

Mas mesmo que o PostgreSQL aja dessa forma podemos querer agrupar alguns comandos em uma única transação como outras vezes podemos desabilitar o gerenciamento do PostgreSQL.

Transações no PostgreSQL

Na maioria das vezes, ao executar comandos no SGBD, não percebemos que transações são executadas em segundo plano, mas algumas vezes é útil iniciar (begin) e finalizar (commit) transações manualmente.

```
BEGIN
```

```
update ...
```

```
update ...
```

```
END
```

Caso aconteça algum erro o PostgreSQL interrompe a transação com um ROLLBACK. Assim como também podemos executar um ROLLBACK manualmente para interromper a transação.

```
BEGIN
```

```
update ...
```

ROLLBACK

update ...

END

Um comando importante ao trabalhar com transações é o comando SET. Ele é usado para definir o nível de isolamento das transações.

SET TRANSACTION ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE }

De acordo com o SQL ANSI/ISO padrão, quatro níveis de isolamento de transações são definidos:

- Read uncommitted
- Read committed
- Repeatable read
- Serializable

O PostgreSQL suporta o read committed e serializable.

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');
```

Os detalhes destes comandos não são importantes aqui; o ponto importante é o fato de 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. Além disso, Alice não continuará sendo uma cliente satisfeita se o dinheiro for debitado de sua conta e não for creditado na conta do Bob. 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.

Desejamos, também, ter a garantia de estando a transação completa e aceita pelo sistema de banco de dados que a mesma fique definitivamente gravada, e não seja perdida mesmo no caso de acontecer uma pane logo em seguida. Por exemplo, se estiver sendo registrado um 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 do Bob 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 ao mesmo tempo, nenhuma delas 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 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*.

Nota: Algumas bibliotecas cliente emitem um comando BEGIN e um comando COMMIT automaticamente, fazendo com que se obtenha o efeito de um bloco de transação sem perguntar se isto é desejado. Verifique a documentação da interface utilizada.

É 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:

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

Fonte: <http://pgdocptbr.sourceforge.net/pg80/tutorial-transactions.html> e em: <http://www.postgresql.org/docs/8.3/static/tutorial-transactions.html>

Isolamento da transação

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

dirty read (leitura suja)

A transação lê dados escritos por uma transação simultânea não efetivada (*uncommitted*). [1]

nonrepeatable read (leitura que não pode ser repetida)

A transação lê novamente dados lidos anteriormente, e descobre que os dados foram alterados por outra transação (que os efetivou após ter sido feita a leitura anterior). [2]

phantom read (leitura fantasma)

A transação executa uma segunda vez uma consulta que retorna um conjunto de linhas que satisfazem uma determinada condição de procura, e descobre que o conjunto de linhas que satisfazem a condição é diferente por causa de uma outra transação efetivada recentemente.

[3]

Os quatro níveis de isolamento de transação, e seus comportamentos correspondentes, estão descritos na [Tabela 12-1](#).

Tabela 12-1. Níveis de isolamento da transação no SQL

Nível de isolamento	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possível	Possível	Possível
Read committed	Impossível	Possível	Possível
Repeatable read	Impossível	Impossível	Possível
Serializable	Impossível	Impossível	Impossível

No PostgreSQL pode ser requisitado qualquer um dos quatros níveis de isolamento padrão. Porém, internamente só existem dois níveis de isolamento distintos, correspondendo aos níveis de isolamento *Read Committed* e *Serializable*. Quando é selecionado o nível de isolamento *Read Committed* realmente obtém-se *Read Committed*, mas quando é selecionado *Repeatable Read* na realidade é obtido *Serializable*. Portanto, o nível de isolamento real pode ser mais estrito do que o selecionado. Isto é permitido pelo padrão SQL: os quatro níveis de isolamento somente definem quais fenômenos não podem acontecer, não definem quais fenômenos devem acontecer. O motivo pelo qual o PostgreSQL só disponibiliza dois níveis de isolamento, é porque esta é a única forma de mapear os níveis de isolamento padrão na arquitetura de controle de simultaneidade multiversão que faz sentido. O comportamento dos níveis de isolamento disponíveis estão detalhados nas próximas subseções.

É utilizado o comando [SET TRANSACTION](#) para definir o nível de isolamento da transação.

Nível de isolamento Read Committed

O *Read Committed* (lê efetivado) é o nível de isolamento padrão do PostgreSQL. Quando uma transação processa sob este nível de isolamento, o comando SELECT enxerga apenas os dados efetivados antes da consulta começar; nunca enxerga dados não efetivados, ou as alterações efetivadas pelas transações simultâneas durante a execução da consulta (Entretanto, o SELECT enxerga os efeitos das atualizações anteriores executadas dentro da sua própria transação, mesmo que ainda não tenham sido efetivadas). Na verdade, o comando SELECT enxerga um instantâneo do banco de dados, como este era no instante em que a consulta começou a executar. Deve ser observado que dois comandos SELECT sucessivos podem enxergar dados diferentes, mesmo estando dentro da mesma transação, se outras transações efetivarem alterações durante a execução do primeiro comando SELECT.

Os comandos UPDATE, DELETE e SELECT FOR UPDATE se comportam do mesmo modo que o SELECT para encontrar as linhas de destino: somente encontram linhas de destino efetivadas até o momento do início do comando. Entretanto, no momento em que foi encontrada alguma linha de destino pode ter sido atualizada (ou excluída ou marcada para atualização) por outra transação

simultânea. Neste caso, a transação que pretende atualizar fica aguardando a transação de atualização que começou primeiro efetivar ou desfazer (se ainda estiver executando). Se a transação de atualização que começou primeiro desfizer as atualizações, então seus efeitos são negados e a segunda transação de atualização pode prosseguir com a atualização da linha original encontrada. Se a transação de atualização que começou primeiro efetivar as atualizações, a segunda transação de atualização ignora a linha caso tenha sido excluída pela primeira transação de atualização, senão tenta aplicar sua operação na versão atualizada da linha. A condição de procura do comando (a cláusula WHERE) é avaliada novamente para verificar se a versão atualizada da linha ainda corresponde à condição de procura. Se corresponder, a segunda transação de atualização prossegue sua operação começando a partir da versão atualizada da linha.

Devido à regra acima, é possível um comando de atualização enxergar um instantâneo inconsistente: pode enxergar os efeitos dos comandos simultâneos de atualização que afetam as mesmas linhas que está tentando atualizar, mas não enxerga os efeitos destes comandos de atualização nas outras linhas do banco de dados. Este comportamento torna o *Read Committed* inadequado para os comandos envolvendo condições de procura complexas. Entretanto, é apropriado para casos mais simples. Por exemplo, considere a atualização do saldo bancário pela transação mostrada abaixo:

```
BEGIN;  
UPDATE conta SET saldo = saldo + 100.00 WHERE num_conta = 12345;  
UPDATE conta SET saldo = saldo - 100.00 WHERE num_conta = 7534;  
COMMIT;
```

Se duas transações deste tipo tentarem mudar ao mesmo tempo o saldo da conta 12345, é claro que desejamos que a segunda transação comece a partir da versão atualizada da linha da conta. Como cada comando afeta apenas uma linha predeterminada, permitir enxergar a versão atualizada da linha não cria nenhum problema de inconsistência.

Como no modo *Read Committed* cada novo comando começa com um novo instantâneo incluindo todas as transações efetivadas até este instante, de qualquer modo os próximos comandos na mesma transação vão enxergar os efeitos das transações simultâneas efetivadas. O ponto em questão é se, dentro de um *único* comando, é enxergada uma visão totalmente consistente do banco de dados.

O isolamento parcial da transação fornecido pelo modo *Read Committed* é adequado para muitos aplicativos, e este modo é rápido e fácil de ser utilizado. Entretanto, para aplicativos que efetuam consultas e atualizações complexas, pode ser necessário garantir uma visão do banco de dados com consistência mais rigorosa que a fornecida pelo modo *Read Committed*.

Nível de isolamento serializável

O nível *Serializable* fornece o isolamento de transação mais rigoroso. Este nível emula a execução serial das transações, como se todas as transações fossem executadas uma após a outra, em série, em vez de simultaneamente. Entretanto, os aplicativos que utilizam este nível de isolamento devem estar preparados para tentar executar novamente as transações, devido a falhas de serialização.

Quando uma transação está no nível serializável, o comando SELECT enxerga apenas os dados efetivados antes da transação começar; nunca enxerga dados não efetivados ou alterações efetivadas durante a execução da transação por transações simultâneas (Entretanto, o comando SELECT enxerga os efeitos das atualizações anteriores executadas dentro da sua própria transação, mesmo que ainda não tenham sido efetivadas). É diferente do *Read Committed*, porque o comando SELECT enxerga um instantâneo do momento de início da transação, e não do momento de início

do comando corrente dentro da transação. Portanto, comandos SELECT sucessivos dentro de uma mesma transação sempre enxergam os mesmos dados.

Os comandos UPDATE, DELETE e SELECT FOR UPDATE se comportam do mesmo modo que o comando SELECT para encontrar as linhas de destino: somente encontram linhas de destino efetivadas até o momento do início da transação. Entretanto, alguma linha de destino pode ter sido atualizada (ou excluída ou marcada para atualização) por outra transação simultânea no momento em que foi encontrada. Neste caso, a transação serializável aguarda a transação de atualização que começou primeiro efetivar ou desfazer as alterações (se ainda estiver executando). Se a transação que começou primeiro desfizer as alterações, então seus efeitos são negados e a transação serializável pode prosseguir com a atualização da linha original encontrada. Porém, se a transação que começou primeiro efetivar (e realmente atualizar ou excluir a linha, e não apenas selecionar para atualização), então a transação serializável é desfeita com a mensagem

ERRO: não foi possível serializar o acesso devido a atualização simultânea

porque uma transação serializável não pode alterar linhas alteradas por outra transação após a transação serializável ter começado.

Quando o aplicativo receber esta mensagem de erro deverá interromper a transação corrente, e tentar executar novamente toda a transação a partir do início. Da segunda vez em diante, a transação passa a enxergar a alteração efetivada anteriormente como parte da sua visão inicial do banco de dados e, portanto, não existirá conflito lógico em usar a nova versão da linha como ponto de partida para atualização na nova transação.

Deve ser observado que somente as transações que fazem atualizações podem precisar de novas tentativas; as transações somente para leitura nunca estão sujeitas a conflito de serialização.

O modo serializável fornece uma garantia rigorosa que cada transação enxerga apenas visões totalmente consistentes do banco de dados. Entretanto, o aplicativo deve estar preparado para executar novamente a transação quando atualizações simultâneas tornarem impossível sustentar a ilusão de uma execução serial. Como o custo de refazer transações complexas pode ser significativo, este modo é recomendado somente quando as transações efetuando atualizações contêm lógica suficientemente complexa a ponto de produzir respostas erradas no modo *Read Committed*. Habitualmente, o modo serializável é necessário quando a transação executa vários comandos sucessivos que necessitam enxergar visões idênticas do banco de dados.

Isolamento serializável versus verdadeira serialidade

O significado intuitivo (e a definição matemática) de execução "serializável" é que quaisquer duas transações simultâneas efetivadas com sucesso parecem ter sido executadas de forma rigorosamente serial, uma após a outra — embora qual das duas parece ter ocorrido primeiro não pode ser previsto antecipadamente. É importante ter em mente que proibir os comportamentos indesejáveis listados na [Tabela 12-1](#) não é suficiente para garantir a verdadeira serialidade e, de fato, o modo serializável do PostgreSQL *não garante a execução serializável neste sentido*. Como exemplo será considerada a tabela `minha_tabela` contendo inicialmente

classe	valor
1	10
1	20
2	100
2	200

Suponha que a transação serializável A calcula

```
SELECT SUM(valor) FROM minha_tabela WHERE classe = 1;
```

e insira o resultado (30) como valor em uma nova linha com classe = 2. Simultaneamente a transação serializável B calcula

```
SELECT SUM(valor) FROM minha_tabela WHERE classe = 2;
```

e obtém o resultado 300, que é inserido em uma nova linha com classe = 1. Em seguida as duas transações efetivam. Nenhum dos comportamentos não desejados ocorreu, ainda assim foi obtido um resultado que não poderia ter ocorrido serialmente em qualquer ordem. Se A tivesse executado antes de B, então B teria calculado a soma como 330, e não 300, e de maneira semelhante a outra ordem teria produzido uma soma diferente na transação A.

Para garantir serialidade matemática verdadeira, é necessário que o sistema de banco de dados imponha o *bloqueio de predicado*, significando que a transação não pode inserir ou alterar uma linha que corresponde à condição WHERE de um comando de outra transação simultânea. Por exemplo, uma vez que a transação A tenha executado o comando `SELECT ... WHERE class = 1`, o sistema de bloqueio de predicado proibiria a transação B inserir qualquer linha com classe igual a 1 até que A fosse efetivada. [4] Um sistema de bloqueio deste tipo é de implementação complexa e de execução extremamente dispendiosa, uma vez que todas as sessões devem estar cientes dos detalhes de todos os comandos executados por todas as transações simultâneas. E este grande gasto em sua maior parte seria desperdiçado, porque na prática a maioria dos aplicativos não fazem coisas do tipo que podem ocasionar problemas (Certamente o exemplo acima é bastante irreal, dificilmente representando um programa de verdade). Portanto, o PostgreSQL não implementa o bloqueio de predicado, e tanto quanto saibamos nenhum outro SGBD de produção o faz.

Nos casos em que a possibilidade de execução não serial representa um perigo real, os problemas podem ser evitados através da utilização apropriada de bloqueios explícitos. São mostrados mais detalhes nas próximas seções.

Notas

- [1] *dirty read* — A transação SQL T1 altera uma linha. Em seguida a transação SQL T2 lê esta linha antes de T1 executar o comando COMMIT. Se depois T1 executar o comando ROLLBACK, T2 terá lido uma linha que nunca foi efetivada e que, portanto, pode ser considerada como nunca tendo existido. (ISO-ANSI Working Draft) Foundation (SQL/Foundation), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)
- [2] *nonrepeatable read* — A transação SQL T1 lê uma linha. Em seguida a transação SQL T2 altera ou exclui esta linha e executa o comando COMMIT. Se T1 tentar ler esta linha novamente, pode receber o valor alterado ou descobrir que a linha foi excluída. (ISO-ANSI Working Draft) Foundation (SQL/Foundation), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)
- [3] *phantom read* — A transação SQL T1 lê um conjunto de linhas N que satisfazem a uma condição de procura. Em seguida a transação SQL T2 executa comandos SQL que geram uma ou mais linhas que satisfazem a condição de procura usada pela transação T1. Se depois a

transação SQL T1 repetir a leitura inicial com a mesma condição de procura, será obtida uma coleção diferente de linhas. (ISO-ANSI Working Draft) Foundation (SQL/Foundation), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)

[4] Em essência, o sistema de bloqueio de predicado evita leituras fantasmas restringindo o que é escrito, enquanto o MVCC evita restringindo o que é lido.

Fontes: <http://pgdocptbr.sourceforge.net/pg80/transaction-iso.html> e em: <http://www.postgresql.org/docs/8.3/static/transaction-iso.html>

SET TRANSACTION

Nome

SET TRANSACTION -- define as características da transação corrente

Sinopse

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

onde modo_da_transação é um entre:

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

Descrição

O comando SET TRANSACTION define as características da transação corrente. Não produz nenhum efeito nas próximas transações. O comando SET SESSION CHARACTERISTICS define as características da transação usadas como padrão nas próximas transações na sessão. Estes padrões podem ser mudados para uma transação individual pelo comando SET TRANSACTION.

[1] [2] [3] [4]

As características da transação disponíveis são o nível de isolamento da transação e o modo de acesso da transação (leitura/escrita ou somente para leitura).

O nível de isolamento de uma transação determina quais dados a transação pode ver quando outras transações estão processando ao mesmo tempo.

READ COMMITTED

O comando consegue ver apenas as linhas efetivadas (*commit*) antes do início da sua execução. Este é o padrão.

SERIALIZABLE

Todos os comandos da transação corrente podem ver apenas as linhas efetivadas antes da

primeira consulta ou comando de modificação de dados ter sido executado nesta transação.

O padrão SQL define dois níveis adicionais, READ UNCOMMITTED e REPEATABLE READ. No PostgreSQL READ UNCOMMITTED é tratado como READ COMMITTED, enquanto REPEATABLE READ é tratado como SERIALIZABLE.

O nível de isolamento da transação não pode ser mudado após a primeira consulta ou comando de modificação de dado (SELECT, INSERT, DELETE, UPDATE, FETCH ou COPY) da transação ter sido executado. Para obter informações adicionais sobre o isolamento de transações e controle de simultaneidade deve ser consultado o [Capítulo 12](#).

O modo de acesso da transação determina se a transação é para leitura/escrita, ou se é somente para leitura. Ler/escrever é o padrão. Quando a transação é somente para leitura, não são permitidos os seguintes comandos SQL: INSERT, UPDATE, DELETE e COPY FROM, se a tabela a ser escrita não for uma tabela temporária; todos os comandos CREATE, ALTER e DROP; COMMENT, GRANT, REVOKE, TRUNCATE; também EXPLAIN ANALYZE e EXECUTE se o comando a ser executado estiver entre os listados. Esta é uma noção de somente para leitura de alto nível, que não impede todas as escritas em disco.

Observações

Se for executado o comando SET TRANSACTION sem ser executado antes o comando START TRANSACTION ou BEGIN, parecerá que não produziu nenhum efeito, uma vez que a transação termina imediatamente.

É possível não utilizar o comando SET TRANSACTION, especificando o modo_da_transação desejado no comando BEGIN ou no comando START TRANSACTION.

Os modos de transação padrão da sessão também podem ser definidos através dos parâmetros de configuração [default_transaction_isolation](#) e [default_transaction_read_only](#) (De fato, SET SESSION CHARACTERISTICS é apenas uma forma verbosa equivalente a definir estas variáveis através do comando SET). Isto significa que os valores padrão podem ser definidos no arquivo de configuração, via ALTER DATABASE, etc. Para obter informações adicionais deve ser consultada a [Seção 16.4](#).

Compatibilidade

Os dois comandos estão definidos no padrão SQL. No padrão SQL SERIALIZABLE é o nível de isolamento padrão da transação; no PostgreSQL normalmente o padrão é READ COMMITTED, mas pode ser mudado conforme mencionado acima. Devido à falta de bloqueio de predicado, o nível SERIALIZABLE não é verdadeiramente serializável. Para obter mais informações deve ser consultada a [Capítulo 12](#).

No padrão SQL existe uma outra característica de transação que pode ser definida por estes comandos: o tamanho da área de diagnósticos. Este conceito é específico da linguagem SQL incorporada e, portanto, não é implementado no servidor PostgreSQL.

O padrão SQL requer a presença de vírgulas entre os modo_da_transação sucessivos, mas por razões históricas o PostgreSQL permite que estas vírgulas sejam omitidas.

Notas

- [1] Oracle — O comando SET TRANSACTION é utilizado para estabelecer a transação corrente como apenas de leitura ou de leitura e escrita, estabelecer o *nível de isolamento*, ou atribuir a sessão para um segmento de *rollback* especificado. As operações realizadas pelo comando SET TRANSACTION afetam apenas a transação corrente, não afetando outros usuários ou outras transações. A transação termina quando é executado o comando COMMIT ou o comando ROLLBACK. O banco de dados Oracle efetiva implicitamente a transação antes e após a execução de um comando da linguagem de definição de dados (DDL). A cláusula ISOLATION LEVEL especifica como as transações que contêm modificações no banco de dados são tratadas. A definição SERIALIZABLE especifica o modo de isolamento da transação como serializável, conforme definido no padrão SQL92. Se a transação serializável contiver comando da linguagem de manipulação de dados (DML) que tenta atualizar qualquer recurso que possa ter sido atualizado por uma transação não efetivada no início da transação serializável, então o comando da DML falhará. A definição READ COMMITTED é o comportamento padrão de transação do banco de dados Oracle. Se a transação contiver comandos da DML requerendo bloqueios de linha mantidos por outras transações, então o comando da DML aguardará até os bloqueios de linha serem liberados. [Oracle® Database SQL Reference 10g Release 1 \(10.1\) Part Number B10759-01](#) (N. do T.)
- [2] SQL Server — O comando SET TRANSACTION ISOLATION LEVEL controla o comportamento de bloqueio e versão da linha dos comandos Transact-SQL emitidos por uma conexão com o SQL Server. Os níveis de isolamento são: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SNAPSHOT e SERIALIZABLE. [SQL Server 2005 Books Online — SET TRANSACTION ISOLATION LEVEL \(Transact-SQL\)](#) (N. do T.)
- [3] DB2 — O comando SET CURRENT ISOLATION atribui um valor ao registrador especial CURRENT ISOLATION. Este comando não está sob controle da transação. [DB2 Version 9 for Linux, UNIX, and Windows](#) (N. do T.)
- [4] DB2 — O comando CHANGE ISOLATION LEVEL muda a maneira do DB2 isolar os dados de outros processos enquanto o banco de dados está sendo acessado. Os níveis de isolamento são: CS (cursor stability); NC (no commit). Não suportado pelo DB2; RR (repeatable read); RS (read stability); UR (uncommitted read). [DB2 Version 9 for Linux, UNIX, and Windows](#) (N. do T.)

Fontes: <http://pgdocptbr.sourceforge.net/pg80/sql-set-transaction.html> e em: <http://www.postgresql.org/docs/8.3/static/sql-set-transaction.html>

Transações com o PostgreSQL - Parte I

De Juliano Ignácio

Este será um artigo dividido em três partes. Trata-se de algo muito importante, um dos principais motivos pelo grande sucesso dos gerenciadores de banco de dados (SGBD/DBMS).



O gerenciamento das transações são feitos pelo PostgreSQL, no entanto, existem comandos, instruções e configurações que definem o seu comportamento. É exatamente isto que veremos neste conjunto de artigos.

Você irá ver como o PostgreSQL permite juntar diversas alterações a serem efetuadas na base de dados como uma unidade única de trabalho, ou seja, quando você tem um conjunto de mudanças que devem ser todas executadas ou, se ocorrer algum problema, não será feito nada.

O QUE SÃO AS TRANSAÇÕES

Antes de mais nada precisamos saber como os dados são atualizados no PostgreSQL. Geralmente quando mostramos exemplos de alterações de dados em uma base, usamos uma única declaração para que esta alteração seja feita. Porém, no mundo real, encontramos diversas situações onde precisamos fazer diversas mudanças às quais seriam impossíveis de serem efetuadas em uma única declaração. Você ainda precisa que todas estas alterações obtenham sucesso, ou nenhuma delas deve ser efetuada, caso surja algum problema em qualquer ponto deste grupo de mudanças.




Um exemplo clássico é a transferência de dinheiro entre duas contas bancárias, podendo ser representadas até mesmo em tabelas distintas, e precisamos que uma conta seja debitada e a outra, creditada. Se debitarmos um valor de uma conta e ocorrer algum problema ao tentarmos creditar este valor, qualquer que seja o motivo, precisamos retornar o valor à primeira conta, ou sendo sendo mais coerente, este valor nunca chegou a ser debitado da primeira conta. Nenhum banco poderia manter-se no negócio caso eventualmente "perdesse" o dinheiro durante uma transação financeira entre duas contas (ou qualquer outra, logicamente).



Em bancos de dados relacionais baseados no ANSI SQL, como o PostgreSQL, isto é possível graças ao recurso de nome **TRANSAÇÃO**.

Uma transação é uma unidade lógica de trabalho que não deve ser subdividida.

Mas o que sabemos sobre unidade lógica de trabalho? Isto é simplesmente um conjunto de mudanças a serem efetuadas na base de dados onde todas devem ter sucesso, ou todas devem falhar. Exatamente como a transferência de dinheiro entre contas bancárias como mencionado acima. No PostgreSQL, estas mudanças são controladas por três instruções:

	BEGIN WORK declara o início de uma transação;
	COMMIT WORK indica que todos os elementos da transação foram executados com sucesso e podem agora serem persistidos e acessados por todas as demais transações concorrentes ou subsequentes;
	ROLLBACK WORK indica que a transação será abandonada e todas as mudanças feitas nos dados pelas instruções em SQL serão canceladas. O banco de dados se apresentará aos seus usuários como se nenhuma mudança tivesse ocorrido desde a instrução BEGIN WORK;

OBS: Tanto **COMMIT WORK** e **ROLLBACK WORK** podem omitir a palavra **WORK**, mas, em **BEGIN WORK** é obrigatório.

A padronização ANSI / ISO SQL não define a instrução BEGIN WORK, define que as transações iniciam-se automaticamente, mas esta é uma extensão muito comum presente - e requerida - em muitos bancos de dados relacionais.

Um segundo aspecto sobre as transações é que qualquer transação no banco de dados é isolada de outras transações que estão ocorrendo ao mesmo tempo. De uma forma ideal, cada transação se comportaria como se tivesse acesso exclusivo ao banco de dados. Infelizmente, como veremos nos próximos artigos, existem níveis de isolamento que, na prática, para atingir melhores performances algo será comprometido.

ACID : ÁCIDO, AGRE, AZEDO,... ?!!!

Não.. não tem nada haver com culinária...

ACID é um mnemônico (uma sigla) que descreve as propriedades que uma transação deve possuir.

Atomic Atômica	Uma transação, mesmo sendo um conjunto de ações, deve ser executada como uma unidade única. Uma transação deve ser executada exatamente uma única vez, sem nenhuma dependência. Em nosso exemplo bancário, a movimentação financeira deve ser atômica. O débito em uma conta e o crédito em outra devem ambos acontecer como sendo uma única ação, mesmo se diversas instruções consecutivas em SQL fossem requeridas.
Consistent Consistente	Ao término de uma transação, o sistema deve ser deixado em um estado consistente, respeitando as integridades de dados e relacional da base sendo manipulada. No exemplo da movimentação bancária, ao final da transação, ambas as contas devem ter sido atualizadas com os montantes corretos.
Isolated Isolada	Isto significa que cada transação, não importando quantas transações estão sendo executadas neste momento no banco de dados, deve parecer ser independente de todas as outras transações. Imagine caixas eletrônicos separados, onde, diferentes pessoas querem executar a mesma operação bancária para a mesma conta. Transações processando duas ações concorrentes devem comportar-se como se cada uma delas estivessem operando com acesso exclusivo à base de dados. Na prática, nós sabemos que as coisas não são tão simples assim. Nós trataremos deste tópico nos próximos artigos.
Durable Durável	Uma vez que a transação seja completada, ela deve permanecer completada. Uma vez que o dinheiro tenha sido transferido com sucesso entre as contas, ele deve permanecer transferido, mesmo que a máquina que está rodando o gerenciador de banco de dados pare por falta de energia elétrica. No PostgreSQL, assim como na maioria dos bancos de dados relacionais, isto é conseguido através de um arquivo que descreve as transações (<i>transaction logfile</i>). A maneira este arquivo funciona é bastante simples. Assim que uma transação é executada, as ações não são somente executadas na base de dados, mas também gravadas neste arquivo. Assim que a transação é completada, uma marca é gravada para dizer que a transação foi finalizada, e os dados gravados no arquivo <i>logfile</i> são forçados a permanecerem armazenados, então isto torna o banco de dados seguro mesmo que ele "caia". A durabilidade da transação acontece sem a intervenção do usuário, ou seja, é uma operação executada pelo próprio banco de dados.

Transações com o PostgreSQL - Parte II

Esta é a segunda parte da série de artigos que está dividida em três momentos.

Vimos no artigo anterior o que são as transações, quais instruções são necessárias, e quais as propriedades que uma transação deve possuir.

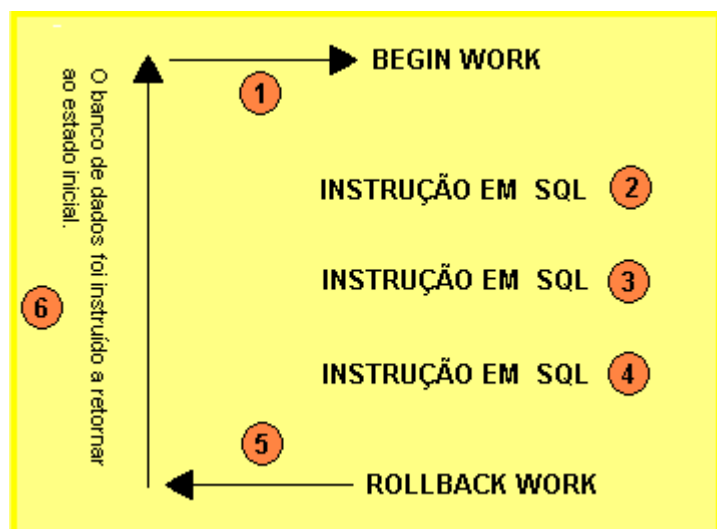


TRANSAÇÕES DE UM ÚNICO USUÁRIO

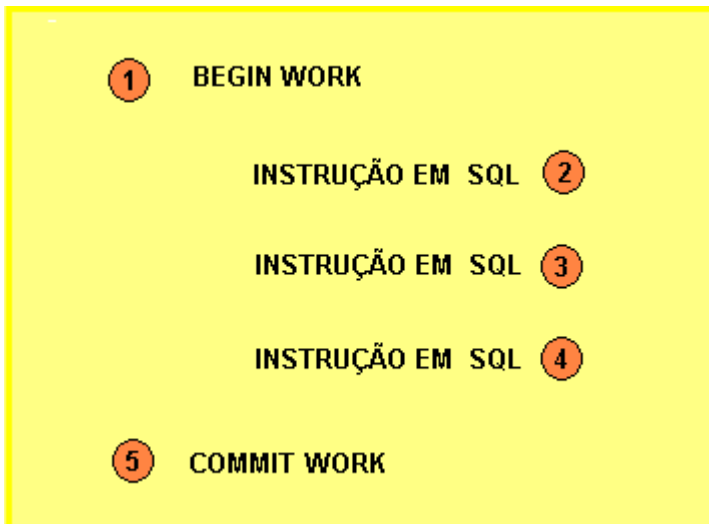
Antes de nós olharmos aspectos mais complexos das transações e como se comportam dentro da concorrência dos usuários do banco de dados, nós precisamos saber primeiro como se comporta com um único usuário.

Mesmo dessa maneira simplista de trabalhar, há vantagens reais em usar o controle de transações. O maior benefício do controle da transação é que permite que sejam executadas diversas instruções SQL, e então, ao final do processo, permitir que você possa desfazer o trabalho que já fez, caso queira isto. Usando uma transação, a aplicação não necessita se preocupar em armazenar quais alterações foram sendo feitas ao banco de dados para desfazê-las então, se necessário. Você simplesmente pede ao sistema gerenciador de banco de dados para desfazer todo o processo de uma só vez.

A seqüência se apresenta da seguinte maneira:



Se você decidir que todas as alterações no banco de dados são válidas até o passo 5, e mesmo assim, você deseja aplicá-las ao banco de dados para então se tornarem permanentes, então, a única coisa que você tem que fazer é substituir a instrução **ROLLBACK WORK** pela instrução **COMMIT WORK**:



Após o passo 5, as mudanças ocorridas no banco de dados são COMETIDAS (escritas, confirmadas), e podem ser consideradas permanentes, dessa maneira não serão perdidas caso haja falta de energia elétrica, problemas no disco ou erros na aplicação.

REPETINDO

Uma transação é uma unidade lógica de trabalho que não deve ser subdividida.

Abaixo está um exemplo bem simples, onde é alterado uma única linha em uma tabela de cadastro chamada `tbl_usuario`, trocando o conteúdo da coluna nome de 'Juliano' por 'Mauricio', e então usando a instrução `ROLLBACK WORK` para cancelar a alteração:

```
USANDO o ROLLBACK WORK
postgres=# BEGIN WORK;
BEGIN
postgres=# SELECT nome FROM tbl_usuario WHERE usuid=1;
nome
-----
Juliano
(1 row)

postgres=# UPDATE tbl_usuario
postgres=# SET nome='Mauricio'
postgres=# WHERE usuid=1;
UPDATE 1
postgres=# SELECT nome FROM tbl_usuario WHERE usuid=1;
nome
-----
Mauricio
(1 row)

postgres=# ROLLBACK WORK;
ROLLBACK
postgres=# SELECT nome FROM tbl_usuario WHERE usuid=1;
nome
-----
Juliano
(1 row)
postgres=#
```



```
USANDO O COMMIT WORK
postgres=# BEGIN WORK;
BEGIN
postgres=# SELECT nome FROM tbl_usuario WHERE usuid=1;
nome
-----
Juliano
(1 row)

postgres=# UPDATE tbl_usuario
postgres=# SET nome='Mauricio'
postgres=# WHERE usuid=1;
UPDATE 1
postgres=# SELECT nome FROM tbl_usuario WHERE usuid=1;
nome
-----
Mauricio
(1 row)

postgres=# COMMIT WORK;
ROLLBACK
postgres=# SELECT nome FROM tbl_usuario WHERE usuid=1;
nome
-----
Mauricio
(1 row)

postgres=#
```

LEMBRE-SE: Tanto **COMMIT WORK** e **ROLLBACK WORK** podem omitir a palavra **WORK**, mas, em **BEGIN WORK** é obrigatório.

LIMITAÇÕES ...mas não por muito tempo.

Há alguns pontos a serem considerados quando estamos falando de transações:

Primeiramente, o PostgreSQL ainda não trabalha com transações aninhadas, este é um recurso muito esperado que está sendo inserido na versão 7.4 (que já está em fase de beta testes). Hoje no PostgreSQL, se você tentar executar uma instrução **BEGIN WORK** enquanto ainda se encontra em uma transação, o PostgreSQL irá ignorar este último comando e emitirá a seguinte mensagem:

WARNING: BEGIN: already a transaction in progress

Assim que o PostgreSQL puder trabalhar com transações aninhadas, o conceito de SAVE POINTS estará disponível e será possível marcar mais de um ponto dentro de uma transação principal e, retornar (**ROLLBACK WORK**) até um determinado ponto (SAVE POINT) ao invés de retornar toda a transação.

Justamente por não trabalhar com transações aninhadas, é de boa prática manter as transações no menor tamanho possível (e viável). Tanto o PostgreSQL quanto qualquer outro banco de dados relacional, tem que executar muito trabalho para assegurar que transações de diferentes usuários devem ser executadas separadamente. Uma consequência disto é que partes do banco de dados envolvidas numa transação, freqüentemente precisam se tornar parcialmente inacessível (trancadas), para garantir que as transações sejam mantidas separadas.

Embora o PostgreSQL tranque o banco de dados automaticamente nestas ocasiões, uma transação de longa duração geralmente impossibilita que outros usuários acessem os dados envolvidos nesta transação, até que a mesma seja completada ou cancelada. Imagine que uma aplicação iniciou uma transação assim que a pessoa chegou cedo ao escritório, sentou para trabalhar, e deixou uma transação rodando o dia inteiro enquanto executava diversas alterações no banco de dados. Supondo que esta pessoa "COMITOU" a transação somente no final do expediente, a performance do banco de dados, e a capacidade de outros usuários acessarem os dados desta enorme transação foram severamente impactados.

PODE PARECER ÓBVIO: ...mas não deixe uma transação aberta (iniciada) e saia pra tomar um café, ou mesmo falar com alguém, isto pode comprometer seriamente o andamento dos trabalhos na empresa que necessitem do banco de dados para executá-los.

Uma instrução **COMMIT WORK** geralmente é executada muito rápido, uma vez que - geralmente - tem muito pouco trabalho a ser executado. Retornar transações no entanto, normalmente envolve muito mais trabalho para o banco de dados executar, pois, além dos trabalhos já executados até então dentro da transação, deve desfazer os mesmos. Ou seja, se você inicia uma transação que demora 3 minutos para ser executada e, ao final você decide executar um **ROLLBACK WORK** para retornar à situação inicial, então, não espere que seja executado instantaneamente. Isto poderá demorar facilmente mais do que 3 minutos para restaurar todas as alterações já executadas pela transação.

Transações com o PostgreSQL - Parte III

Esta é a terceira, porém, ainda não, a última parte deste artigo originalmente dividido em três partes. Haverá uma quarta parte, pois, após os vários e-mail que recebi com relação à este artigo, decidi refiná-lo um pouco mais, o que fez com que eu acrescentasse uma parte a mais.



Nos artigos anteriores, vimos como as transações precisam trabalhar em ambientes com múltiplos usuários de forma concorrente e como cada transação deve ser isolada uma das outras, nos importando então somente, verificar como se comporta uma única transação. Agora, iremos retornar às regras **ACID** e olhar mais de perto o que significa o **I** de **ACID**, Isolated (Isolado).

NÍVEIS DE ISOLAMENTO ANSI

Como já foi mencionado, um dos aspectos mais difíceis dos bancos de dados relacionais é o isolamento entre diferentes usuários em relação às atualizações feitas na base. Claro que atingir o isolamento não é difícil. Simplesmente, permita somente uma única conexão para o banco de dados, com somente uma transação sendo processada por vez. Isto garantirá à você um isolamento completo entre as diferentes transações. A dificuldade aparece quando procuramos atingir um isolamento mais prático sem prejudicar significativamente a performance, prevenindo o acesso multi-usuário ao banco de dados.

O verdadeiro isolamento é extremamente difícil de ser atingido sem uma alta degradação de performance como consequência, o padrão ANSI / ISO SQL define diferentes níveis de isolamento que os bancos de dados podem implementar. Frequentemente, um banco de dados relacional irá

implementar pelo menos um destes níveis como default (padrão) e, normalmente, permitem aos usuários especificar pelo menos um outro nível de isolamento para ser utilizado.

Antes que possamos entender os padrões de níveis de isolamento, nós precisamos nos familiarizar com algumas outras terminologias. Embora o comportamento padrão do PostgreSQL se mostre suficiente na maioria dos casos, há circunstâncias onde seja útil entendê-lo em detalhes.

FENÔMENOS INDESEJÁVEIS

O padrão ANSI / ISO SQL define os níveis de isolamento em termos de fenômenos indesejáveis que podem acontecer em bancos de dados multi-usuários quando interagem com as transações.

Dirty Read (leitura suja)

O *Dirty Read* ocorre quando algumas instruções em SQL em uma transação lêem dados que foram alterados por uma outra transação, porém, a transação que alterou os dados ainda não completou (**COMMIT**) seu bloco de instruções.

Como já mostrado antes (de maneira insistente), uma transação é um bloco ou unidade lógica de trabalho que deve ser atômica. Mesmo que todos os elementos de uma transação ocorram ou nenhuma delas ocorra. Até que uma transação tenha sido completada (**COMMITed**), sempre haverá a possibilidade de que ela falhe, ou seja abandonada através de uma instrução **ROLLBACK WORK**. Por isso, nenhum outro usuário do banco de dados deveria ver estas mudanças antes da transação ser completada.

Para ilustrar isso, considerando que diferentes transações podem ser vistas através da coluna **nome** da tabela **tbl_usuario** que possui o valor 1 na coluna **usuid**. Assumindo que o *Dirty Read* está permitido, o que o PostgreSQL nunca permitirá, então só veremos o comportamento da transação como um exemplo teórico:

UM EXEMPLO DE <i>DIRT READ</i>			
Transação 1	Dados vistos pela Transação 1	O que o <i>Dirty Read</i> em outra transação veria	O que outras transações veriam se o <i>Dirty Read</i> não ocorresse
postgres=# BEGIN WORK; BEGIN	Juliano	Juliano	Juliano
postgres=# UPDATE tbl_usuario SET nome='Marcio' WHERE usuid=1; UPDATE 1	Marcio	Marcio	Juliano
postgres=# COMMIT WORK; COMMIT	Marcio	Marcio	Marcio
postgres=# BEGIN WORK; BEGIN	Marcio	Marcio	Marcio
postgres=# UPDATE tbl_usuario SET nome='Andrea' WHERE usuid=1; UPDATE 1	Andrea	Andrea	Marcio
postgres=# ROLLBACK WORK; ROLLBACK	Marcio	Marcio	Marcio

Note que uma leitura suja (*Dirty Read*) permite que outras transações possam ver os dados que ainda não foram completados (**COMMITed**). Isto significa que eles podem ver mudanças que podem ser descartadas em seguida, por causa de uma instrução **ROLLBACK WORK** por exemplo.

O PostgreSQL NÃO permite dirty read.

Unrepeatable Reads (leituras não podem ser repetidas)

Um *Unrepeatable Read* (leitura que não pode ser repetida) ocorre quando uma transação lê um conjunto de dados, depois mais tarde, re-lê os dados, e descobre que os mesmos mudaram. Isto é muito menos sério do que o *Dirt Read*. Vamos ver como isso se parece:

UM EXEMPLO DE UNREPEATABLE READ			
Transação 1	Dados vistos pela Transação 1	O que o <i>Unrepeatable Read</i> em outra transação veria	O que outras transações veriam se o <i>Unrepeatable Read</i> não ocorresse
postgres=# BEGIN WORK; BEGIN	Marcio	Marcio	Marcio
postgres=# UPDATE tbl_usuario SET nome='Mauricio' WHERE usuid=1; UPDATE 1	Mauricio	Marcio	Marcio
postgres=# COMMIT WORK; COMMIT	Mauricio	Mauricio	Marcio
postgres=# SELECT nome FROM tbl_usuario WHERE usuid=1; (1 row)	Mauricio	Mauricio	Mauricio

Note que o *Unrepeatable Read* significa que a transação pode ver as mudanças completadas (**COMMITadas**) por outras transações, mesmo que embora uma transação de leitura não tenha sido completada. Se as leituras que não podem ser repetidas forem prevenidas, então outras transações não poderão ver as alterações no banco de dados até que eles próprios sejam completados (**COMMIT**).

Por padrão, o PostgreSQL não permite *Unrepeatable Reads*, embora, como veremos adiante, nós poderemos alterar este comportamento padrão.

O PostgreSQL NÃO permite unrepeatable reads.

Phantom Reads (leituras fantasmas)

Este problema ocorre quando uma nova tupla (registro) aparece na tabela, enquanto uma transação diferente está atualizando a tabela, e a nova tupla deveria ter sido atualizada, mas não foi.

Suponha que temos duas transações atualizando a tabela `tbl_usuario`. A primeira está adicionando uma vírgula ao fim do nome, e a segunda adicionando um novo nome:

UM EXEMPLO DE PHANTOM READ	
Transação 1	Transação 2
postgres=# BEGIN WORK; BEGIN	postgres=# BEGIN WORK; BEGIN
postgres=# UPDATE tbl_usuario SET nome = nome ','; UPDATE 1	postgres=# INSERT INTO tbl_usuario (nome) VALUES ('Adriana'); COMMIT
postgres=# COMMIT WORK; COMMIT	postgres=# COMMIT WORK; COMMIT

O que a coluna nome de `tbl_usuario` deveria conter? O **INSERT** começou antes da atualização ser confirmada, então, esperamos que seja razoável que o nome inserido esteja com a vírgula adicionada. Se um *Phantom Read* (leitura fantasma) ocorre, então a nova tupla que aparece depois da Transação 1 determina quais tuplas serão atualizadas, e a vírgula do novo item não é

concatenada.

Leituras fantasmas são muito raras, e quase impossíveis de demonstrar, no entanto, geralmente você não precisa se preocupar com isto, embora por padrão o PostgreSQL permitirá algum tipo de *Phantom Reads*.

Lost Updates (atualizações perdidas)

Atualizações perdidas é um pouco diferente dos três casos anteriores, que são geralmente problemas de nível de aplicação, não estando relacionado com a maneira como o banco de dados trabalha. Um *Lost Update* por outro lado, ocorre quando duas mudanças diferentes são escritas no banco de dados, e a segunda mudança causa a perda da primeira.

Suponha dois usuários que estão utilizando uma aplicação baseada em console, que está atualizando o cadastro de usuários:

UM EXEMPLO DE LOST UPDATES			
Usuário 1	Dados vistos pelo Usuário 2	Usuário 1	Dados vistos pelo Usuário 2
Tentando alterar o nome 'Marcio' para 'Maurice'		Tentando alterar o sobrenome 'Costa' para 'Fischer'	
postgres=# BEGIN WORK; BEGIN		postgres=# BEGIN WORK; BEGIN	
postgres=# SELECT nome, sobrenome FROM tbl_usuario WHERE usuid=1;	Marcio, Costa	postgres=# SELECT sobrenome, nome FROM tbl_usuario WHERE usuid=1;	Costa, Marcio
postgres=# UPDATE tbl_usuario SET nome='Maurice', sobrenome = 'Costa' WHERE usuid=1; UPDATE 1	Maurice, Costa		
postgres=# COMMIT WORK; COMMIT			Maurice, Costa
		postgres=# UPDATE tbl_usuario SET nome='Marcio', sobrenome='Fischer' WHERE usuid=1; UPDATE 1	
	Maurice, Costa	postgres=# COMMIT WORK; COMMIT	Marcio, Fischer
	arcio, Fischer		arcio, Fischer

O **nome** alterado pelo Usuário 1 é perdido, não por haver algum erro no banco de dados, mas porque o Usuário 2 leu o **nome**, então o manteve por alguns momentos, e gravou-o novamente no banco de dados, destruindo a mudança que o Usuário 1 tinha feito. O banco de dados está quase isolando corretamente os dois conjuntos de dados, mas a aplicação ainda está perdendo dados.

Há diversas maneiras de contornar este problema, e decidir qual a maneira mais apropriada irá depender de cada aplicação. Como um primeiro passo, as aplicações devem se atentar a manter as transações o mais curtas possíveis, nunca segurando-as em execução por muito mais tempo além do estritamente necessário. Como um segundo passo, as aplicações deveriam escrever de volta os dados que ele têm alterado.. Estes dois passos irá preveni-los de muitas ocorrências de perda de atualizações, incluindo o erro mostrado acima.

No próximo artigo: *Set Transaction Isolation Level, Auto Commit, Locking, Deadlocks, etc.*

Transações com o PostgreSQL - Parte IV

Agora sim! A quarta e última parte deste artigo. Até aqui nós vimos como as transações são úteis mesmo em banco de dados de um só usuário, permitindo-nos agrupar instruções SQL em uma unidade atômica, à qual pode tanto ser completada, quanto abandonada.



Também vimos como as transações trabalham em ambientes multi-usuários. Vimos os significados das regras de cada letra do acrônimo **ACID** e o que significam em termos de banco de dados. Também vimos termos padrões **ANSI** de "fenômenos indesejáveis".

NÍVEIS DE ISOLAMENTO ANSI / ISO

Usando a nossa mais recente terminologia, nós estamos agora na posição de entender o caminho definido pelo ANSI / ISO para diferentes níveis de isolamento que um banco de dados pode usar. Cada nível ANSI / ISO é uma combinação dos primeiros três tipos de comportamentos indesejáveis mostrados logo abaixo:

Definição do Nível de Isolamento ANSI / ISO	Dirty Read	Unrepeatable Read	Phantom
Read Uncommitted	Possible	Possible	Possible
Read Committed	Not Possible	Possible	Possible
Repeatable Read	Not Possible	Not Possible	Possible
Serializable	Not Possible	Not Possible	Not Possible

Você pode ver que, assim como o nível de isolamento se move de "Read Uncommitted" para "Read Committed" e "Repeatable Read" para o último "Serializable", os tipo de comportamentos indesejáveis que podem acontecer em degraus de redução.

O nível de isolamento está configurado para usar o comando **SET TRANSACTION ISOLATION LEVEL**, usando a seguinte sintaxe:

```
SET TRANSACTION ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE }
```

Por padrão, o modo será configurado para "Read Committed".

Note que o PostgreSQL, no momento em que foi escrito, não pode prover o nível intermediário "Repeatable Read", nem o nível de entrada "Read Uncommitted". Geralmente, "Read Uncommitted" é um comportamento pobre que poucos bancos de dados o oferece como opção, e seria rara uma aplicação que fosse forte (ou temerária) o bastante para escolher usá-la.

Similarmente, o nível intermediário "Repeatable Read", provê somente proteção adicional contra "Phantom Reads", o qual nós dissemos ser extremamente raros, então a falta deste nível não oferece

maiores consequências. É muito comum os bancos de dados oferecerem menos opções do que o conjunto completo de possibilidades, e prover "Read Committed" e "Serializable" como solução.

MODO ENCADEADO (Auto Commit) E NÃO ENCADEADO

Através deste tópico, nós teremos que usar explicitamente **BEGIN WORK** e **COMMIT** (ou **ROLLBACK**) **WORK** para delimitar nossas transações. É muito comum no entanto, nós executarmos diretamente diversas alterações em nossas bases de dados, sem uma instrução **BEGIN WORK** ser vista.

Por padrão o PostgreSQL opera em modo Auto Commit, algumas vezes referenciado como modo encadeado ou modo de transações implícitas, onde cada instrução SQL que pode modificar um dado age como se ela fosse uma transação completa com seus próprios direitos. Isto é realmente ótimo quando se trata de interação através de linhas de comando, mas não tão bom quando se trata de sistemas reais, onde temos que ter acesso à transações com instruções explícitas **COMMIT** ou **ROLLBACK**.

Em outros servidores SQL que implementam diferentes modos, você normalmente é obrigado a digitar um comando explícito para alterar o modo. No PostgreSQL, tudo o que você precisa fazer é digitar a instrução **BEGIN COMMIT**, que o PostgreSQL automaticamente entra no modo encadeado até que uma instrução **COMMIT** (ou **ROLLBACK**) **WORK** seja digitada.

O padrão SQL não define uma instrução **BEGIN WORK**, porém, a maneira do PostgreSQL tratar transações, com um **BEGIN WORK** explícito, é muito comum.

BLOQUEIO (Locking)

Muitos bancos de dados implementam transações, em particular isolando diferentes transações entre alguns usuários, usando bloqueios para restringir o acesso aos dados por outros usuários. Simplificando, existem dois tipos de bloqueios:

- Bloqueio compartilhado (share), que permite a outros usuários ler, mas não atualizar os dados;
- Bloqueio exclusivo (exclusive), que previne que outras transações até mesmo leiam os dados;

Por exemplo, o servidor irá bloquear os registros que estão sendo alterados por uma transação, até que a transação esteja completa, quando os bloqueios são automaticamente liberados. Tudo isto é feito automaticamente, geralmente sem que os usuários saibam ou se preocupem com o que está acontecendo.

A atual mecânica e estratégia necessária para bloqueios é altamente complexa, possuindo diversos tipos diferentes de bloqueios para serem usados conforme as circunstâncias. A documentação do PostgreSQL descrevem sete tipos diferentes de permutação de bloqueios. O PostgreSQL também implementa um mecanismo específico para isolar transações utilizando um modelo de múltiplas versões, que reduz conflitos entre bloqueios, aumentando significativamente a performance comparado com outros esquemas.

Felizmente, os usuários de bancos de dados geralmente precisam se preocupar sobre bloqueios somente em duas circunstâncias, evitando deadlocks (o "abraço mortal", e se recuperando dele) e, num bloqueio explícito através de uma aplicação.

O ABRAÇO MORTAL (Deadlocks)

O que acontece quando duas diferentes aplicações tentam e alteram os mesmos dados na mesma hora? É fácil de ver, inicie duas sessões do psql, e preste atenção em alterar o mesmo registro em

ambas sessões:

DEADLOCK	
Sessão 1	Sessão 2
UPDATE row 14;	UPDATE row 15;
UPDATE row 15;	UPDATE row 14;

Neste ponto, ambas as sessões são bloqueadas, pois, cada uma delas está aguardando a outra para serem liberadas.

Inicie duas sessões do psql, e tente a seguinte sequência de comandos:

UM EXEMPLO DE DEADLOCK	
Sessão 1	Sessão 2
postgres=# BEGIN WORK; BEGIN	postgres=# BEGIN WORK; BEGIN
postgres=# UPDATE tbl_usuario SET nome='Mauricio' WHERE usuid=2;	postgres=# UPDATE tbl_usuario SET nome='Juliano' WHERE usuid=1;
postgres=# UPDATE tbl_usuario SET nome='Julio' WHERE usuid=1;	postgres=# UPDATE tbl_usuario SET nome='Marcio' WHERE usuid=2;

Você verá que ambas as sessões serão bloqueadas, e então, depois de uma curta pausa, ocorrerá o seguinte erro em uma das sessões:

```
ERROR:Deadlock detected.
```

```
See the lock(1) manual page for a possible cause.
```

A outra sessão irá continuar. A sessão que resultou na mensagem de deadlock é desfeita (rolled back), e as mudanças efetuadas até então, perdidas. A sessão que continuou poderá executar o restante das instruções que porventura ainda tenha até que a instrução COMMIT WORK seja executada, e torne as alterações do banco de dados permanente.

Não há uma maneira de saber qual sessão o PostgreSQL irá interromper e qual continuará. Ele irá testar e escolher a que considerar de menor custo de processamento, mas, isto está longe de ser uma ciência perfeita.

BLOQUEIO EXPLÍCITO

Ocasionalmente, você pode achar que o bloqueio automático que o PostgreSQL provê não é suficiente para suas necessidades, e se encontrará em uma situação onde precisa bloquear explicitamente tanto alguns registros como toda uma tabela. Você deve evitar bloqueios explícitos tanto o quanto puder. O padrão SQL nem sequer define um modo de bloquear toda uma tabela, isto é uma extensão do PostgreSQL.

É possível somente bloquear registros ou tabelas de dentro de uma transação. Assim que a transação termina, todos os bloqueios impostos pela transação são automaticamente liberados. Também não há uma maneira de liberar um bloqueio explicitamente durante uma transação, por uma simples razão que, liberando o bloqueio em um registro que está sendo alterado durante uma transação, deve possibilitar que outra aplicação o altere também, o que impossibilitaria um **ROLLBACK** de

desfazer a transação em si até a alteração inicial.

Bloqueando Registros

Para bloquear um conjunto de registros, nós simplesmente utilizamos a instrução **SELECT**, adicionando a cláusula **FOR UPDATE**:

```
SELECT * FROM tbl_usuario WHERE nome like 'J%' FOR UPDATE
```

Bloqueando Tabelas

Com o PostgreSQL também é possível bloquear uma tabela, porém, geralmente, isso não é muito recomendado, pois a performance de outras transações serão seriamente afetadas.

A sintaxe é:

```
LOCK [ TABLE ] table  
LOCK [ TABLE ] table IN [ ROW | ACCESS ] {SHARE | EXCLUSIVE} MODE  
LOCK [ TABLE ] table IN SHARE ROW EXCLUSIVE MODE
```

Geralmente, aplicações que necessitam bloquear uma tabela, usam simplesmente:

```
LOCK [ TABLE ] table
```

que é a mesma coisa que:

```
LOCK [ TABLE ] table IN ACCESS EXCLUSIVE MODE
```

Embora transações e bloqueios não seja um assunto - diga-se de passagem - muito interessante, porém, é de extrema importância um entendimento geral sobre os mesmos, ter uma boa noção de como estes recursos irão contribuir no desenvolvimento de aplicações sólidas, além de interagir com o servidor para minimizar implicações de performance e aumentar a segurança nos ambientes atuais.

Links do original:

http://imasters.uol.com.br/artigo/1067/postgresql/transacoes_com_o_postgresql_-_parte_i/
http://imasters.uol.com.br/artigo/1082/postgresql/transacoes_com_o_postgresql_-_parte_ii/
http://imasters.uol.com.br/artigo/1097/postgresql/transacoes_com_o_postgresql_-_parte_iii/
http://imasters.uol.com.br/artigo/1109/postgresql/transacoes_com_o_postgresql_-_parte_iv/

12) Entendendo as constraints e a integridade referencial

Restrições

Os tipos de dado são uma forma de limitar os dados que podem ser armazenados na tabela. Entretanto, para muitos aplicativos a restrição obtida não possui o refinamento necessário. Por exemplo, uma coluna contendo preços de produtos provavelmente só pode aceitar valores positivos, mas não existe nenhum tipo de dado que aceite apenas números positivos. Um outro problema é que pode ser necessário restringir os dados de uma coluna com relação a outras colunas ou linhas. Por exemplo, em uma tabela contendo informações sobre produtos deve haver apenas uma linha para cada código de produto.

Para esta finalidade, a linguagem SQL permite definir restrições em colunas e tabelas. As restrições permitem o nível de controle sobre os dados da tabela que for desejado. Se o usuário tentar armazenar dados em uma coluna da tabela violando a restrição, ocorrerá um erro. Isto se aplica até quando o erro é originado pela definição do valor padrão.

Restrições de verificação (check)

Uma restrição de verificação é o tipo mais genérico de restrição. Permite especificar que os valores de uma determinada coluna devem estar de acordo com uma expressão booleana (valor-verdade [\[1\]](#)). Por exemplo, para permitir apenas preços com valores positivos utiliza-se:

```
CREATE TABLE produtos (  
    cod_prod integer,  
    nome text,  
    preco numeric CHECK (preco > 0)  
);
```

Como pode ser observado, a definição da restrição vem após o tipo de dado, assim como a definição do valor padrão. O valor padrão e a restrição podem estar em qualquer ordem. A restrição de verificação é formada pela palavra chave CHECK seguida por uma expressão entre parênteses. A expressão da restrição de verificação deve envolver a coluna sendo restringida, senão não fará muito sentido.

Também pode ser atribuído um nome individual para a restrição. Isto torna mais clara a mensagem de erro, e permite fazer referência à restrição quando se desejar alterá-la. A sintaxe é:

```
CREATE TABLE produtos (  
    cod_prod integer,  
    nome text,  
    preco numeric CONSTRAINT chk_preco_positivo CHECK (preco > 0)  
);
```

Portanto, para especificar o nome da restrição deve ser utilizada a palavra chave CONSTRAINT, seguida por um identificador, seguido por sua vez pela definição da restrição (Se não for escolhido o nome da restrição desta maneira, o sistema escolherá um nome para a restrição).

Uma restrição de verificação também pode referenciar várias colunas. Supondo que serão armazenados o preço normal e o preço com desconto, e que se deseje garantir que o preço com desconto seja menor que o preço normal:

```
CREATE TABLE produtos (  
    cod_prod          integer,  
    nome              text,  
    preco              numeric CHECK (preco > 0),  
    preco_com_desconto numeric CHECK (preco_com_desconto > 0),  
    CHECK (preco > preco_com_desconto)  
);
```

As duas primeiras formas de restrição já devem ser familiares. A terceira utiliza uma nova sintaxe, e não está anexada a uma coluna em particular. Em vez disso, aparece como um item à parte na lista de colunas separadas por vírgula. As definições das colunas e as definições destas restrições podem estar em qualquer ordem.

Dizemos que as duas primeiras restrições são restrições de coluna, enquanto a terceira é uma restrição de tabela, porque está escrita separado das definições de colunas. As restrições de coluna também podem ser escritas como restrições de tabela, enquanto o contrário nem sempre é possível, porque supostamente a restrição de coluna somente faz referência à coluna em que está anexada (O PostgreSQL não impõe esta regra, mas deve-se segui-la se for desejado que a definição da tabela sirva para outros sistemas de banco de dados). O exemplo acima também pode ser escrito do seguinte modo:

```
CREATE TABLE produtos (  
    cod_prod          integer,  
    nome              text,  
    preco              numeric,  
    CHECK (preco > 0),  
    preco_com_desconto numeric,  
    CHECK (preco_com_desconto > 0),  
    CHECK (preco > preco_com_desconto)  
);
```

ou ainda

```
CREATE TABLE produtos (  
    cod_prod          integer,  
    nome              text,  
    preco              numeric CHECK (preco > 0),  
    preco_com_desconto numeric,  
    CHECK (preco_com_desconto > 0 AND preco > preco_com_desconto)  
);
```

É uma questão de gosto.

Podem ser atribuídos nomes para as restrições de tabela da mesma maneira que para as restrições de coluna:

```
CREATE TABLE produtos (  
    cod_prod          integer,  
    nome              text,  
    preco              numeric,  
    CHECK (preco > 0),  
    preco_com_desconto numeric,  
    CHECK (preco_com_desconto > 0),  
    CONSTRAINT chk_desconto_valido CHECK (preco > preco_com_desconto)  
);
```

Deve ser observado que a restrição de verificação está satisfeita se o resultado da expressão de

verificação for verdade ou o valor nulo. Como a maioria das expressões retorna o valor nulo quando um dos operandos é nulo, estas expressões não impedem a presença de valores nulos nas colunas com restrição. Para garantir que a coluna não contém o valor nulo, deve ser utilizada a restrição de não nulo descrita a seguir.

Restrições de não-nulo

Uma restrição de não-nulo simplesmente especifica que uma coluna não pode assumir o valor nulo. Um exemplo da sintaxe:

```
CREATE TABLE produtos (  
    cod_prod    integer NOT NULL,  
    nome        text     NOT NULL,  
    preco       numeric  
);
```

A restrição de não-nulo é sempre escrita como restrição de coluna. A restrição de não-nulo é funcionalmente equivalente a criar uma restrição de verificação CHECK (nome_da_coluna IS NOT NULL), mas no PostgreSQL a criação de uma restrição de não-nulo explícita é mais eficiente. A desvantagem é que não pode ser dado um nome explícito para uma restrição de não nulo criada deste modo.

Obviamente, uma coluna pode possuir mais de uma restrição, bastando apenas escrever uma restrição em seguida da outra:

```
CREATE TABLE produtos (  
    cod_prod    integer NOT NULL,  
    nome        text     NOT NULL,  
    preco       numeric NOT NULL CHECK (preco > 0)  
);
```

A ordem das restrições não importa, porque não determina, necessariamente, a ordem de verificação das restrições.

A restrição NOT NULL possui uma inversa: a restrição NULL. Isto não significa que a coluna deva ser nula, o que com certeza não tem utilidade. Em vez disto é simplesmente definido o comportamento padrão dizendo que a coluna pode ser nula. A restrição NULL não é definida no padrão SQL, não devendo ser utilizada em aplicativos portáteis (somente foi adicionada ao PostgreSQL para torná-lo compatível com outros sistemas de banco de dados). Porém, alguns usuários gostam porque torna fácil inverter a restrição no script de comandos. Por exemplo, é possível começar com

```
CREATE TABLE produtos (  
    cod_prod    integer NULL,  
    nome        text     NULL,  
    preco       numeric NULL  
);
```

e depois colocar a palavra chave NOT onde se desejar.

Dica: Na maioria dos projetos de banco de dados, a maioria das colunas deve ser especificada como não-nula.

Restrições de unicidade

A restrição de unicidade garante que os dados contidos na coluna, ou no grupo de colunas, é único em relação a todas as outras linhas da tabela. A sintaxe é

```
CREATE TABLE produtos (  
    cod_prod    integer UNIQUE,  
    nome        text,  
    preco       numeric  
);
```

quando escrita como restrição de coluna, e

```
CREATE TABLE produtos (  
    cod_prod    integer,  
    nome        text,  
    preco       numeric,  
    UNIQUE (cod_prod)  
);
```

quando escrita como restrição de tabela.

Se uma restrição de unicidade fizer referência a um grupo de colunas, as colunas deverão ser listadas separadas por vírgula:

```
CREATE TABLE exemplo (  
    a integer,  
    b integer,  
    c integer,  
    UNIQUE (a, c)  
);
```

Isto especifica que a combinação dos valores das colunas indicadas deve ser único para toda a tabela, embora não seja necessário que cada uma das colunas seja única (o que geralmente não é).

Também é possível atribuir nomes às restrições de unicidade:

```
CREATE TABLE produtos (  
    cod_prod    integer CONSTRAINT unq_cod_prod UNIQUE,  
    nome        text,  
    preco       numeric  
);
```

De um modo geral, uma restrição de unicidade é violada quando existem duas ou mais linhas na tabela onde os valores de todas as colunas incluídas na restrição são iguais. Entretanto, os valores nulos não são considerados iguais nesta comparação. Isto significa que, mesmo na presença da restrição de unicidade, é possível armazenar um número ilimitado de linhas que contenham o valor nulo em pelo menos uma das colunas da restrição. Este comportamento está em conformidade com o padrão SQL, mas já ouvimos dizer que outros bancos de dados SQL não seguem esta regra. Portanto, seja cauteloso ao desenvolver aplicativos onde se pretenda haver portabilidade. [\[2\]](#) [\[3\]](#) [\[4\]](#)

Chaves primárias

Tecnicamente a restrição de chave primária é simplesmente a combinação da restrição de unicidade com a restrição de não-nulo. Portanto, as duas definições de tabela abaixo aceitam os mesmos

dados:

```
CREATE TABLE produtos (  
    cod_prod    integer UNIQUE NOT NULL,  
    nome        text,  
    preco       numeric  
);
```

```
CREATE TABLE produtos (  
    cod_prod    integer PRIMARY KEY,  
    nome        text,  
    preco       numeric  
);
```

As chaves primárias também podem restringir mais de uma coluna; a sintaxe é semelhante à da restrição de unicidade:

```
CREATE TABLE exemplo (  
    a integer,  
    b integer,  
    c integer,  
    PRIMARY KEY (a, c)  
);
```

A chave primária indica que a coluna, ou grupo de colunas, pode ser utilizada como identificador único das linhas da tabela (Isto é uma consequência direta da definição da chave primária. Deve ser observado que a restrição de unicidade não fornece, por si só, um identificador único, porque não exclui os valores nulos). A chave primária é útil tanto para fins de documentação quanto para os aplicativos cliente. Por exemplo, um aplicativo contendo uma Interface de Usuário Gráfica (GUI), que permite modificar os valores das linhas, provavelmente necessita conhecer a chave primária da tabela para poder identificar as linhas de forma única.

Uma tabela pode ter no máximo uma chave primária (embora possa ter muitas restrições de unicidade e de não-nulo). A teoria de banco de dados relacional dita que toda tabela deve ter uma chave primária. Esta regra não é imposta pelo PostgreSQL, mas normalmente é melhor segui-la.

Chaves Estrangeiras

A restrição de chave estrangeira especifica que o valor da coluna (ou grupo de colunas) deve corresponder a algum valor existente em uma linha de outra tabela. Diz-se que a chave estrangeira mantém a *integridade referencial* entre duas tabelas relacionadas.

Supondo que já temos a tabela de produtos utilizada diversas vezes anteriormente:

```
CREATE TABLE produtos (  
    cod_prod    integer PRIMARY KEY,  
    nome        text,  
    preco       numeric  
);
```

Agora vamos assumir a existência de uma tabela armazenando os pedidos destes produtos. Desejamos garantir que a tabela de pedidos contenha somente pedidos de produtos que realmente existem. Para isso é definida uma restrição de chave estrangeira na tabela de pedidos fazendo referência à tabela de produtos:

```
CREATE TABLE pedidos (  
    cod_pedido integer PRIMARY KEY,  
    cod_prod integer REFERENCES produtos (cod_prod),  
    quantidade integer  
);
```

Isto torna impossível criar um pedido com `cod_prod` não existente na tabela de produtos.

Nesta situação é dito que a tabela de pedidos é a tabela *que faz referência*, e a tabela de produtos é a tabela *referenciada*. Da mesma forma existem colunas fazendo referência e sendo referenciadas.

O comando acima pode ser abreviado escrevendo

```
CREATE TABLE pedidos (  
    cod_pedido integer PRIMARY KEY,  
    cod_prod integer REFERENCES produtos,  
    quantidade integer  
);
```

porque, na ausência da lista de colunas, a chave primária da tabela referenciada é usada como a coluna referenciada.

A chave estrangeira também pode restringir e referenciar um grupo de colunas. Como usual, é necessário ser escrito na forma de restrição de tabela. Abaixo está mostrado um exemplo artificial da sintaxe:

```
CREATE TABLE t1 (  
    a integer PRIMARY KEY,  
    b integer,  
    c integer,  
    FOREIGN KEY (b, c) REFERENCES outra_tabela (c1, c2)  
);
```

Obviamente, o número e tipo das colunas na restrição devem corresponder ao número e tipo das colunas referenciadas.

Pode ser atribuído um nome à restrição de chave estrangeira da forma habitual.

Uma tabela pode conter mais de uma restrição de chave estrangeira, o que é utilizado para implementar relacionamentos muitos-para-muitos entre tabelas. Digamos que existam as tabelas de produtos e de pedidos, e desejamos permitir que um pedido possa conter vários produtos (o que não é permitido na estrutura anterior). Podemos, então, utilizar a seguinte estrutura de tabela:

```
CREATE TABLE produtos (  
    cod_prod integer PRIMARY KEY,  
    nome text,  
    preco numeric  
);  
  
CREATE TABLE pedidos (  
    cod_pedido integer PRIMARY KEY,  
    endereco_entrega text,  
    ...  
);  
  
CREATE TABLE itens_pedidos (  
    cod_prod integer REFERENCES produtos,  
    cod_pedido integer REFERENCES pedidos,
```

```
quantidade integer,  
PRIMARY KEY (cod_prod, cod_pedido)  
);
```

Deve ser observado, também, que a chave primária está sobreposta às chaves estrangeiras na última tabela.

Sabemos que a chave estrangeira não permite a criação de pedidos não relacionados com algum produto. Porém, o que acontece se um produto for removido após a criação de um pedido fazendo referência a este produto? A linguagem SQL permite tratar esta situação também. Intuitivamente temos algumas opções:

- Não permitir a exclusão de um produto referenciado
- Excluir o pedido também
- Algo mais?

Para ilustrar esta situação, vamos implementar a seguinte política no exemplo de relacionamento muitos-para-muitos acima: Quando se desejar remover um produto referenciado por um pedido (através de itens_pedidos), isto não será permitido. Se um pedido for removido, os itens do pedido também serão removidos.

```
CREATE TABLE produtos (  
    cod_prod integer PRIMARY KEY,  
    nome text,  
    preco numeric  
);  
  
CREATE TABLE pedidos (  
    cod_pedido integer PRIMARY KEY,  
    endereco_entrega text,  
    ...  
);  
  
CREATE TABLE itens_pedidos (  
    cod_prod integer REFERENCES produtos ON DELETE RESTRICT,  
    cod_pedido integer REFERENCES pedidos ON DELETE CASCADE,  
    quantidade integer,  
    PRIMARY KEY (cod_prod, cod_pedido)  
);
```

As duas opções mais comuns são restringir, ou excluir em cascata. RESTRICT não permite excluir a linha referenciada. NO ACTION significa que, se as linhas referenciadas ainda existirem quando a restrição for verificada, será gerado um erro; este é o comportamento padrão se nada for especificado (A diferença essencial entre estas duas opções é que NO ACTION permite postergar a verificação para mais tarde na transação, enquanto RESTRICT não permite). CASCADE especifica que, quando a linha referenciada é excluída, as linhas que fazem referência também devem ser excluídas automaticamente. Existem outras duas opções: SET NULL e SET DEFAULT. Estas opções fazem com que as colunas que fazem referência sejam definidas como nulo ou com o valor padrão, respectivamente, quando a linha referenciada é excluída. Deve ser observado que isto não evita a observância das restrições. Por exemplo, se uma ação especificar SET DEFAULT, mas o valor padrão não satisfizer a chave estrangeira, a operação não será bem-sucedida.

Semelhante a ON DELETE existe também ON UPDATE, chamada quando uma coluna referenciada é alterada (atualizada). As ações possíveis são as mesmas.

Mais informações sobre atualização e exclusão de dados podem ser encontradas no [Capítulo 6](#).

Para terminar, devemos mencionar que a chave estrangeira deve referenciar colunas de uma chave primária ou de uma restrição de unicidade. Se a chave estrangeira fizer referência a uma restrição de unicidade, existem algumas possibilidades adicionais sobre como os valores nulos serão correspondidos. Esta parte está explicada na documentação de referência para [CREATE TABLE](#).

Exemplos do tradutor

Exemplo. Restrição de unicidade com valor nulo em chave única simples

Abaixo são mostrados exemplos de inserção de linhas contendo valor nulo no campo da chave única simples da restrição de unicidade. Deve ser observado que, nestes exemplos, o PostgreSQL e o Oracle consideram os valores nulos diferentes.

PostgreSQL 8.0.0:

```
=> \pset null '(nulo) '
=> CREATE TABLE tbl_unique (c1 int UNIQUE);
=> INSERT INTO tbl_unique VALUES (1);
=> INSERT INTO tbl_unique VALUES (NULL);
=> INSERT INTO tbl_unique VALUES (NULL);
=> INSERT INTO tbl_unique VALUES (2);
=> SELECT * FROM tbl_unique;
```

```
      c1
-----
      1
 (nulo)
 (nulo)
      2
(4 linhas)
```

SQL Server 2000:

```
CREATE TABLE tbl_unique (c1 int UNIQUE)
INSERT INTO tbl_unique VALUES (1)
INSERT INTO tbl_unique VALUES (NULL)
INSERT INTO tbl_unique VALUES (NULL)
Violation of UNIQUE KEY constraint 'UQ__tbl_unique__37A5467C'.
Cannot insert duplicate key in object 'tbl_unique'.
The statement has been terminated.
INSERT INTO tbl_unique VALUES (2)
SELECT * FROM tbl_unique
```

```
c1
-----
NULL
1
2
(3 row(s) affected)
```

Oracle 10g:

```
SQL> SET NULL (nulo)
SQL> CREATE TABLE tbl_unique (c1 int UNIQUE);
SQL> INSERT INTO tbl_unique VALUES (1);
SQL> INSERT INTO tbl_unique VALUES (NULL);
```

```
SQL> INSERT INTO tbl_unique VALUES (NULL);
SQL> INSERT INTO tbl_unique VALUES (2);
SQL> SELECT * FROM tbl_unique;
```

```

      c1
-----
      1
(nulo)
(nulo)
      2

```

Exemplo. Restrição de unicidade com valor nulo em chave única composta

Abaixo são mostrados exemplos de inserção de linhas contendo valores nulos em campos da chave única composta da restrição de unicidade. Deve ser observado que, nestes exemplos, somente o PostgreSQL considera os valores nulos diferentes.

PostgreSQL 8.0.0:

```
=> \pset null '(nulo)'
=> CREATE TABLE tbl_unique (c1 int, c2 int, UNIQUE (c1, c2));
=> INSERT INTO tbl_unique VALUES (1,1);
=> INSERT INTO tbl_unique VALUES (1,NULL);
=> INSERT INTO tbl_unique VALUES (NULL,1);
=> INSERT INTO tbl_unique VALUES (NULL,NULL);
=> INSERT INTO tbl_unique VALUES (1,NULL);
=> SELECT * FROM tbl_unique;
```

```

      c1  |      c2
-----+-----
      1  |      1
      1  | (nulo)
(nulo)  |      1
(nulo)  | (nulo)
      1  | (nulo)
(5 linhas)

```

SQL Server 2000:

```
CREATE TABLE tbl_unique (c1 int, c2 int, UNIQUE (c1, c2))
INSERT INTO tbl_unique VALUES (1,1)
INSERT INTO tbl_unique VALUES (1,NULL)
INSERT INTO tbl_unique VALUES (NULL,1)
INSERT INTO tbl_unique VALUES (NULL,NULL)
INSERT INTO tbl_unique VALUES (1,NULL)
Violation of UNIQUE KEY constraint 'UQ__tbl_unique__33D4B598'.
Cannot insert duplicate key in object 'tbl_unique'.
The statement has been terminated.
SELECT * FROM tbl_unique
```

```

      c1      c2
-----
NULL      NULL
NULL      1
1         NULL
1         1
(4 row(s) affected)

```

Oracle 10g:

```
SQL> SET NULL (nulo)
SQL> CREATE TABLE tbl_unique (c1 int, c2 int, UNIQUE (c1, c2));
SQL> INSERT INTO tbl_unique VALUES (1,1);
SQL> INSERT INTO tbl_unique VALUES (1,NULL);
SQL> INSERT INTO tbl_unique VALUES (NULL,1);
SQL> INSERT INTO tbl_unique VALUES (NULL,NULL);
SQL> INSERT INTO tbl_unique VALUES (1,NULL);
INSERT INTO tbl_unique VALUES (1,NULL)
*
ERROR at line 1:
ORA-00001: unique constraint (SCOTT.SYS_C005273) violated
SQL> SELECT * FROM tbl_unique;
```

C1	C2
1	1
1 (nulo)	
(nulo)	1
(nulo)	(nulo)

Exemplo. Cadeia de caracteres vazia e valor nulo

Abaixo são mostrados exemplos de consulta a uma tabela contendo tanto o valor nulo quanto uma cadeia de caracteres vazia em uma coluna. Deve ser observado que apenas o Oracle 10g não faz distinção entre a cadeia de caracteres vazia e o valor nulo. Foram utilizados os seguintes comandos para criar e inserir dados na tabela em todos os gerenciadores de banco de dados:

```
CREATE TABLE c (c1 varchar(6), c2 varchar(6));
INSERT INTO c VALUES ('x', 'x');
INSERT INTO c VALUES ('VAZIA', '');
INSERT INTO c VALUES ('NULA', null);
```

PostgreSQL 8.0.0:

```
=> \pset null '(nulo)'
=> SELECT * FROM c WHERE c2 IS NULL;
```

c1	c2
NULA	(nulo)

(1 linha)

SQL Server 2000:

```
SELECT * FROM c WHERE c2 IS NULL
```

c1	c2
NULA	NULL

(1 row(s) affected)

Oracle 10g:

```
SQL> SET NULL (nulo)
SQL> SELECT * FROM c WHERE c2 IS NULL;
```

C1	C2
----	----

```

-----
VAZIA   (nulo)
NULA    (nulo)

```

DB2 9.1:

```
db2 > SELECT * FROM c WHERE c2 IS NULL;
```

```

C1      C2
-----
NULA    -

```

Exemplo. Coluna sem restrição de não nulo em chave primária

Abaixo são mostrados exemplos de criação de uma tabela definindo uma chave primária em uma coluna que não é definida como não aceitando o valor nulo. O padrão SQL diz que, neste caso, a restrição de não nulo é implícita, mas o DB2 não implementa desta forma, enquanto o PostgreSQL, o SQL Server e o Oracle seguem o padrão. Também são mostrados comandos para exibir a estrutura da tabela nestes gerenciadores de banco de dados.

PostgreSQL 8.0.0:

```
=> CREATE TABLE c (c1 int, PRIMARY KEY(c1));
=> \d c
```

```

          Tabela "public.c"
  Coluna | Tipo      | Modificadores
-----+-----+-----
  c1     | integer   | not null
Índices:
    "c_pkey" chave primária, btree (c1)

```

SQL Server 2000:

```
CREATE TABLE c (c1 int, PRIMARY KEY(c1));
sp_help c
```

```

Name Owner Type          Created_datetime
----
c     dbo   user table 2005-03-28 11:00:24.027

Column_name Type Computed Length Prec Scale Nullable ...
-----
c1          int   no         4      10    0      no

index_name      index_description                                     index_keys
-----
PK__c__403A8C7D clustered, unique, primary key located on PRIMARY c1

```

Oracle 10g:

```
SQL> CREATE TABLE c (c1 int, PRIMARY KEY(c1));
SQL> DESCRIBE c
```

```

Name Null?      Type
-----
C1     NOT NULL    NUMBER(38)

```

```
SQL> SELECT index_name, index_type, tablespace_name, uniqueness
2 FROM user_indexes
3 WHERE table_name='C';
```

INDEX_NAME	INDEX_TYPE	TABLESPACE_NAME	UNIQUENES
SYS_C005276	NORMAL	USERS	UNIQUE

DB2 9.1:

```
db2 => CREATE TABLE c (c1 int, PRIMARY KEY(c1));
```

SQL0542N "C1" não pode ser uma coluna de uma chave primária ou exclusiva, porque pode conter valores nulos. SQLSTATE=42831

```
db2 => CREATE TABLE c (c1 int NOT NULL, PRIMARY KEY(c1));
```

DB20000I O comando SQL terminou com sucesso.

```
db2 => DESCRIBE TABLE c;
```

Nome da coluna	Esquema do tipo	Nome do tipo	Tamanho	Escala	Nulos
C1	SYSIBM	INTEGER	4	0	Não

1 registro(s) selecionado(s).

```
db2 => DESCRIBE INDEXES FOR TABLE c SHOW DETAIL;
```

Esquema do índice	Nome do índice	Regra unicidade	Número de colunas	Nomes das colunas
SYSIBM	SQL050328184542990	P	1	+C1

1 registro(s) selecionado(s).

Notas

[1] *truth-value* — Na lógica, o valor verdade, ou valor-verdade, é um valor indicando até que ponto uma declaração é verdadeira; Na lógica clássica, os únicos valores verdade possíveis são verdade e falso. Entretanto, são possíveis outros valores em outras lógicas. A lógica intuicionista simples possui os valores verdade: verdade, falso e desconhecido. A lógica *fuzzy*, e outras formas de lógica multi-valoradas, também possuem mais valores verdade do que simplesmente verdade e falso; Algebricamente, o conjunto {verdade, falso} forma a lógica booleana simples. [Dictionary.LaborLawTalk.com](http://www.LaborLawTalk.com) (N. do T.)

[2] Oracle — Para satisfazer a restrição de unicidade, não podem haver duas linhas na tabela com o mesmo valor para a chave única. Entretanto, a chave única formada por uma única coluna pode conter nulos. Para satisfazer uma chave única composta, não podem haver duas linhas na tabela ou na visão com a mesma combinação de valores nas colunas chave. Qualquer linha contendo nulo em todas as colunas chave satisfaz, automaticamente, a restrição. Entretanto, duas linhas contendo nulo em uma ou mais colunas chave, e a mesma combinação de valores para as outras colunas chave, violam a restrição. [Oracle® Database](http://www.Oracle® Database)

SQL Reference 10g Release 1 (10.1) Part Number B10759-01 (N. do T.)

- [3] SQL Server — As restrições de unicidade podem ser utilizadas para garantir que não serão entrados valores duplicados em colunas específicas que não participam da chave primária. Embora tanto a restrição UNIQUE quanto a restrição PRIMARY KEY imponham a unicidade, deve ser utilizado UNIQUE em vez de PRIMARY KEY quando se deseja impor a unicidade de uma coluna, ou combinação de colunas, que não seja chave primária. Podem ser definidas várias restrições UNIQUE em uma tabela, mas somente uma restrição PRIMARY KEY. Também, ao contrário de PRIMARY KEY, a restrição UNIQUE permite o valor nulo. Entretanto, da mesma maneira que qualquer valor participante da restrição UNIQUE, é permitido somente um valor nulo por coluna. — A restrição UNIQUE também pode ser referenciada pela restrição FOREIGN KEY. — Quando é adicionada uma restrição UNIQUE a uma coluna, ou colunas, existentes em uma tabela, os dados existentes nas colunas são verificados para garantir que todos os valores, exceto os nulos, são únicos. [SQL Server 2005 Books Online — UNIQUE Constraints](#) (N. do T.)
- [4] DB2 — A restrição de unicidade é a regra que especifica que os valores de uma chave são válidos apenas se forem únicos na tabela. As colunas especificadas em uma restrição de unicidade devem ser definidas como NOT NULL. O gerenciador de banco de dados usa um índice único para impor a unicidade da chave durante as alterações nas colunas da restrição de unicidade. A restrição de unicidade que é referenciada por uma chave estrangeira de uma restrição referencial é chamada de chave ancestral. Deve ser observado que existe uma distinção entre definir uma restrição de unicidade e criar um índice único: embora ambos imponham a unicidade, o índice único permite colunas com valor nulo e, geralmente, não pode ser utilizado como uma chave ancestral. [DB2 Version 9 for Linux, UNIX, and Windows](#) (N. do T.)

13) Entendendo a Herança de tabelas

Pelo visto o uso de herança no PostgreSQL além de ser um recurso que não está redondo não acrescenta nada ao modelo relacional, devendo ser evitado.

Veja abaixo alguns comentários e o tutorial de uso da documentação oficial.

Em primeiro lugar, o uso de herança não é uma boa ainda, eu não aconselharia, mas dê uma olhada na documentação oficial:
<http://pgdocptbr.sourceforge.net/pg80/tutorial-inheritance.html>

Coutinho

Herança

Vamos criar duas tabelas. A tabela capitais contém as capitais dos estados, que também são cidades. Por consequência, a tabela capitais deve herdar da tabela cidades.

```
CREATE TABLE cidades (  
    nome          text,  
    populacao     float,  
    altitude      int      -- (em pés)  
);  
  
CREATE TABLE capitais (  
    estado        char(2)  
) INHERITS (cidades);
```

Neste caso, as linhas da tabela capitais *herdam* todos os atributos (nome, população e altitude) de sua tabela ancestral, cidades. As capitais dos estados possuem um atributo adicional chamado estado, contendo seu estado. No PostgreSQL uma tabela pode herdar de zero ou mais tabelas, e uma consulta pode referenciar tanto todas as linhas de uma tabela, quanto todas as linhas de uma tabela mais todas as linhas de suas descendentes.

Nota: A hierarquia de herança é, na verdade, um grafo acíclico dirigido. [1]

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:

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

nome	altitude
Las Vegas	2174
Mariposa	1953

Madison | 845

Por outro lado, a consulta abaixo retorna todas as cidades situadas a uma altitude superior a 500 pés, que não são capitais de estados:

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

nome	altitude
Las Vegas	2174
Mariposa	1953

O termo "ONLY" antes de cidades indica que a consulta deve ser executada apenas na tabela cidades, sem incluir as tabelas descendentes de cidades na hierarquia de herança. Muitos comandos mostrados até agora — SELECT, UPDATE e DELETE — suportam esta notação de "ONLY".

Em obsolescência: Nas versões anteriores do PostgreSQL, o comportamento padrão era não incluir as tabelas descendentes nos comandos. Descobriu-se que isso ocasionava muitos erros, e que também violava o padrão SQL:1999. Na sintaxe antiga, para incluir as tabelas descendentes era necessário anexar um * ao nome da tabela. Por exemplo:

```
SELECT * FROM cidades*;
```

Ainda é possível especificar explicitamente a varredura das tabelas descendentes anexando o *, assim como especificar explicitamente para não varrer as tabelas descendentes escrevendo "ONLY". A partir da versão 7.1 o comportamento padrão para nomes de tabelas sem adornos passou a ser varrer as tabelas descendentes também, enquanto antes desta versão o comportamento padrão era não varrer as tabelas descendentes. Para ativar o comportamento padrão antigo, deve ser definida a opção de configuração SQL_Inheritance como desativada como, por exemplo,

```
SET SQL_Inheritance TO OFF;
```

ou definir o parâmetro de configuração [sql_inheritance](#).

Em alguns casos pode-se desejar saber de qual tabela uma determinada linha se origina. Em cada tabela existe uma coluna do sistema chamada tableoid que pode informar a tabela de origem:

```
SELECT c.tableoid, c.nome, c.altitude
FROM cidades c
WHERE c.altitude > 500;
```

tableoid	nome	altitude
139793	Las Vegas	2174
139793	Mariposa	1953
139798	Madison	845

Se for tentada a reprodução deste exemplo, os valores numéricos dos OIDs provavelmente serão diferentes. Fazendo uma junção com a tabela "pg_class" é possível mostrar o nome da tabela:

```
SELECT p.relname, c.nome, c.altitude
```



```
FROM   cidades c, pg_class p
WHERE  c.altitude > 500 AND c.tableoid = p.oid;
```

relname	nome	altitude
cidades	Las Vegas	2174
cidades	Mariposa	1953
capitais	Madison	845

Uma tabela pode herdar de mais de uma tabela ancestral e, neste caso, possuirá a união das colunas definidas nas tabelas ancestrais (além de todas as colunas declaradas especificamente para a tabela filha).

Uma limitação séria da funcionalidade da herança é que os índices (incluindo as restrições de unicidade) e as chaves estrangeiras somente se aplicam a uma única tabela, e não às suas descendentes. Isto é verdade tanto do lado que faz referência quanto do lado que é referenciado na chave estrangeira. Portanto, em termos do exemplo acima:

- Se for declarado cidades.nome como sendo UNIQUE ou PRIMARY KEY, isto não impedirá que a tabela capitais tenha linhas com nomes idênticos aos da tabela cidades e, por padrão, estas linhas duplicadas aparecem nas consultas à tabela cidades. Na verdade, por padrão, a tabela capitais não teria nenhuma restrição de unicidade e, portanto, poderia conter várias linhas com nomes idênticos. Poderia ser adicionada uma restrição de unicidade à tabela capitais, mas isto não impediria um nome idêntico na tabela cidades.
- De forma análoga, se for especificado cidades.nome REFERENCES alguma outra tabela, esta restrição não se propagará automaticamente para a tabela capitais. Neste caso, o problema poderia ser contornado adicionando manualmente a restrição REFERENCES para a tabela capitais.
- Especificar para uma coluna de outra tabela REFERENCES cidades(nome) permite à outra tabela conter os nomes das cidades, mas não os nomes das capitais. Não existe uma maneira boa para contornar este problema.

Estas deficiências deverão, provavelmente, serem corrigidas em alguma versão futura, mas enquanto isso deve haver um cuidado considerável ao decidir se a herança é útil para resolver o problema em questão.

Notas

[1] *Grafo*: uma coleção de vértices e arestas; *Grafo dirigido*: um grafo com arestas unidirecionais; *Grafo acíclico dirigido*: um grafo dirigido que não contém ciclos. [FOLDOC - Free On-Line Dictionary of Computing](#) (N. do T.)

CREATE TABLE -- cria uma tabela

Sintaxe

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE nome_da_tabela (
  { nome_da_coluna tipo_de_dado [ DEFAULT expressão_padrão ]
  [ restrição_de_coluna [ ... ] ]
  | restrição_de_tabela
  | LIKE tabela_ancestral [ { INCLUDING | EXCLUDING } DEFAULTS ] } [, ... ]
```

```
)
[ INHERITS ( tabela_ancestral [, ... ] ) ]
```

```
...
INHERITS ( tabela_ancestral [, ... ] )
```

A cláusula opcional INHERITS (herda) especifica uma lista de tabelas das quais a nova tabela herda, automaticamente, todas as colunas.

O uso de INHERITS cria um relacionamento persistente entre a nova tabela descendente e suas tabelas ancestrais. As modificações de esquema nas tabelas ancestrais normalmente se propagam para as tabelas descendentes também e, por padrão, os dados das tabelas descendentes são incluídos na varredura das tabelas ancestrais.

Se existir o mesmo nome de coluna em mais de uma tabela ancestral é relatado um erro, a menos que o tipo de dado das colunas seja o mesmo em todas as tabelas ancestrais. Não havendo conflito, então as colunas duplicadas são combinadas para formar uma única coluna da nova tabela. Se a lista de nomes de colunas da nova tabela contiver um nome de coluna que também é herdado, da mesma forma o tipo de dado deverá ser o mesmo das colunas herdadas, e a definição das colunas será combinada em uma única coluna. Entretanto, declarações de colunas novas e herdadas com o mesmo nome não precisam especificar restrições idênticas: todas as restrições fornecidas em qualquer uma das declarações são combinadas, sendo todas aplicadas à nova tabela. Se a nova tabela especificar explicitamente um valor padrão para a coluna, este valor padrão substituirá o valor padrão das declarações herdadas da coluna. Se não especificar, toda tabela ancestral que especificar um valor padrão para a coluna deverá especificar o mesmo valor, ou será relatado um erro.

Herança

Heranças múltiplas por meio da cláusula INHERITS é uma extensão do PostgreSQL à linguagem. O SQL:1999 (mas não o SQL-92) define herança única utilizando uma sintaxe diferente e semânticas diferentes. O estilo de herança do SQL:1999 ainda não é suportado pelo PostgreSQL.

pg_inherits

O catálogo `pg_inherits` registra informações sobre hierarquias de herança de tabelas. Existe uma entrada para cada tabela diretamente descendente no banco de dados (As descendências indiretas podem ser determinadas seguindo a cadeia de entradas).

Tabela 42-17. Colunas de *pg_inherits*

Nome	Tipo	Referencia	Descrição
inhrelid	oid	pg_class.oid	OID da tabela descendente.
inhparent	oid	pg_class.oid	OID da tabela ancestral.
inhseqno	int4		Se houver mais de um ancestral direto para a tabela descendente (herança múltipla), este número informa a ordem pela qual as

Nome	Tipo	Referencia	Descrição
			colunas herdadas devem ser dispostas. O contador começa por 1.

Herança no PostgreSQL

Olá amigos,

nesse artigo vamos tratar um conceito que tem, a cada dia, sido mais discutido no âmbito dos bancos de dados relacionais, em especial o PostgreSQL, o conceito de herança.

Inicialmente, herança é um conceito da orientação à objetos onde, uma determinada classe herda características (no caso das classes - propriedades ou atributos e métodos ou funções) de uma outra "super" classe. A relação de herança pode ser lida da seguinte forma: "a classe que herda É UM TIPO DA super classe (classe herdada)".

Assim teríamos, por exemplo, uma classe funcionário que contém os dados comuns dos funcionários (matricula, nome, dataNascimento, ...) e uma classe gerente que É UM TIPO DE funcionário, mas com algumas outras características exclusivas como por exemplo (percentualParticipacaoLucro, telCeluar, ...).

Então, no exemplo acima temos uma classe gerente que herda da classe funcionário. Isso significa que não precisamos replicar todas as características de um funcionário para um gerente. A herança se encarrega de fazer isso e o gerente passa a ter todas as características do funcionário + as suas características próprias.

Agora falando do PostgreSQL. Ao contrário do que lemos muitas vezes por ai, o PostgreSQL não é um SGBD (Sistema de Gerência de Banco de Dados) Orientado a Objetos. Ele é o que chamamos de Sistema de Gerência de Banco de Dados Relacional Estendido ou Objeto-Relacional. Essa sua Extensão nos permite utilizar alguns conceitos próprios de um Banco Orientado a Objetos, como por exemplo tipos de dados definidos pelo usuário e herança. No caso da herança no PostgreSQL, uma tabela pode herdar de nenhuma, uma, ou várias tabelas.

Implementando o exemplo acima no PostgreSQL teríamos:

```
create table funcionario(  
    matricula int,  
    nome varchar,  
    dataNascimento Date,  
    primary key(matricula)  
)
```

Ao executar o comando acima, o PostgreSQL automaticamente cria a tabela funcionário (com a estrutura informada) no esquema public.

```
create table gerente(  
    percentParticipacaoLucro int,  
    telCel varchar  
) inherits (funcionario);
```

Ao executar o comando acima, o PostgreSQL cria uma tabela gerente com os atributos de funcionário + os atributos de gerente.

Então agora temos as seguintes tabelas em nossa base:

Funcionário
matricula int,
nome varchar,
datanascimento date,
CONSTRAINT funcionario_pkey PRIMARY KEY (matricula)

Gerente
matricula int4 NOT NULL,
nome varchar,
datanascimento date,
percentparticipacalucro int4,
telcel varchar

Notem que ao contrário do que dissemos acima com relação às classes, na herança de tabelas no PostgreSQL os campos herdados se repetem "fisicamente" nas duas tabelas.

Para inserir um gerente usamos:
insert into gerente values (1000, 'Hesley', '01/01/1975', 10, '99999999'); -- Ao inserir um gerente, automaticamente os atributos herdados (matricula, nome, dataNascimento) são inseridos também na tabela de funcionários.

Para inserir um funcionário usamos:
insert into funcionario values (2000, 'Maria', '02/02/1980'); -- Os dados são inseridos somente na tabela funcionários e não na tabela gerente.

Para visualizarmos os dados das tabelas.

```
select * from gerente;  
Retorna  
1000;"Hesley";"1975-01-01";10;"99999999"
```

```
select * from funcionario;  
Retorna  
2000;"Maria";"1980-02-02"  
1000;"Hesley";"1975-01-01"
```

Bom, agora eu quero saber quais os funcionários que não são gerentes. Em uma situação normal (sem herança) podia usar o conceito de subquery para selecionar os funcionários que não estão na tabela gerente. Uma vez implementada a herança, basta eu usar a palavrinha ONLY como na query abaixo.

```
select * from only funcionario; -- Retorna somente os funcionários que não são gerentes.  
Retorna  
2000;"Maria";"1980-02-02"
```

Como no SELECT, os comando UPDATE e DELETE também suportam o uso do "ONLY".

Existem algumas deficiências com relação à restrição de integridade no uso das heranças. Por exemplo, a tabela funcionário foi criada com uma chave primária matrícula, o que impede que eu possua dois funcionários com uma mesma matrícula, entretanto essa restrição não é válida para a tabela herdada. Dessa forma poderiam existir dois gerentes com a mesma matrícula.

Com relação à utilidade da herança no PostgreSQL, ainda não consegui ver muita utilidade (posso estar enganado) e a maioria dos textos que tenho lido sobre o tema, desencoraja seu uso. Particularmente, acredito que os recursos relacionais existentes no PostgreSQL e nos outros bancos relacionais são suficientes e adequados para resolver os problemas que seriam tratados usando a herança (posso estar enganado).

Hesley Py

<http://www.devmedia.com.br/articles/viewcomp.asp?comp=9182>

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. A equipe de desenvolvimento do PostgreSQL sempre teve uma grande preocupação em manter a compatibilidade com os padrões SQL92/SQL99.

Boa tarde pessoal,

Vi **alguns** posts passados questionando **a herança** na nova versão 8.3 e pelo que entendi não melhorou e nem vai melhorar.

Bem... há **algum** tempo eu fui **forçado** (monografia) **a** modelar um banco que ficasse implementado o conceito de **herança**. No PG estava indo legal **até** que vieram os problemas de integridade referencial, procurei **ajuda** e percebi que **herança** no PG era uma "bomba". Coloquei os miolos para funcionar e o Tico-e-Teco me deram uma idéia de implementação, daí estou passando em **anexo** um script de como eu consegui implementar o conceito de **herança**.

Para o que eu precisava serviu legal e **até agora** não deu problema e talvez, quem sabe, essa idéia consiga **ajudar alguém**.

Celso Henrique Mendes Ferreira

Analista de Requisitos

www.itecgyn.com.br

Abaixo a solução do Celso:

-- LANGUAGE

CREATE PROCEDURAL LANGUAGE plpgsql;

-- TRIGGERS

```
CREATE FUNCTION adicionarpessoa() RETURNS "trigger"
AS $$
begin
    insert into tbPessoa (cdPessoa, tpPessoa, nmPessoa, cpf, dtNascimento)
        values (new.cdPessoa, new.tpPessoa, new.nmPessoa, new.cpf, new.dtNascimento);
    return null;
end;
$$
LANGUAGE plpgsql;
```

```
CREATE FUNCTION atualizarpessoa() RETURNS "trigger"
AS $$
begin
    update tbPessoa set
        tpPessoa    = new.tpPessoa,
        nmPessoa    = new.nmPessoa,
        cpf         = new.cpf,
        dtNascimento = new.dtNascimento
    where cdPessoa = old.cdPessoa;
    return null;
end;
$$
LANGUAGE plpgsql;
```

```
CREATE FUNCTION removerpessoa() RETURNS "trigger"
AS $$
begin
    delete from tbPessoa where cdPessoa = old.cdPessoa;
    return null;
end;
$$
LANGUAGE plpgsql;
```

```
-- SEQUENCE
```

```
CREATE SEQUENCE tbpessoa_cd Pessoa_seq
INCREMENT BY 1
NO MAXVALUE
NO MINVALUE
```

CACHE 1;

-- TABLES

```
CREATE TABLE tbpessoa (  
    cdpessoa integer NOT NULL,  
    nmpessoa character varying(50),  
    tppessoa integer,  
    cpf character varying(20),  
    dtnascimento date  
);
```

```
ALTER TABLE ONLY tbpessoa  
    ADD CONSTRAINT pk_tbpessoa PRIMARY KEY (cdpessoa);
```

```
-----  
CREATE TABLE tbaluno (  
    cdpessoa integer DEFAULT nextval('tbpessoa_cdpessoa_seq'::regclass) NOT NULL,  
    nmpessoa character varying(50),  
    tppessoa integer DEFAULT 0,  
    matricula character varying(20),  
    cpf character varying(20),  
    dtnascimento date  
);
```

```
ALTER TABLE ONLY tbaluno  
    ADD CONSTRAINT pk_tbaluno PRIMARY KEY (cdpessoa);
```

```
CREATE TRIGGER tgadicionarpessoa  
    AFTER INSERT ON tbaluno  
    FOR EACH ROW  
    EXECUTE PROCEDURE adicionarpessoa();
```

```
CREATE TRIGGER tgatualizarpessoa  
    AFTER UPDATE ON tbaluno  
    FOR EACH ROW  
    EXECUTE PROCEDURE atualizarpessoa();
```

```
CREATE TRIGGER tgremoverpessoa  
    AFTER DELETE ON tbaluno  
    FOR EACH ROW  
    EXECUTE PROCEDURE removerpessoa();
```

```
CREATE TABLE tbfuncionario (  
    cdpessoa integer DEFAULT nextval('tbpessoa_cdpessoa_seq'::regclass) NOT NULL,  
    nmpessoa character varying(50),  
    tppessoa integer DEFAULT 1,  
    cargo character varying(100),  
    cdfuncionario character varying(20),  
    cpf character varying(20),  
    dtnascimento date  
);  
  
ALTER TABLE ONLY tbfuncionario  
    ADD CONSTRAINT pk_tbfuncionario PRIMARY KEY (cdpessoa);  
  
CREATE TRIGGER tgadicionarpessoa  
    AFTER INSERT ON tbfuncionario  
    FOR EACH ROW  
    EXECUTE PROCEDURE adicionarpessoa();  
  
CREATE TRIGGER tgatualizarpessoa  
    AFTER UPDATE ON tbfuncionario  
    FOR EACH ROW  
    EXECUTE PROCEDURE atualizarpessoa();  
  
CREATE TRIGGER tgremoverpessoa  
    AFTER DELETE ON tbfuncionario  
    FOR EACH ROW  
    EXECUTE PROCEDURE removerpessoa();
```

Em 26/02/2008, às 00:37, Euler Taveira de Oliveira escreveu:

```
> Celso Henrique Mendes Ferreira wrote:  
>  
>> Para o que eu precisava serviu legal e até agora não deu problema e  
>> talvez, quem sabe, essa idéia consiga ajudar alguém.  
>>  
> Mas isso *não* é herança. É de certo modo uma "replicação parcial" das  
> tabelas do banco de dados.
```

De **acordo** com o Euler, porque replicar os **atributos** de pessoa (cpf e dtnascimento) em **aluno** por exemplo?

Acho que se mapearmos os conceitos OO da seguinte forma:

Classe -> Variável de relação

Atributo de classe -> **Atributo** de relação

Objeto -> Tupla

Composição de objetos -> Relacionamentos

O conceito mais próximo da **herança** no banco relacional será especialização/generalização.

PS.: Se **alguém** me **acusar** de estar cometendo um grande erro de mapeamento (como no livro do date) vou responder que não proponho um banco relacional/oo mas um mapeamento mais "solto" entre um SGBDR puro e modelagem OO (e uma definição bem geral de classes).

Pelo contrário, Diogo, você está seguindo o Date direitinho, Terceiro Manifesto e tudo.

A única nota que eu faria — que de maneira **alguma** é questionamento do que você escreveu — é que 'objeto' é um conceito **ambíguo** entre livros e implementações várias, de modo que um objeto pode ser tanto uma tupla quanto uma relação. Mas concordo contigo em que o mais são, em princípio, me parece objeto como tupla.
Leandro

Diogo,

Eu concordo que não **atentei** para o detalhe da Especialização/Generalização. **Assumindo a** OO eu não precisaria dos campos nome, cpf e dtnascimento nas tabelas filhas, **apenas** uma chave estrangeira de relacionamento. **Assim**, qualquer **alteração** na estrutura desses campos poderia ser feita direto na tabela mãe, o que não poderia ser feito com o esquema que criei, pois nele eu teria que **alterar** não só **a** mãe, mas em todas **as** filhas.

Um detalhe que não comentei é que essa implementação foi feita baseada no funcionamento de **herança** dentro do próprio PG, onde um registro numa tabela filha é **automaticamente** inserido na tabela mãe. Vendo pela Especialização/Generalização percebo que **a** estrutura fica mais simples e flexível, tornando **as** triggers completamente desnecessárias. Só que isso não é nada **além** de normalização de banco de dados, daí penso o seguinte: qual **a** dificuldade que os desenvolvedores têm em corrigir o problema de **herança** no PG?

--

Celso Henrique Mendes Ferreira

2008/2/26, Celso Henrique Mendes Ferreira <celsohenrique@...>:

> Vendo pela

> Especialização/Generalização percebo que **a** estrutura fica mais simples e

> flexível, tornando **as** triggers completamente desnecessárias. Só que isso não

> é nada **além** de normalização de banco de dados, daí penso o seguinte: qual **a**

> dificuldade que os desenvolvedores têm em corrigir o problema de **herança** no

> PG?

Exatamente — é que não tem ganho **algum**.

Atualmente, o único uso legítimo que conheço de **herança** no **PostgreSQL** é particionamento de tabelas, que todo mundo **admite** que é **apenas** uma gambiarra, merecendo uma implementação melhor.

Deve haver outros usos legítimos, mas todos gambiarras como o particionamento.

OO em bases de dados foi uma idéia infeliz.

Leandro

14) Funções em SQL e em PL/pgSQL, Gatilhos e Regras

Funções em SQL no PostgreSQL

Funções em SQL no PostgreSQL executam rotinas com diversos comandos em SQL em seu interior e retornará o resultado da última consulta da lista.

Uma função em SQL também pode retornar um consulto, quando especificamos o retorno da função como sendo do tipo SETOF. Neste caso todos os registros da última consulta serão retornados.

Exemplos simples:

Criando:

```
CREATE FUNCTION dois() RETURNS integer AS '  
    SELECT 2;  
' LANGUAGE SQL;
```

Executando:

```
SELECT dois();
```

Excluindo empregados com salário negativo:

```
CREATE FUNCTION limpar_emp() RETURNS void AS '  
    DELETE FROM empregados  
        WHERE salario < 0;  
' LANGUAGE SQL;
```

```
SELECT limpar_emp();
```

Função Passando Parâmetros:

```
CREATE FUNCTION adicao(integer, integer) RETURNS integer AS $$  
    SELECT $1 + $2;  
$$ LANGUAGE SQL;
```

```
SELECT adicao(1, 2) AS resposta;
```

Observe que psraâmetros são usados como: \$1, \$2, etc.

Outro exemplo (debitando valor em uma conta):

```
CREATE FUNCTION debitar (integer, numeric) RETURNS integer AS $$  
    UPDATE banco  
        SET saldo = saldo - $2  
        WHERE conta = $1;  
    SELECT 1;  
$$ LANGUAGE SQL;
```

Exemplo com retorno mais interessante:

```
CREATE FUNCTION debitar2 (integer, numeric) RETURNS numeric AS $$  
    UPDATE banco  
        SET saldo = saldo - $2  
        WHERE conta = $1;
```

```
SELECT saldo FROM banco WHERE conta = $1;
$$ LANGUAGE SQL;
```

Exemplo com tipos compostos

```
CREATE TABLE empregados (
    nome          text,
    salario       numeric,
    idade         integer,
    cubiculo      point
);

insert into empregados values('Ribamar FS', 3856.45, 51, '(2,1)');

CREATE FUNCTION dobrar_salario(empregado) RETURNS numeric AS $$
    SELECT $1.salario * 2 AS salario;
$$ LANGUAGE SQL;

SELECT nome, dobrar_salario(empregados.*) AS sonho
FROM empregados
WHERE empregados.cubiculo ~= point '(2,1)';
```

Retornando um tipo composto:

```
CREATE FUNCTION novo_empregado() RETURNS empregados AS $$
    SELECT text 'Brito Cunha' AS nome,
           1000.0 AS salario,
           25 AS idade,
           point '(2,2)' AS cubiculo;
$$ LANGUAGE SQL;
```

Definindo de outra maneira:

```
CREATE FUNCTION novo_empregado() RETURNS empregados AS $$
    SELECT ROW('Brito Cunha', 1000.0, 25, '(2,2)')::empregados;
$$ LANGUAGE SQL;
```

```
select novo_empregado();
select (novo_empregado()).nome
```

```
CREATE FUNCTION recebe_nome(empregados) RETURNS text AS $$
    SELECT $1.nome;
$$ LANGUAGE SQL;
```

Usando uma outra função como parâmetro:

```
SELECT recebe_nome(novo_empregado());
```

Usando parâmetros de saída:

```
CREATE FUNCTION adicionar (IN x int, IN y int, OUT soma int)
AS 'SELECT $1 + $2'
LANGUAGE SQL;
```

```
SELECT adicionar(3,7);
```

```
CREATE FUNCTION soma_n_produtos (x int, y int, OUT soma int, OUT produtos int)
AS 'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;
```

```
SELECT * FROM soma_n_produtos(11,42);
```

Criando Tipo Denifido pelo Usuário

```
CREATE TYPE soma_produtos AS (soma int, produto int);
```

```
CREATE FUNCTION soma_n_produtos (int, int) RETURNS soma_produtos
AS 'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;
```

```
select soma_n_produtos (4,5);
```

Tabela como fonte de Funções

```
CREATE TABLE tab (id int, subid int, nome text);
INSERT INTO tab VALUES (1, 1, 'Joe');
INSERT INTO tab VALUES (1, 2, 'Ed');
INSERT INTO tab VALUES (2, 1, 'Mary');
```

```
CREATE FUNCTION recebe_tab(int) RETURNS tab AS $$
    SELECT * FROM tab WHERE id = $1;
$$ LANGUAGE SQL;
```

```
SELECT *, upper(nome) FROM recebe_tab(1) AS tab1;
```

Observe o retorno:

1	1	Joe	JOE
---	---	-----	-----

Primeiro retornam todos os campos da tab (*) e depois retorna o nome em maiúsculas.

Função Retornando Conjunto

```
CREATE FUNCTION recebe_tabela(int) RETURNS SETOF tabela1 AS $$
    SELECT * FROM tabela1 WHERE id = $1;
$$ LANGUAGE SQL;
```

```
SELECT * FROM recebe_tabela(1) AS tab1;
```

Retornando múltiplos registros:

```
CREATE FUNCTION soma_n_produtos_com_tab (x int, OUT soma int, OUT produtos int)
RETURNS SETOF record AS $$
    SELECT x + tab.y, x * tab.y FROM tab;
$$ LANGUAGE SQL;
```

Precisamos indicar o retorno com RETURNS SETOF record para que sejam retornados vários registros.

Exemplo que retorna conjunto através de select:

```
CREATE FUNCTION listchildren(text) RETURNS SETOF text AS $$
    SELECT name FROM nodes WHERE parent = $1
$$ LANGUAGE SQL;

SELECT * FROM nodes;
```

Funções Polimórficas

```
CREATE FUNCTION make_array(anelement, anelement) RETURNS anyarray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;

SELECT make_array(1, 2) AS intarray, make_array('a'::text, 'b') AS textarray;

CREATE FUNCTION is_greater(anelement, anelement) RETURNS boolean AS $$
    SELECT $1 > $2;
$$ LANGUAGE SQL;

SELECT is_greater(1, 2);

CREATE FUNCTION dup (f1 anelement, OUT f2 anelement, OUT f3 anyarray)
AS 'select $1, array[$1,$1]' LANGUAGE sql;

SELECT * FROM dup(22);
```

Retornando sempre a última consulta (independente do parâmetro passado):

```
CREATE OR REPLACE FUNCTION somar(integer) RETURNS BIGINT AS $$
    SELECT sum(codigo) as x from cliente where codigo < $1;
    SELECT sum(codigo) as x from cliente where codigo < 2;
$$ LANGUAGE SQL;

SELECT somar(3) AS resposta;
```

Exemplo com CASE

```
create function categoria(int) returns char as
,
select
    case when idade < 20 then 'a'
         when idade >= 20 and idade < 30 then 'b'
         when idade >= 30 then 'c'
    end as categoria
    from clientes where codigo = $1
,
language 'sql';

create function get_numdate (date) returns integer as
,
select (substr($1 , 6,2) || substr( $1 , 9,2))::integer;
,
language 'SQL';
```

```
create function get_cliente (int) returns varchar as
,
select nome from clientes where codigo = $1;
,
language 'SQL';
```

```
create function get_cliente (int) returns varchar as
,
select nome from clientes where codigo = $1;
,
language 'SQL';
```

```
create function menores() returns setof clientes as
,
select * from clientes where idade < 18;
,
language 'SQL';
```

```
create function km2mi (float) returns float as
,
select $1 * 0.6;
,
language 'SQL';
```

```
create function get_signo (int) returns varchar as
,
select
    case
        when $1 <=0120 then \'capricornio\'
        when $1 >=0121 and $1 <=0219 then \'aquario\'
        when $1 >=0220 and $1 <=0320 then \'peixes\'
        when $1 >=0321 and $1 <=0420 then \'aries\'
        when $1 >=0421 and $1 <=0520 then \'touro\'
        when $1 >=0521 and $1 <=0620 then \'gemeos\'
        when $1 >=0621 and $1 <=0722 then \'cancer\'
        when $1 >=0723 and $1 <=0822 then \'leao\'
        when $1 >=0823 and $1 <=0922 then \'virgem\'
        when $1 >=0923 and $1 <=1022 then \'libra\'
        when $1 >=1023 and $1 <=1122 then \'escorpiao\'
        when $1 >=1123 and $1 <=1222 then \'sagitario\'
        when $1 >=1223 then \'capricornio\'
    end as signo
,
language 'SQL';
```

Consulta usando duas funções:

```
select ref_cliente, get_cliente(ref_cliente),  
       quantidade, preco, ref_produto, get_produto(ref_produto)  
from compras;
```

Alguns Exemplos de uso das funções:

```
select get_numdate('14/04/1985');  
get_numdate: 414
```

```
select get_signo(get_numdate('14/04/1985'));  
get_signo: aries
```

```
select get_numdate('24/09/1980');  
get_numdate: 924
```

```
select get_signo(get_numdate('24/09/1980'));  
get_signo: libra
```

Mais Detalhes em:

<http://pgdocptbr.sourceforge.net/pg80/xfunc-sql.html>

<http://www.postgresql.org/docs/current/static/xfunc-sql.html>

http://www.linux-magazine.com.br/images/uploads/pdf_aberto/LM07_postgresql.pdf

Stored Procedures no PostgreSQL (Usando funções em PL/PgSQL)

No PostgreSQL todos os procedimentos armazenados (stored procedures) são funções, apenas elas usam linguagens de programação procedurais como PL/pgSQL, java, php, ruby, tcl, python, perl, etc.

A linguagem SQL é a que o PostgreSQL (e a maioria dos SGBDs relacionais) utiliza como linguagem de comandos. É portátil e fácil de ser aprendida. Entretanto, todas as declarações SQL devem ser executadas individualmente pelo servidor de banco de dados.

Isto significa que o aplicativo cliente deve enviar o comando para o servidor de banco de dados, aguardar que seja processado, receber os resultados, realizar algum processamento, e enviar o próximo comando para o servidor. Tudo isto envolve comunicação entre processos e pode, também, envolver tráfego na rede se o cliente não estiver na mesma máquina onde se encontra o servidor de banco de dados.

Executar vários comandos de uma vez:

Usando a linguagem PL/pgSQL pode ser agrupado um bloco de processamento e uma série de comandos *dentro* do servidor de banco de dados, juntando o poder da linguagem procedural com a facilidade de uso da linguagem SQL, e economizando muito tempo, porque não há necessidade da sobrecarga de comunicação entre o cliente e o servidor. Isto pode aumentar o desempenho consideravelmente.

Vantagens no uso da linguagem procedural PL/pgSQL:

- Com ela podemos criar funções e triggers procedurais;
- Adicionar estruturas de controle para a linguagem SQL;
- Executar cálculos complexos;
- Herdar todos os tipos definidos pelo usuário, funções e operadores
- Pode ser definida para ser confiável para o servidor
- É fácil de usar

Nas funções em PL/pgSQL também podemos usar todos os tipos de dados, funções e operadores do SQL.

Argumentos Suportados e Tipos de Dados de Retorno

Funções escritas em PL/pgSQL podem aceitar como argumento qualquer tipo de dados escalar ou array suportado pelo servidor e podem retornar como resultado qualquer desses tipos.

Também podem aceitar ou retornar qualquer tipo composto (tipo row) especificado pelo nome.

Também podemos declarar uma função em PL/pgSQL retornando record, que permite que o resultado seja um registro cujos campos sejam determinados especificando-se na consulta.

Funções em PL/pgSQL também podem ser declaradas para aceitar e retornar os tipos polimórficos `anyelement`, `anyarray`, `anynonarray` e `anyenum`.

Também podem retornar um conjunto (uma tabela) de qualquer tipo de dados que pode retornar como uma instância simples. A função gera sua saída executando RETURN NEXT para cada elemento desejado do conjunto resultante ou usando RETURN QUERY para a saída resultante da avaliação da consulta.

Uma função em PL/pgSQL também, finalmente, pode ser declarada para retornar void, caso o retorno não tenha valor útil.

As funções em PL/pgSQL também podem ser declaradas com parâmetros de saída no lugar de uma especificação explícita do tipo de retorno. Isso não adiciona nenhuma capacidade fundamental para a linguagem, mas isso é conveniente, especialmente para retornar múltiplos valores.

A PL/pgSQL é uma linguagem estruturada em blocos.

O texto completo da definição de uma função precisa ser um bloco. Um bloco é definido como:

```
[ <<rótulo>> ]  
[ DECLARE  
    declaração de variáveis ];  
BEGIN  
    Instruções;  
END [ rótulo ];
```

Delimitador de blocos

Cada DECLARE e cada BEGIN precisam terminar com ponto e vírgula.

O rótulo só é necessário se pretendemos usá-lo em um comando EXIT ou para qualificar os nomes das variáveis declaradas.

Todas as palavras-chaves são case-insensitivas.

A função **sempre** tem que retornar um valor.

Podemos retornar qualquer tipo de dados normal como boolean, text, varchar, integer, double, date, time, void, etc.

Comentários:

```
-- Comentário para uma linha  
/* Comentário para  
múltiplas  
linhas  
*/
```

Exemplo Simples:

```
CREATE or replace FUNCTION f_ola_mundo() RETURNS varchar AS $$  
DECLARE  
    ola varchar := 'Olá';  
BEGIN
```

```
PERFORM ola;  
RETURN ola;  
END;  
$$ LANGUAGE PLpgSQL;
```

Executando:

```
select f_ola_mundo()
```

Obs.: Uma única função pode ter até 16 parametros.

Create function numero(num1 text) returns integer as

```
$$  
Declare  
    resultado integer;  
Begin  
    resultado := num1;  
    return resultado;  
End;  
$$ language 'plpgsql';
```

Create function texto(texto1 text, texto2 text) returns char as

```
$$  
Declare  
    resultado text;  
Begin  
    resultado := texto1 || texto2; --- || é o caracter para concatenação.  
    return resultado;  
End;  
$$ language 'plpgsql';
```

Função que recebe uma string e retorna seu comprimento:

create function calc_comprim(text) returns int4 as

```
,  
    declare  
        textoentrada alias for $1;  
        resultado int4;  
    begin  
        resultado := (select length(textoentrada));  
        return resultado;  
    end;  
,  
language 'plpgsql';
```

Chamando a função:

```
select calc_comprim('Ribamar');
```

Excluir função:

```
drop calc_comprim(text);
```

Exemplo mais elaborado

```
CREATE FUNCTION f_escopo() RETURNS integer AS $$
DECLARE
    quantidade integer := 30;
BEGIN
    RAISE NOTICE 'Quantidade aqui vale %', quantidade; -- Imprime 30
    quantidade := 50;
    --
    -- Criar um sub-bloco
    --
    DECLARE
        quantidade integer := 80;
    BEGIN
        RAISE NOTICE 'Quantidade aqui vale %', quantidade; -- Imprime 80
        RAISE NOTICE 'Quantidade aqui vale %', tabelaext.quantidade; -- Imprime
50
    END;
    RAISE NOTICE 'Quantidade aqui vale %', quantidade; -- Imprime 50
    RETURN quantidade;
END;
$$ LANGUAGE PLpgSQL;
```

Declaração de Variáveis

Todas as variáveis usadas em um bloco precisam ser declaradas na seção DECLARE do bloco. A única exceção é para variáveis de iteração do loop for.

Variáveis em PL/pgSQL podem ter qualquer tipo de dados do SQL, como integer, varchar, char, etc.

Alguns exemplos de declaração de variáveis:

```
user_id integer;
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
aow RECORD;
```

Sintaxe geral de declaração de variável:

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

Atribuição de Variáveis (com :=):

```
quantity integer DEFAULT 32;
url varchar := 'http://mysite.com';
user_id CONSTANT integer := 10;
```

```
create function sec_func(int4,int4,int8,text,varchar) returns int4 as'
declare
myint constant integer := 5;
```

```

mystring char default "T";
firstint alias for $1;
secondint alias for $2;
third alias for $3;
fourth alias for $4;
fifth alias for $5;
ret_val int4;

```

```
begin
```

```

select into ret_val employee_id from masters where code_id = firstint and dept_id = secondint;
return ret_value;
end;
'language 'plpgsql';

```

A função acima precisa ser chamada assim:

```
select sec_func(3,4,cast(5 as int8),cast('trial text' as text),'some text');
```

Números passados como parâmetros tem valor default int4. Então precisamos fazer um cast para int8 ou bigint.

Alias para Parâmetros de Funções

Os parâmetros passados para as funções são nomeados como \$1, \$2, ... Para facilitar a leitura podemos criar alias para os parâmetros. Recomenda-se criar junto ao parâmetro na criação da função.

Na criação da função:

```

CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE PLpgSQL;

```

Na seção Declare:

```

CREATE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE PLpgSQL;

```

No primeiro caso subtotal pode ser referenciada como sales_tax.subtotal mas no segundo caso não pode.

```

CREATE FUNCTION instr(vvarchar, integer) RETURNS integer AS $$
DECLARE
    v_string ALIAS FOR $1;
    index ALIAS FOR $2;
BEGIN
    -- some computations using v_string and index here
END;

```

```
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION concat_selected_fields(in_t sometablename) RETURNS text AS $$
BEGIN
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION sales_tax(subtotal real, OUT tax real) AS $$
BEGIN
    tax := subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod int) AS $$
BEGIN
    sum := x + y;
    prod := x * y;
END;
$$ LANGUAGE plpgsql;
```

Quando um tipo de retorno de uma função é declarado como tipo polimórfico, um parâmetro especial \$0 é criado. Este é o tipo de dados de retorno da função.

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement)
RETURNS anyelement AS $$
DECLARE
    result ALIAS FOR $0;
BEGIN
    result := v1 + v2 + v3;
    RETURN result;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement,
OUT sum anyelement)
AS $$
BEGIN
    sum := v1 + v2 + v3;
END;
$$ LANGUAGE plpgsql;
```

Copiando Tipos

variavel%TYPE

%TYPE contém o tipo de dados de uma variável ou campo de tabela.

Para declarar uma variável chamada user_id com o mesmo tipo de dados de users.user_id:

```
user_id users.user_id%TYPE;
```

Usando %TYPE não precisamos conhecer o tipo de dados.

Tipos de Registros

```
nome nome_da_tabela%ROWTYPE;
nome nome_do_tipo_composto;
```

Uma variável do tipo composto é chamada de variável row (linha). Este tipo de variável pode armazenar toda uma linha de resultado de um comando SELECT ou FOR, desde que o conjunto de colunas do comando corresponda ao tipo declarado para a variável. Os campos individuais do valor linha são acessados utilizando a notação usual de ponto como, por exemplo, `variável_linha.campo`. Uma variável-linha pode ser declarada como tendo o mesmo tipo de dado das linhas de uma tabela ou de uma visão existente, utilizando a notação `nome_da_tabela%ROWTYPE`; ou pode ser declarada especificando o nome de um tipo composto (Uma vez que todas as tabelas possuem um tipo composto associado, que possui o mesmo nome da tabela, na verdade não faz diferença para o PostgreSQL se `%ROWTYPE` é escrito ou não, mas a forma contendo `%ROWTYPE` é mais portátil).

Os parâmetros das funções podem ser de tipo composto (linhas completas da tabela). Neste caso, o identificador correspondente `$n` será uma variável linha, e os campos poderão ser selecionados a partir deste identificador como, por exemplo, `$1.id_usuario`.

Somente podem ser acessadas na variável tipo-linha as colunas definidas pelo usuário presentes na linha da tabela, a coluna OID e as outras colunas do sistema não podem ser acessadas por esta variável (porque a linha pode ser de uma visão). Os campos do tipo-linha herdam o tamanho do campo da tabela, ou a precisão no caso de tipos de dado como `char(n)`.

```
create function third_func(text) returns varchar as'
declare
fir_text alias for $1;
sec_text mytable.last_name%type;
--here in the line above will assign the variable sec_text the datatype of
--of table mytable and column last_name.
begin

--some code here
end;
'language 'plpgsql';
```

`%ROWTYPE` representa a estrutura de uma tabela.

Abaixo está mostrado um exemplo de utilização de tipo composto:

```
CREATE FUNCTION mesclar_campos(t_linha nome_da_tabela) RETURNS text AS $$
DECLARE
    t2_linha nome_tabela2%ROWTYPE;
BEGIN
    SELECT * INTO t2_linha FROM nome_tabela2 WHERE ... ;
    RETURN t_linha.f1 || t2_linha.f3 || t_linha.f5 || t2_linha.f7;
END;
$$ LANGUAGE plpgsql;

SELECT mesclar_campos(t.*) FROM nome_da_tabela t WHERE ... ;
```

```
CREATE FUNCTION populate() RETURNS integer AS $$  
DECLARE  
    -- declarações  
BEGIN  
    PERFORM minha_funcao();  
END;  
$$ LANGUAGE plpgsql;
```

create function third(int4) returns varchar as'

declare

myvar alias for \$1;
mysecvar mytable%rowtype;
mythirdvar varchar;

begin

select into mysecvar * from mytable where code_id = myvar;
--now mysecvar is a recordset
mythirdvar := mysecvar.first_name|| ' '|| mysecvar.last_name;
--|| is the concatenation symbol
--first_name and last_name are columns in the table mytable
return mythirdvar;
end;
'language 'plpgsql';

create function mess() returns varchar as'

declare

myret := "done";

begin

raise notice "hello there";
raise debug "this is the debug message";
raise exception "this is the exception message";

return myret;

end;

'language 'plpgsql';

Chamar com:

select mess();

Função dentro de Função

Podemos chamar uma função de dentro de outro sem retornar um valor.

create function test(int4,int4,int4) returns int4 as'

declare

first alias for \$1;


```
sec alias for $2;  
third alias for $3;  
  
perform another_func(first,sec);  
return (first + sec);  
  
end;  
'language 'plpgsql';
```

Se a função acima for executada fará referência ao OID da minha_funcao() no plano de execução gerado para a instrução PERFORM. Mais tarde, se a função minha_funcao() for removida e recriada, então populate() não vai mais conseguir encontrar minha_funcao(). Por isso é necessário recriar populate(), ou pelo menos começar uma nova sessão de banco de dados para que a função seja compilada novamente. Outra forma de evitar este problema é utilizar CREATE OR REPLACE FUNCTION ao atualizar a definição de minha_funcao (quando a função é "substituída" o OID não muda).

Tipos registro

```
nome RECORD;
```

As variáveis registro são semelhantes às variáveis tipo-linha, mas não possuem uma estrutura pré-definida. Assumem a estrutura da linha para a qual são atribuídas pelo comando SELECT ou FOR. A subestrutura da variável registro pode mudar toda vez que é usada em uma atribuição. Como consequência, antes de ser utilizada em uma atribuição a variável registro não possui subestrutura, e qualquer tentativa de acessar um de seus campos produz um erro em tempo de execução.

Deve ser observado que RECORD não é um tipo de dado real, mas somente um guardador de lugar. Deve-se ter em mente, também, que declarar uma função do PL/pgSQL como retornando o tipo record não é exatamente o mesmo conceito de variável registro, embora a função possa utilizar uma variável registro para armazenar seu resultado. Nos dois casos a verdadeira estrutura da linha é desconhecida quando a função é escrita, mas na função que retorna o tipo record a estrutura verdadeira é determinada quando o comando que faz a chamada é analisado, enquanto uma variável registro pode mudar a sua estrutura de linha em tempo de execução.

RENAME

```
RENAME nome_antigo TO novo_nome;
```

O nome de uma variável, registro ou linha pode ser mudado através da instrução RENAME. A utilidade principal é quando NEW ou OLD devem ser referenciados por outro nome dentro da função de gatilho. Consulte também ALIAS.

Exemplos:

```
RENAME id TO id_usuario;  
RENAME esta_variável TO aquela_variável;
```

Nota: RENAME parece estar com problemas desde o PostgreSQL 7.3. A correção possui baixa prioridade, porque o ALIAS cobre a maior parte dos usos práticos do RENAME.

Expressões

As duas funções abaixo são diferentes:

```
CREATE FUNCTION logfunc1(logtxt text) RETURNS timestamp AS $$  
BEGIN  
    INSERT INTO logtable VALUES (logtxt, 'now');  
    RETURN 'now';  
END;  
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION logfunc2(logtxt text) RETURNS timestamp AS $$  
DECLARE  
    curtime timestamp;  
BEGIN  
    curtime := 'now';  
    INSERT INTO logtable VALUES (logtxt, curtime);  
    RETURN curtime;  
END;  
$$ LANGUAGE plpgsql;
```

Instruções básicas

Esta seção e as seguintes descrevem todos os tipos de instruções compreendidas explicitamente pela PL/pgSQL. Tudo que não é reconhecido como um destes tipos de instrução é assumido como sendo um comando SQL, e enviado para ser executado pela máquina de banco de dados principal (após a substituição das variáveis do PL/pgSQL na instrução). Desta maneira, por exemplo, os comandos SQL INSERT, UPDATE e DELETE podem ser considerados como sendo instruções da linguagem PL/pgSQL, mas não são listados aqui.

Atribuições

A atribuição de um valor a uma variável, ou a um campo de linha ou de registro, é escrita da seguinte maneira:

```
identificador := expressão;
```

Conforme explicado anteriormente, a expressão nesta instrução é avaliada através de um comando SELECT do SQL enviado para a máquina de banco de dados principal. A expressão deve produzir um único valor.

Se o tipo de dado do resultado da expressão não corresponder ao tipo de dado da variável, ou se a variável possuir um tipo/precisão específico (como char(20)), o valor do resultado será convertido implicitamente pelo interpretador do PL/pgSQL, utilizando a função de saída do tipo do resultado e a função de entrada do tipo da variável. Deve ser observado que este procedimento pode ocasionar erros em tempo de execução gerados pela função de entrada, se a forma cadeia de caracteres do valor do resultado não puder ser aceita pela função de entrada.

Exemplos:

```
id_usuario := 20;  
taxa := subtotal * 0.06;
```

SELECT INTO

O resultado de um comando SELECT que retorna várias colunas (mas apenas uma linha) pode ser atribuído a uma variável registro, a uma variável tipo-linha, ou a uma lista de variáveis escalares. É feito através de

```
SELECT INTO destino expressões_de_seleção FROM ...;
```

Exemplo:

```
SELECT INTO x SUM(quantidade) AS total FROM produtos;
```

Observe que a variável x retornará o valor da consulta SQL.

onde **destino** pode ser uma variável registro, uma variável linha, ou uma lista separada por vírgulas de variáveis simples e campos de registro/linha. A expressões_de_seleção e o restante do comando são os mesmos que no SQL comum.

Deve ser observado que é bem diferente da interpretação normal de SELECT INTO feita pelo PostgreSQL, onde o destino de INTO é uma nova tabela criada. Se for desejado criar uma tabela dentro de uma função PL/pgSQL a partir do resultado do SELECT, deve ser utilizada a sintaxe CREATE TABLE ... AS SELECT.

Se for utilizado como destino uma linha ou uma lista de variáveis, os valores selecionados devem corresponder exatamente à estrutura do destino, senão ocorre um erro em tempo de execução. Quando o destino é uma variável registro, esta se autoconfigura automaticamente para o tipo linha das colunas do resultado da consulta.

Exceto pela cláusula INTO, a instrução SELECT é idêntica ao comando SELECT normal do SQL, podendo utilizar todos os seus recursos.

A cláusula INTO pode aparecer em praticamente todos os lugares na instrução SELECT. Habitualmente é escrita logo após o SELECT, conforme mostrado acima, ou logo antes do FROM — ou seja, logo antes ou logo após a lista de expressões_de_seleção.

Se a consulta não retornar nenhuma linha, são atribuídos valores nulos aos destinos. Se a consulta retornar várias linhas, a primeira linha é atribuída aos destinos e as demais são desprezadas; deve ser observado que "a primeira linha" não é bem definida a não ser que seja utilizado ORDER BY.

A variável especial FOUND pode ser verificada imediatamente após a instrução SELECT INTO para determinar se a atribuição foi bem-sucedida, ou seja, foi retornada pelo menos uma linha pela consulta. (consulte a [Seção 35.6.6](#)). Por exemplo:

```
SELECT INTO meu_registro * FROM emp WHERE nome_emp = meu_nome;
IF NOT FOUND THEN
    RAISE EXCEPTION 'não foi encontrado o empregado %!', meu_nome;
END IF;
```

Para testar se o resultado do registro/linha é nulo, pode ser utilizada a condição IS NULL. Entretanto, não existe maneira de saber se foram desprezadas linhas adicionais. A seguir está mostrado um exemplo que trata o caso onde não foi retornada nenhuma linha:

```
DECLARE
    registro_usuario RECORD;
BEGIN
    SELECT INTO registro_usuario * FROM usuarios WHERE id_usuario=3;
```

```
IF registro_usuario.pagina_web IS NULL THEN
    -- o usuario não informou a página na web, retornar "http://"
    RETURN 'http://';
END IF;
END;
```

Exemplo Prático

Quero pegar o retorno de duas consulta, somar e receber este resultado no retorno.

```
CREATE OR REPLACE FUNCTION somar(quant integer) RETURNS BIGINT AS $$
declare
    x bigint;
    y bigint;
begin
    SELECT INTO x sum(codigo) from cliente where codigo <= quant;
    IF NOT FOUND THEN
        -- A frase abaixo precisa ser delimitada por duas aspas simples,
        -- caso o delimitador da função seja aspas simples
        RAISE EXCEPTION 'Não foi encontrado o código %!', codigo;
    END IF;
    SELECT INTO y sum(codigo) from cliente where codigo < quant;
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Não foi encontrado o código %!', codigo;
    END IF;
    --z := cast(x as integer) + cast(y as integer);
    RETURN x+y;
end;
$$ LANGUAGE PLPGSQL;
```

```
SELECT somar();
```

Execução de expressão ou de consulta sem resultado

Instrução PERFORM

Algumas vezes se deseja **avaliar uma expressão ou comando e desprezar o resultado** (normalmente quando está sendo chamada uma função que produz efeitos colaterais, mas não possui nenhum valor de resultado útil). **Para se fazer isto no PL/pgSQL é utilizada a instrução PERFORM:**

```
PERFORM comando;
```

Esta instrução executa o comando e despreza o resultado. A instrução deve ser escrita da mesma maneira que se escreve um comando SELECT do SQL, mas com a palavra chave inicial SELECT substituída por PERFORM. As variáveis da linguagem PL/pgSQL são substituídas no comando da maneira usual. Além disso, a variável especial FOUND é definida como

verdade se a instrução produzir pelo menos uma linha, ou falso se não produzir nenhuma linha.

Nota: Poderia se esperar que SELECT sem a cláusula INTO produzisse o mesmo resultado, mas atualmente a única forma aceita para isto ser feito é através do PERFORM.

Exemplo:

```
PERFORM create_mv('cs_session_page_requests_mv', my_query);
```

Não fazer nada

Algumas vezes uma instrução guardadora de lugar que não faz nada é útil. Por exemplo, pode indicar que uma ramificação da cadeia if/then/else está deliberadamente vazia. Para esta finalidade deve ser utilizada a instrução NULL:

```
NULL;
```

Por exemplo, os dois fragmentos de código a seguir são equivalentes:

```
BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        NULL; -- ignorar o erro
END;

BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN -- ignorar o erro
END;
```

Qual dos dois escolher é uma questão de gosto.

Nota: Na linguagem PL/SQL do Oracle não é permitida instrução vazia e, portanto, a instrução NULL é *requerida* em situações como esta. Mas a linguagem PL/pgSQL permite que simplesmente não se escreva nada.

Execução de comandos dinâmicos

As vezes é necessário gerar comandos dinâmicos dentro da função PL/pgSQL, ou seja, comandos que envolvem tabelas diferentes ou tipos de dado diferentes cada vez que são executados. A tentativa normal do PL/pgSQL de colocar planos para os comandos no *cache* não funciona neste cenário. A instrução EXECUTE é fornecida para tratar este tipo de problema:

```
EXECUTE cadeia_de_caracteres_do_comando;
```

onde cadeia_de_caracteres_do_comando é uma expressão que produz uma cadeia de caracteres (do tipo text) contendo o comando a ser executado. **A cadeia de caracteres é enviada literalmente para a máquina SQL.**

Em particular, deve-se observar que não é feita a substituição das variáveis do PL/pgSQL na cadeia de caracteres do comando. Os valores das variáveis devem ser inseridos na cadeia de caracteres do

comando quando esta é construída.

Diferentemente de todos os outros comandos do PL/pgSQL, o comando executado pela instrução EXECUTE não é preparado e salvo apenas uma vez por todo o tempo de duração da sessão. Em vez disso, o comando é preparado cada vez que a instrução é executada. A cadeia de caracteres do comando pode ser criada dinamicamente dentro da função para realizar ações em tabelas e colunas diferentes.

Os resultados dos comandos SELECT são desprezados pelo EXECUTE e, atualmente, o SELECT INTO não é suportado pelo EXECUTE. Portanto não há maneira de extrair o resultado de um comando SELECT criado dinamicamente utilizando o comando EXECUTE puro. Entretanto, há duas outras maneiras disto ser feito: uma é utilizando o laço FOR-IN-EXECUTE descrito na [Seção 35.7.4](#), e a outra é utilizando um cursor com OPEN-FOR-EXECUTE, conforme descrito na [Seção 35.8.2](#).

Quando se trabalha com comandos dinâmicos, muitas vezes é necessário tratar o escape dos apóstrofes. O método recomendado para delimitar texto fixo no corpo da função é utilizar o cifrão (Caso exista código legado que não utiliza a delimitação por cifrão por favor consulte a visão geral na [Seção 35.2.1](#), que pode ajudar a reduzir o esforço para converter este código em um esquema mais razoável).

Os valores dinâmicos a serem inseridos nos comandos construídos requerem um tratamento especial, uma vez que estes também podem conter apóstrofes ou aspas. Um exemplo (**assumindo que está sendo utilizada a delimitação por cifrão para a função como um todo e, portanto, os apóstrofes não precisam ser duplicados**) é:

```
EXECUTE 'UPDATE tbl SET '
      || quote_ident(nome_da_coluna)
      || ' = '
      || quote_literal(novo_valor)
      || ' WHERE key = '
      || quote_literal(valor_chave);
```

Este exemplo mostra o uso das funções `quote_ident(text)` e `quote_literal(text)`. **Por motivo de segurança, as variáveis contendo identificadores de coluna e de tabela devem ser passadas para a função `quote_ident`.** As variáveis contendo valores que devem se tornar literais cadeia de caracteres no comando construído devem ser passadas para função `quote_literal`. Estas duas funções executam os passos apropriados para retornar o texto de entrada envolto por aspas ou apóstrofes, respectivamente, com todos os caracteres especiais presentes devidamente colocados em seqüências de escape.

Deve ser observado que **a delimitação por cifrão somente é útil para delimitar texto fixo**. Seria uma péssima idéia tentar codificar o exemplo acima na forma

```
EXECUTE 'UPDATE tbl SET '
      || quote_ident(nome_da_coluna)
      || ' = $$'
      || novo_valor
      || '$$ WHERE key = '
      || quote_literal(valor_chave);
```

porque não funcionaria se o conteúdo de `novo_valor` tivesse \$\$. A mesma objeção se aplica a qualquer outra delimitação por cifrão escolhida. Portanto, **para delimitar texto que não é previamente conhecido deve ser utilizada a função `quote_literal`**.

Pode ser visto no [Exemplo 35-8](#), onde é construído e executado um comando CREATE FUNCTION para definir uma nova função, um caso muito maior de comando dinâmico e EXECUTE.

Obtenção do status do resultado

Existem diversas maneiras de determinar o efeito de um comando. O primeiro método é utilizar o comando **GET DIAGNOSTICS**, que possui a forma:

```
GET DIAGNOSTICS variável = item [ , ... ] ;
```

Este comando permite obter os indicadores de status do sistema. Cada item é uma palavra chave que identifica o valor de estado a ser atribuído a variável especificada (que deve ser do tipo de dado correto para poder receber o valor). Os itens de status disponíveis atualmente são ROW_COUNT, o número de linhas processadas pelo último comando SQL enviado para a máquina SQL, e RESULT_OID, o OID da última linha inserida pelo comando SQL mais recente. Deve ser observado que RESULT_OID só tem utilidade após um comando INSERT.

Exemplo:

```
GET DIAGNOSTICS variável_inteira = ROW_COUNT;
create or replace function diag() returns integer as '
declare
variavel_inteira integer;
begin
GET DIAGNOSTICS variavel_inteira = ROW_COUNT;
return variavel_inteira;
end;
' language plpgsql;
```

O segundo método para determinar os efeitos de um comando é verificar a variável especial FOUND, que é do tipo boolean. A variável FOUND é iniciada como falso dentro de cada chamada de função PL/pgSQL. É definida por cada um dos seguintes tipos de instrução:

- A instrução SELECT INTO define FOUND como verdade quando retorna uma linha, e como falso quando não retorna nenhuma linha.
- A instrução PERFORM define FOUND como verdade quando produz (e despreza) uma linha, e como falso quando não produz nenhuma linha.
- As instruções UPDATE, INSERT e DELETE definem FOUND como verdade quando pelo menos uma linha é afetada, e como falso quando nenhuma linha é afetada.
- A instrução FETCH define FOUND como verdade quando retorna uma linha, e como falso quando não retorna nenhuma linha.
- A instrução FOR define FOUND como verdade quando interage uma ou mais vezes, senão define como falso. Isto se aplica a todas três variantes da instrução FOR (laços FOR inteiros, laços FOR em conjuntos de registros, e laços FOR em conjuntos de registros dinâmicos). A variável FOUND é definida desta maneira ao sair do laço FOR: dentro da execução do laço a variável FOUND não é modificada pela instrução FOR, embora possa ser modificada pela execução de outras instruções dentro do corpo do laço.

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

Estruturas de controle

As estruturas de controle provavelmente são a parte mais útil (e mais importante) da linguagem PL/pgSQL. Com as estruturas de controle do PL/pgSQL os dados do PostgreSQL podem ser manipulados de uma forma muito flexível e poderosa.

Retorno de uma função

Estão disponíveis dois comandos que permitem retornar dados de uma função: RETURN e RETURN NEXT.

RETURN

```
RETURN expressão;
```

O comando RETURN com uma expressão termina a função e retorna o valor da expressão para quem chama. Esta forma é utilizada pelas funções do PL/pgSQL que não retornam conjunto.

Qualquer expressão pode ser utilizada para retornar um tipo escalar. O resultado da expressão é automaticamente convertido no tipo de retorno da função conforme descrito nas atribuições. Para retornar um valor composto (linha), deve ser escrita uma variável registro ou linha como a expressão.

O valor retornado pela função não pode ser deixado indefinido. Se o controle atingir o final do bloco de nível mais alto da função sem atingir uma instrução RETURN, ocorrerá um erro em tempo de execução.

Se a função for declarada como retornando void, ainda assim deve ser especificada uma instrução RETURN; mas neste caso a expressão após o comando RETURN é opcional, sendo ignorada caso esteja presente.

RETURN NEXT

```
RETURN NEXT expressão;
```

Quando uma função PL/pgSQL é declarada como retornando SETOF algum_tipo, o procedimento a ser seguido é um pouco diferente. Neste caso, os itens individuais a serem retornados são especificados em comandos RETURN NEXT, e um comando RETURN final, sem nenhum argumento, é utilizado para indicar que a função chegou ao fim de sua execução. O comando RETURN NEXT pode ser utilizado tanto com tipos de dado escalares quanto compostos; no último caso toda uma "tabela" de resultados é retornada.

As funções que utilizam RETURN NEXT devem ser chamadas da seguinte maneira:

```
SELECT * FROM alguma_função();
```

Ou seja, a função deve ser utilizada como uma fonte de tabela na cláusula FROM.

Na verdade, o comando RETURN NEXT não faz o controle sair da função: simplesmente salva o valor da expressão. Em seguida, a execução continua na próxima instrução da função PL/pgSQL. O

conjunto de resultados é construído se executando comandos RETURN NEXT sucessivos. O RETURN final, que não deve possuir argumentos, faz o controle sair da função.

Nota: A implementação atual de RETURN NEXT para o PL/pgSQL armazena todo o conjunto de resultados antes de retornar da função, conforme foi mostrado acima. Isto significa que, se a função PL/pgSQL produzir um conjunto de resultados muito grande, o desempenho será ruim: os dados serão escritos em disco para evitar exaurir a memória, mas a função não retornará antes que todo o conjunto de resultados tenha sido gerado. Uma versão futura do PL/pgSQL deverá permitir aos usuários definirem funções que retornam conjuntos que não tenham esta limitação. Atualmente, o ponto onde os dados começam a ser escritos em disco é controlado pela variável de configuração work_mem. Os administradores que possuem memória suficiente para armazenar conjuntos de resultados maiores, devem considerar o aumento deste parâmetro.

Condicionais

As instruções IF permitem executar os comandos com base em certas condições. A linguagem PL/pgSQL possui cinco formas de IF:

- IF ... THEN
- IF ... THEN ... ELSE
- IF ... THEN ... ELSE IF
- IF ... THEN ... ELSIF ... THEN ... ELSE
- IF ... THEN ... ELSEIF ... THEN ... ELSE

IF-THEN

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

As instruções IF-THEN são a forma mais simples de IF. As instruções entre o THEN e o END IF são executadas se a condição for verdade. Senão, são saltadas.

Exemplo:

```
IF v_id_usuario <> 0 THEN
    UPDATE usuarios SET email = v_email WHERE id_usuario = v_id_usuario;
END IF;
```

Outro exemplo

```
CREATE FUNCTION calclonger(text,text) RETURNS int4 AS
'
DECLARE
```

```

in_one ALIAS FOR $1;
in_two ALIAS FOR $2;
len_one int4;
len_two int4;
result int4;
BEGIN
    len_one := (SELECT LENGTH(in_one));
    len_two := (SELECT LENGTH(in_two));

    IF    len_one > len_two THEN
        RETURN len_one;
    ELSE
        RETURN len_two;
    END IF;
END;
' LANGUAGE 'plpgsql';

IF    len_one > 20 AND len_one < 40 THEN
    RETURN len_one;
ELSE
    RETURN len_two;
END IF;

```

IF-THEN-ELSE

```

IF expressão_booleana THEN
    instruções
ELSE
    instruções
END IF;

```

As instruções IF-THEN-ELSE ampliam o IF-THEN permitindo especificar um conjunto alternativo de instruções a serem executadas se a condição for avaliada como falsa.

Exemplos:

```

IF id_pais IS NULL OR id_pais = ''
THEN
    RETURN nome_completo;
ELSE
    RETURN hp_true_filename(id_pais) || '/' || nome_completo;
END IF;

IF v_contador > 0 THEN
    INSERT INTO contador_de_usuários (contador) VALUES (v_contador);
    RETURN 't';
ELSE
    RETURN 'f';
END IF;

```

IF-THEN-ELSE IF

As instruções IF podem ser aninhadas, como no seguinte exemplo:

```
IF linha_demo.sexo = 'm' THEN
    sexo_extenso := 'masculino';
ELSE
    IF linha_demo.sexo = 'f' THEN
        sexo_extenso := 'feminino';
    END IF;
END IF;
```

Na verdade, quando esta forma é utilizada uma instrução IF está sendo aninhada dentro da parte ELSE da instrução IF externa. Portanto, há necessidade de uma instrução END IF para cada IF aninhado, mais um para o IF-ELSE pai. Embora funcione, cresce de forma tediosa quando existem muitas alternativas a serem verificadas. Por isso existe a próxima forma.

IF-THEN-ELSIF-ELSE

```
IF expressão_booleana THEN
    instruções
[ ELSEIF expressão_booleana THEN
    instruções
[ ELSEIF expressão_booleana THEN
    instruções
    ...]]
[ ELSE
    instruções ]
END IF;
```

A instrução IF-THEN-ELSIF-ELSE fornece um método mais conveniente para verificar muitas alternativas em uma instrução. Formalmente equivale aos comandos IF-THEN-ELSE-IF-THEN aninhados, mas somente necessita de um END IF.

Abaixo segue um exemplo:

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

IF-THEN-ELSEIF-ELSE

ELSEIF é um aliás para ELSIF.

Laços simples

Com as instruções LOOP, EXIT, WHILE e FOR pode-se fazer uma função PL/pgSQL repetir uma série de comandos.

LOOP

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

```
END LOOP;
```

A instrução LOOP define um laço incondicional, repetido indefinidamente até ser terminado por uma instrução EXIT ou RETURN. Nos laços aninhados pode ser utilizado um rótulo opcional na instrução EXIT para especificar o nível de aninhamento que deve ser terminado.

EXIT

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

Se não for especificado nenhum rótulo, o laço mais interno será terminado, e a instrução após o END LOOP será executada a seguir. Se o rótulo for fornecido, deverá ser o rótulo do nível corrente, ou o rótulo de algum nível externo ao laço ou bloco aninhado. Nesse momento o laço ou bloco especificado será terminado, e o controle continuará na instrução após o END correspondente ao laço ou bloco.

Quando WHEN está presente, a saída do laço ocorre somente se a condição especificada for verdadeira, senão o controle passa para a instrução após o EXIT.

Pode ser utilizado EXIT para causar uma saída prematura de qualquer tipo de laço; não está limitado aos laços incondicionais.

Exemplos:

```
LOOP
    -- algum processamento
    IF contador > 0 THEN
        EXIT; -- sair do laço
    END IF;
END LOOP;

LOOP
    -- algum processamento
    EXIT WHEN contador > 0; -- mesmo resultado do exemplo acima
END LOOP;

BEGIN
    -- algum processamento
    IF estoque > 100000 THEN
        EXIT; -- causa a saída do bloco BEGIN
    END IF;
END;
```

WHILE

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

A instrução WHILE repete uma seqüência de instruções enquanto a expressão de condição for avaliada como verdade. A condição é verificada logo antes de cada entrada no corpo do laço.

Por exemplo:

```
WHILE quantia_devida > 0 AND saldo_do_certificado_de_bonus > 0 LOOP
    -- algum processamento
END LOOP;
```

```
WHILE NOT expressão_booleana LOOP
    -- algum processamento
END LOOP;
```

```
CREATE FUNCTION countc (text, text) RETURNS int4 AS '
    DECLARE
        intext ALIAS FOR $1;
        inchar ALIAS FOR $2;
        len    int4;
        result int4;
        i      int4;
        tmp    char;
    BEGIN
        len := length(intext);
        i   := 1;
        result := 0;
        WHILE i <= len LOOP
            tmp := substr(intext, i, 1);
            IF   tmp = inchar THEN
                result := result + 1;
            END IF;
            i:= i+1;
        END LOOP;
        RETURN result;
    END;
' LANGUAGE 'plpgsql';
```

FOR (variação inteira)

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

Esta forma do FOR cria um laço que interage num intervalo de valores inteiros. A variável nome é definida automaticamente como sendo do tipo integer, e somente existe dentro do laço. As duas expressões que fornecem o limite inferior e superior do intervalo são avaliadas somente uma vez, ao entrar no laço. Normalmente o passo da interação é 1, mas quando REVERSE é especificado se torna -1.

Alguns exemplos de laços FOR inteiros:

```
FOR i IN 1..10 LOOP
    -- algum processamento
```

```
        RAISE NOTICE 'i é %', i;
END LOOP;
```

```
FOR i IN REVERSE 10..1 LOOP
    -- algum processamento
END LOOP;
```

Se o limite inferior for maior do que o limite superior (ou menor, no caso do REVERSE), o corpo do laço não é executado nenhuma vez. Nenhum erro é gerado.

Laço através do resultado da consulta

Utilizando um tipo diferente de laço FOR, é possível interagir através do resultado de uma consulta e manipular os dados. A sintaxe é:

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

Cada linha de resultado do comando (que deve ser um SELECT) é atribuída, sucessivamente, à variável registro ou linha, e o corpo do laço é executado uma vez para cada linha. Abaixo segue um exemplo:

```
CREATE FUNCTION cs_refresh_mviews() RETURNS integer AS $$
DECLARE
    mviews RECORD;
BEGIN
    PERFORM cs_log('Atualização das visões materializadas...');

    FOR mviews IN SELECT * FROM cs_materialized_views ORDER BY sort_key LOOP

        -- Agora "mviews" possui um registro de cs_materialized_views

        PERFORM cs_log('Atualizando a visão materializada ' ||
quote_ident(mviews.mv_name) || ' ...');
        EXECUTE 'TRUNCATE TABLE ' || quote_ident(mviews.mv_name);
        EXECUTE 'INSERT INTO ' || quote_ident(mviews.mv_name) || ' ' ||
mviews.mv_query;
    END LOOP;

    PERFORM cs_log('Fim da atualização das visões materializadas.');
```

```
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

Se o laço for terminado por uma instrução EXIT, o último valor de linha atribuído ainda é acessível após o laço.

A instrução FOR-IN-EXECUTE é outra forma de interagir sobre linhas:

```
[<<rótulo>>]
FOR registro_ou_linha IN EXECUTE texto_da_expressão LOOP
    instruções
END LOOP;
```

Esta forma é semelhante à anterior, exceto que o código fonte da instrução SELECT é especificado como uma expressão cadeia de caracteres, que é avaliada e replanejada a cada entrada no laço FOR. Isto permite ao programador escolher entre a velocidade da consulta pré-planejada e a flexibilidade da consulta dinâmica, da mesma maneira que na instrução EXECUTE pura.

Nota: Atualmente o analisador da linguagem PL/pgSQL faz distinção entre os dois tipos de laços FOR (inteiro e resultado de consulta), verificando se aparece .. fora de parênteses entre IN e LOOP. Se não for encontrado .., então o laço é assumido como sendo um laço sobre linhas. Se .. for escrito de forma errada, pode causar uma reclamação informando que "a variável do laço, para laço sobre linhas, deve ser uma variável registro ou linha", em vez de um simples erro de sintaxe como poderia se esperar.

```
CREATE FUNCTION countc(text, text, int4, int4) RETURNS int4 AS '
DECLARE
    intext      ALIAS FOR $1;
    inchar      ALIAS FOR $2;
    startpos    ALIAS FOR $3;
    endpos      ALIAS FOR $4;
    tmp         text;
    i           int4;
    len         int4;
    result      int4;
BEGIN
    result = 0;
    len := LENGTH(intext);
    FOR i IN startpos..endpos LOOP
        tmp := substr(intext, i, 1);
        IF tmp = inchar THEN
            result := result + 1;
        END IF;
    END LOOP;
    RETURN result;
END;
' LANGUAGE 'plpgsql';
```

Escrever a função anterior sem o FOR, com LOOP/EXIT

```
CREATE FUNCTION countc(text, text, int4, int4) RETURNS int4 AS '
DECLARE
    intext      ALIAS FOR $1;
    inchar      ALIAS FOR $2;
    startpos    ALIAS FOR $3;
    endpos      ALIAS FOR $4;
    i           int4;
    tmp         text;
    len         int4;
    result      int4;
```

```

BEGIN
    result = 0;
    i := startpos;
    len := LENGTH(intext);
    LOOP
        IF    i <= endpos AND i <= len THEN
            tmp := substr(intext, i, 1);
            IF    tmp = inchar THEN
                result := result + 1;
            END IF;
            i := i + 1;
        ELSE
            EXIT;
        END IF;
    END LOOP;
    RETURN result;
END;
' LANGUAGE 'plpgsql';

```

Sobrecarga de Funções

```

CREATE TABLE employees(id serial, name varchar(50), room int4, salary int4);
INSERT INTO employees (name, room, salary) VALUES ('Paul', 1, 3000);
INSERT INTO employees (name, room, salary) VALUES ('Josef', 1, 2945);
INSERT INTO employees (name, room, salary) VALUES ('Linda', 2, 3276);
INSERT INTO employees (name, room, salary) VALUES ('Carla', 1, 1200);
INSERT INTO employees (name, room, salary) VALUES ('Hillary', 2, 4210);
INSERT INTO employees (name, room, salary) VALUES ('Alice', 3, 1982);
INSERT INTO employees (name, room, salary) VALUES ('Hugo', 4, 1982);

```

```

CREATE FUNCTION insertupdate(text, int4) RETURNS bool AS '
DECLARE
    intext ALIAS FOR $1;
    newsal ALIAS FOR $2;
    checkit record;
BEGIN
    SELECT INTO checkit * FROM employees
        WHERE name=intext;
    IF NOT FOUND THEN
        INSERT INTO employees(name, room, salary)
            VALUES(intext,"1",newsal);
        RETURN "t";
    ELSE
        UPDATE employees SET
            salary=newsal, room=checkit.room
        WHERE name=intext;
        RETURN "f";
    END IF;

```



```

        RETURN 't';
    END;
' LANGUAGE 'plpgsql';

SELECT insertupdate('Alf',700);

SELECT * FROM employees WHERE name='Alf';

SELECT insertupdate('Alf',1250);

SELECT * FROM employees WHERE name='Alf';

```

Trabalhando com SELECT e LOOP

```

CREATE FUNCTION countsel(text) RETURNS int4 AS '
    DECLARE
        inchar ALIAS FOR $1;
        colval record;
        tmp    text;
        result int4;
    BEGIN
        result = 0;
        FOR colval IN SELECT name FROM employees LOOP
            tmp := substr(colval.name, 1, 1);
            IF   tmp = inchar THEN
                result := result + 1;
            END IF;
        END LOOP;
        RETURN result;
    END;
' LANGUAGE 'plpgsql';

```

Captura de erros

Por padrão, qualquer erro que ocorra em uma função PL/pgSQL interrompe a execução da função, e também da transação envoltória. É possível capturar e se recuperar de erros utilizando um bloco BEGIN com a cláusula EXCEPTION. A sintaxe é uma extensão da sintaxe normal do bloco BEGIN:

```

[ <<rótulo>> ]
[ DECLARE
    declarações ]
BEGIN
    instruções
EXCEPTION
    WHEN condição [ OR condição ... ] THEN
        instruções_do_tratador
    [ WHEN condição [ OR condição ... ] THEN
        instruções_do_tratador

```

```
... ]
END;
```

Caso não ocorra nenhum erro, esta forma do bloco simplesmente executa todas as instruções, e depois o controle passa para a instrução seguinte ao END. Mas se acontecer algum erro dentro de instruções, o processamento das instruções é abandonado e o controle passa para a lista de EXCEPTION. É feita a procura na lista da primeira condição correspondendo ao erro encontrado. Se for encontrada uma correspondência, as instruções_do_tratador correspondentes são executadas, e o controle passa para a instrução seguinte ao END. Se não for encontrada nenhuma correspondência, o erro se propaga para fora como se a cláusula EXCEPTION não existisse: o erro pode ser capturado por um bloco envoltório contendo EXCEPTION e, se não houver nenhum, o processamento da função é interrompido.

O nome da condição pode ser qualquer um dos mostrados no [Apêndice A](#). Um nome de categoria corresponde a qualquer erro desta categoria. O nome de condição especial OTHERS corresponde a qualquer erro, exceto QUERY_CANCELED (É possível, mas geralmente não aconselhável, capturar QUERY_CANCELED por nome). Não há diferença entre letras maiúsculas e minúsculas nos nomes das condições.

Caso ocorra um novo erro dentro das instruções_do_tratador selecionadas, este não poderá ser capturado por esta cláusula EXCEPTION, mas é propagado para fora. Uma cláusula EXCEPTION envoltória pode capturá-lo.

Quando um erro é capturado pela cláusula EXCEPTION, as variáveis locais da função PL/pgSQL permanecem como estavam quando o erro ocorreu, mas todas as modificações no estado persistente do banco de dados dentro do bloco são desfeitas. Como exemplo, consideremos este fragmento de código:

```
INSERT INTO minha_tabela(nome, sobrenome) VALUES('Tom', 'Jones');
BEGIN
    UPDATE minha_tabela SET nome = 'Joe' WHERE sobrenome = 'Jones';
    x := x + 1;
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'capturado division_by_zero';
        RETURN x;
END;
```

Quando o controle chegar à atribuição de y, vai falhar com um erro de division_by_zero. Este erro será capturado pela cláusula EXCEPTION. O valor retornado na instrução RETURN será o valor de x incrementado, mas os efeitos do comando UPDATE foram desfeitos. Entretanto, o comando INSERT que precede o bloco não é desfeito e, portanto, o resultado final no banco de dados é Tom Jones e não Joe Jones.

Dica: Custa significativamente mais entrar e sair de um bloco que contém a cláusula EXCEPTION que de um bloco que não contém esta cláusula. Portanto, a cláusula EXCEPTION só deve ser utilizada quando for necessária.

```
CREATE FUNCTION calcsun(int4, int4) RETURNS int4 AS '
DECLARE
    lower ALIAS FOR $1;
    higher ALIAS FOR $2;
```

```

lowres int4;
lowtmp int4;
highres int4;
result int4;
BEGIN
  IF (lower < 1) OR (higher < 1) THEN
    RAISE EXCEPTION "both param. have to be > 0";
  ELSE
    IF (lower <= higher) THEN
      lowtmp := lower - 1;
      lowres := (lowtmp+1)*lowtmp/2;
      highres := (higher+1)*higher/2;
      result := highres-lowres;
    ELSE
      RAISE EXCEPTION "The first value (%) has to be higher than the second
value (%)", higher, lower;
    END IF;
  END IF;
  RETURN result;
END;
' LANGUAGE 'plpgsql';

```

Cursorres

Em vez de executar toda a consulta de uma vez, é possível definir um *cursor* encapsulando a consulta e, depois, ler umas poucas linhas do resultado da consulta de cada vez. Um dos motivos de se fazer desta maneira, é para evitar o uso excessivo de memória quando o resultado contém muitas linhas (Entretanto, normalmente não há necessidade dos usuários da linguagem PL/pgSQL se preocuparem com isto, uma vez que os laços FOR utilizam internamente um cursor para evitar problemas de memória, automaticamente). Uma utilização mais interessante é retornar a referência a um cursor criado pela função, permitindo a quem chamou ler as linhas. Esta forma proporciona uma maneira eficiente para a função retornar conjuntos grandes de linhas.

Declaração de variável cursor

Todos os acessos aos cursores na linguagem PL/pgSQL são feitos através de variáveis cursor, que sempre são do tipo de dado especial *refcursor*. Uma forma de criar uma variável cursor é simplesmente declará-la como sendo do tipo *refcursor*. Outra forma é utilizar a sintaxe de declaração de cursor, cuja forma geral é:

```
nome CURSOR [ ( argumentos ) ] FOR comando ;
```

(O FOR pode ser substituído por IS para ficar compatível com o Oracle). Os argumentos, quando especificados, são uma lista separada por vírgulas de pares nome tipo_de_dado. Esta lista define nomes a serem substituídos por valores de parâmetros na consulta. Os valores verdadeiros que substituirão estes nomes são especificados posteriormente, quando o cursor for aberto.

Alguns exemplos:

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (chave integer) IS SELECT * FROM tenk1 WHERE unicol = chave;
```

Todas estas três variáveis possuem o tipo de dado `refcursor`, mas a primeira pode ser utilizada em qualquer consulta, enquanto a segunda possui uma consulta totalmente especificada *ligada* à mesma, e a terceira possui uma consulta parametrizada ligada à mesma (O parâmetro `chave` será substituído por um valor inteiro quando o cursor for aberto). A variável `curs1` é dita como *desligada* (`unbound`), uma vez que não está ligada a uma determinada consulta.

Abertura de cursor

Antes do cursor poder ser utilizado para trazer linhas, este deve ser *aberto* (É a ação equivalente ao comando SQL `DECLARE CURSOR`). A linguagem PL/pgSQL possui três formas para a instrução `OPEN`, duas das quais utilizam variáveis cursor desligadas, enquanto a terceira utiliza uma variável cursor ligada.

OPEN FOR SELECT

```
OPEN cursor_desligado FOR SELECT ...;
```

A variável cursor é aberta e recebe a consulta especificada para executar. O cursor não pode estar aberto, e deve ter sido declarado como um cursor desligado, ou seja, simplesmente como uma variável do tipo `refcursor`. O comando `SELECT` é tratado da mesma maneira que nas outras instruções `SELECT` da linguagem PL/pgSQL: Os nomes das variáveis da linguagem PL/pgSQL são substituídos, e o plano de execução é colocado no *cache* para uma possível reutilização.

Exemplo:

```
OPEN curs1 FOR SELECT * FROM foo WHERE chave = minha_chave;
```

OPEN FOR EXECUTE

```
OPEN cursor_desligado FOR EXECUTE cadeia_de_caracteres_da_consulta;
```

A variável cursor é aberta e recebe a consulta especificada para executar. O cursor não pode estar aberto, e deve ter sido declarado como um cursor desligado, ou seja, simplesmente como uma variável do tipo `refcursor`. A consulta é especificada como uma expressão cadeia de caracteres da mesma maneira que no comando `EXECUTE`. Como habitual, esta forma provê flexibilidade e, portanto, a consulta pode variar entre execuções.

Exemplo:

```
OPEN curs1 FOR EXECUTE 'SELECT * FROM ' || quote_ident($1);
```

Abertura de cursor ligado

```
OPEN cursor_ligado [ ( valores_dos_argumentos ) ];
```

Esta forma do OPEN é utilizada para abrir uma variável cursor cuja consulta foi ligada à mesma ao ser declarada. O cursor não pode estar aberto. Deve estar presente uma lista de expressões com os valores reais dos argumentos se, e somente se, o cursor for declarado como recebendo argumentos. Estes valores são substituídos na consulta. O plano de comando do cursor ligado é sempre considerado como passível de ser colocado no *cache*; neste caso não há forma EXECUTE equivalente.

Exemplos:

```
OPEN curs2;  
OPEN curs3(42);
```

Utilização de cursores

Uma vez que o cursor tenha sido aberto, este pode ser manipulado pelas instruções descritas a seguir.

Para começar, não há necessidade destas manipulações estarem na mesma função que abriu o cursor. Pode ser retornado pela função um valor refcursor, e deixar por conta de quem chamou operar o cursor (Internamente, o valor de refcursor é simplesmente uma cadeia de caracteres com o nome do tão falado portal que contém a consulta ativa para o cursor. Este nome pode ser passado, atribuído a outras variáveis refcursor, e por aí em diante, sem perturbar o portal).

Todos os portais são fechados implicitamente ao término da transação. Portanto, o valor de refcursor pode ser utilizado para fazer referência a um cursor aberto até o fim da transação.

FETCH

```
FETCH cursor INTO destino;
```

A instrução FETCH coloca a próxima linha do cursor no destino, que pode ser uma variável linha, uma variável registro, ou uma lista separada por vírgulas de variáveis simples, da mesma maneira que no SELECT INTO. Como no SELECT INTO, pode ser verificada a variável especial FOUND para ver se foi obtida uma linha, ou não.

Exemplos:

```
FETCH curs1 INTO variável_linha;  
FETCH curs2 INTO foo, bar, baz;
```

CLOSE

```
CLOSE cursor;
```

A instrução CLOSE fecha o portal subjacente ao cursor aberto. Pode ser utilizada para liberar recursos antes do fim da transação, ou para liberar a variável cursor para que esta possa ser aberta novamente.

Exemplo:

```
CLOSE curs1;
```

Retornar cursor

As funções PL/pgSQL podem retornar cursores para quem fez a chamada. É útil para retornar várias linhas ou colunas, especialmente em conjuntos de resultados muito grandes. Para ser feito, a função abre o cursor e retorna o nome do cursor para quem chamou (ou simplesmente abre o cursor utilizando o nome do portal especificado por, ou de outra forma conhecido por, quem chamou). Quem chamou poderá então ler as linhas usando o cursor. O cursor pode ser fechado por quem chamou, ou será fechado automaticamente ao término da transação.

O nome do portal utilizado para o cursor pode ser especificado pelo programador ou gerado automaticamente. Para especificar o nome do portal deve-se, simplesmente, atribuir uma cadeia de caracteres à variável `refcursor` antes de abri-la. O valor cadeia de caracteres da variável `refcursor` será utilizado pelo OPEN como o nome do portal subjacente. Entretanto, quando a variável `refcursor` é nula, o OPEN gera automaticamente um nome que não conflita com nenhum portal existente, e atribui este nome à variável `refcursor`.

Nota: Uma variável cursor ligada é inicializada com o valor cadeia de caracteres que representa o seu nome e, portanto, o nome do portal é o mesmo da variável cursor, a menos que o programador mude este nome fazendo uma atribuição antes de abrir o cursor. Porém, uma variável cursor desligada tem inicialmente o valor nulo por padrão e, portanto, recebe um nome único gerado automaticamente, a menos que este seja mudado.

O exemplo a seguir mostra uma maneira de fornecer o nome do cursor por quem chama:

```
CREATE TABLE teste (col text);
INSERT INTO teste VALUES ('123');

CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
BEGIN
    OPEN $1 FOR SELECT col FROM teste;
    RETURN $1;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc('funccursor');

    reffunc
-----
    funccursor
(1 linha)

FETCH ALL IN funccursor;

    col
-----
    123
(1 linha)

COMMIT;
```

O exemplo a seguir usa a geração automática de nome de cursor:

```
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
```

```

        ref refcursor;
BEGIN
    OPEN ref FOR SELECT col FROM teste;
    RETURN ref;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc2();

        reffunc2
-----
<unnamed portal 1>
(1 linha)

FETCH ALL IN "<unnamed cursor 1>";

col
----
123
(1 linha)

COMMIT;
```

Os exemplos a seguir mostram uma maneira de retornar vários cursores de uma única função:

```

CREATE FUNCTION minha_funcao(refcursor, refcursor) RETURNS SETOF refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT * FROM tabela_1;
    RETURN NEXT $1;
    OPEN $2 FOR SELECT * FROM tabela_2;
    RETURN NEXT $2;
    RETURN;
END;
$$ LANGUAGE plpgsql;

-- é necessário estar em uma transação para poder usar cursor
BEGIN;

SELECT * FROM minha_funcao('a', 'b');

FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```

Erros e mensagens

A instrução RAISE é utilizada para gerar mensagens informativas e causar erros.

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

Os níveis possíveis são DEBUG, LOG, INFO, NOTICE, WARNING, e EXCEPTION. O nível EXCEPTION causa um erro (que normalmente interrompe a transação corrente); os outros níveis apenas geram mensagens com diferentes níveis de prioridade. Se as mensagens de uma determinada

prioridade são informadas ao cliente, escritas no *log* do servidor, ou as duas coisas, é controlado pelas variáveis de configuração [log_min_messages](#) e [client_min_messages](#). Para obter informações adicionais deve ser consultada a [Seção 16.4](#).

Dentro da cadeia de caracteres de formatação, o caractere % é substituído pela representação na forma de cadeia de caracteres do próximo argumento opcional. Deve ser escrito %% para produzir um % literal. Deve ser observado que atualmente os argumentos opcionais devem ser variáveis simples, e não expressões, e o formato deve ser um literal cadeia de caracteres simples.

Neste exemplo o valor de `v_job_id` substitui o caractere % na cadeia de caracteres:

```
RAISE NOTICE 'Chamando cs_create_job(%)', v_job_id;
```

Este exemplo interrompe a transação com a mensagem de erro fornecida:

```
RAISE EXCEPTION 'ID inexistente --> %', id_usuario;
```

Atualmente RAISE EXCEPTION sempre gera o mesmo código SQLSTATE, P0001, não importando a mensagem com a qual seja chamado. É possível capturar esta exceção com EXCEPTION ... WHEN RAISE_EXCEPTION THEN ..., mas não há como diferenciar um RAISE de outro.

```
CREATE FUNCTION checksal(text) RETURNS int4 AS '
```

```
  DECLARE
```

```
    inname ALIAS FOR $1;
```

```
    sal    employees%ROWTYPE;
```

```
    myval  employees.salary%TYPE;
```

```
  BEGIN
```

```
    SELECT INTO myval salary
```

```
      FROM employees WHERE name=inname;
```

```
    RETURN myval;
```

```
  END;
```

```
' LANGUAGE 'plpgsql';
```

```
SELECT checksal('Paul');
```

```
select 5/2;
```

```
select timestamp(5000000/2);
```


Tratamento dos apóstrofes

O código da função PL/pgSQL é especificado no comando CREATE FUNCTION como um literal cadeia de caracteres. Se o literal cadeia de caracteres for escrito da maneira usual, que é entre apóstrofes ('), então os apóstrofes dentro do corpo da função devem ser duplicados; da mesma maneira, as contrabarras dentro do corpo da função (\) devem ser duplicadas. Duplicar os apóstrofes é no mínimo entediante, e nos casos mais complicados pode tornar o código difícil de ser compreendido, porque pode-se chegar facilmente a uma situação onde são necessários seis ou mais apóstrofes adjacentes. Por isso, recomenda-se que o corpo da função seja escrito em um literal cadeia de caracteres delimitado por "cifrão" (consulte a [Seção 4.1.2.2](#)) em vez de delimitado por apóstrofes. Na abordagem delimitada por cifrão os apóstrofes nunca são duplicados e, em vez disso, toma-se o cuidado de escolher uma marca diferente para cada nível de aninhamento necessário. Por exemplo, o comando CREATE FUNCTION pode ser escrito da seguinte maneira:

```
CREATE OR REPLACE FUNCTION funcao_teste(integer) RETURNS integer AS $PROC$
    ....
$PROC$ LANGUAGE plpgsql;
```

No corpo da função podem ser utilizados apóstrofes para delimitar cadeias de caracteres simples nos comandos SQL, e \$\$ para delimitar fragmentos de comandos SQL montados como cadeia de caracteres. Se for necessário delimitar um texto contendo \$\$, deve ser utilizado \$Q\$, e assim por diante.

O quadro abaixo mostra o que deve ser feito para escrever o corpo da função entre apóstrofes (sem uso da delimitação por cifrão). Pode ser útil para tornar códigos anteriores à delimitação por cifrão mais fácil de serem compreendidos.

1 apóstrofo

para começar e terminar o corpo da função como, por exemplo:

```
CREATE FUNCTION foo() RETURNS integer AS '
    ....
' LANGUAGE plpgsql;
```

Em todas as ocorrências dentro do corpo da função os apóstrofes *devem* aparecer em pares.

2 apóstrofes

Para literais cadeia de caracteres dentro do corpo da função como, por exemplo:

```
a_output := 'Blah';
SELECT * FROM users WHERE f_nome='foobar';
```

Na abordagem delimitada por cifrão seria escrito apenas

```
a_output := 'Blah';
SELECT * FROM users WHERE f_nome='foobar';
```

que é exatamente o código visto pelo analisador do PL/pgSQL nos dois casos.

4 apóstrofes

Quando é necessário colocar um apóstrofo em uma constante cadeia de caracteres dentro do corpo da função como, por exemplo:

```
a_output := a_output || ' AND nome LIKE '''foobar''' AND xyz'
```

O verdadeiro valor anexado a a_output seria: AND nome LIKE 'foobar' AND xyz.

Na abordagem delimitada por cifrão seria escrito

```
a_output := a_output || $$ AND nome LIKE 'foobar' AND xyz$$
```

tendo-se o cuidado de que todos os delimitadores por cifrão envolvendo este comando não sejam apenas \$\$.

6 apóstrofes

Quando o apóstrofo na cadeia de caracteres dentro do corpo da função está adjacente ao final da constante cadeia de caracteres como, por exemplo:

```
a_output := a_output || ' AND nome LIKE '''foobar''''
```

O valor anexado à a_output seria: AND nome LIKE 'foobar'.

Na abordagem delimitada por cifrão se tornaria

```
a_output := a_output || $$ AND nome LIKE 'foobar'$$
```

10 apóstrofes

Quando é necessário colocar dois apóstrofes em uma constante cadeia de caracteres (que necessita de 8 apóstrofes), e estes dois apóstrofes estão adjacentes ao final da constante cadeia de caracteres (mais 2 apóstrofes). Normalmente isto só é necessário quando são escritas funções que geram outras funções como no [Exemplo 35-8](#). Por exemplo:

```
a_output := a_output || ' if v_ ' ||
referrer_keys.kind || ' like ' ||
referrer_keys.key_string || ' ' ||
then return ' ' || referrer_keys.referrer_type
|| ' '; end if;'
```

O valor de a_output seria então:

```
if v_... like '...' then return '...'; end if;
```

Na abordagem delimitada por cifrão se tornaria

```
a_output := a_output || $$ if v_ $$ || referrer_keys.kind || $$ like '$$
referrer_keys.key_string || $$
then return '$$ || referrer_keys.referrer_type
|| $$'; end if;$$;
```

onde se assume que só é necessário colocar um único apóstrofo em a_output, porque este será delimitado novamente antes de ser utilizado.

Uma outra abordagem é fazer o escape dos apóstrofes no corpo da função utilizando a contrabarra em vez de duplicá-los. Desta forma é escrito \" no lugar de ". Alguns acham esta forma mais fácil, porém outros não concordam.

Sobrecarga de função.

O conceito de sobrecarga de função na PL/pgSQL é o mesmo encontrado nas linguagens de programação orientadas a objeto. Abaixo um exemplo bem simples que ilustra esse conceito.

Imagine uma função soma.

```
Create function soma(num1 integer, num2 integer) returns integer as
$$
Declare
    resultado integer := 0;
Begin
    resultado := num1 + num2;
    return resultado;
End;
$$ language 'plpgsql';
```

Essa é uma função soma que recebe dois números inteiros. Mas e se o usuário passar dois dados do tipo caracter ao invés de números, qual seria o comportamento da minha função? Ai é que entra o conceito de sobrecarga de função. O PostgreSQL me permite ter uma função com o mesmo nome, mas com uma assinatura diferente. A assinatura é formada pelo nome + os parâmetros recebidos pela função. Por isso é que para remover uma função nós temos que utilizar o comando drop function + a assinatura da função e não somente o nome da função.

Para resolvermos o problema acima basta criarmos uma função soma da seguinte forma:

```
Create function soma(num1 text, num2 text) returns char as
$$
Declare
    resultado text;
Begin
    resultado := num1 || num2; --- || é o caracter para concatenação.
    return resultado;
End;
$$ language 'plpgsql';
```

Dessa forma, teríamos duas funções "soma", uma que recebe dois inteiros e devolve o resultado da soma dos dois e outra que recebe dois caracteres e retorna a concatenação desses caracteres. O usuário só precisa saber que para concatenar caracteres ou somar valores basta chamar uma função soma.

Baseado no tipo dos parâmetros passados, o PostgreSQL se encarrega de definir qual a função será utilizada.

Observações: Nas funções acima eu poderia ter suprimido a variável resultado e retornado os valores diretamente chamando "return num1 + num2;". A intenção foi a de reforçar o conceito da declaração de variável.

O exemplo é bem simples o que quero mostrar é o conceito.

Exemplo prático de uso de funções plpgsql para validar CPF e CNPJ:

```
-- *****
-- Função: f_cnpjcpf
-- Objetivo:
-- Validar o número do documento especificado
-- (CNPJ ou CPF) ou não (livre)
-- Argumentos:
-- Pessoa [Jurídica(0),Física(1) ou
-- Livre(2)] (integer), Número com dígitos
-- verificadores e sem pontuação (bpchar)
-- Retorno:
-- -1: Tipo de Documento invalido.
-- -2: Caracter inválido no numero do documento.
-- -3: Numero do Documento invalido.
-- 1: OK (smallint)
-- *****
--
CREATE OR REPLACE FUNCTION f_cnpjcpf (integer,bpchar)
RETURNS integer
AS '
DECLARE

-- Argumentos
-- Tipo de verificacao : 0 (PJ), 1 (PF) e 2 (Livre)
pTipo ALIAS FOR $1;
-- Numero do documento
pNumero ALIAS FOR $2;

-- Variaveis
i INT4; -- Contador
iProd INT4; -- Somatório
iMult INT4; -- Fator
iDigito INT4; -- Digito verificador calculado
sNumero VARCHAR(20); -- numero do docto completo

BEGIN

-- verifica Argumentos validos
IF (pTipo < 0) OR (pTipo > 2) THEN
RETURN -1;
END IF;
```

```
-- se for Livre, nao eh necessario a verificacao
IF pTipo = 2 THEN
    RETURN 1;
END IF;

sNumero := trim(pNumero);
FOR i IN 1..char_length(sNumero) LOOP
    IF position(substring(sNumero, i, 1) in "1234567890") = 0 THEN
        RETURN -2;
    END IF;
END LOOP;
sNumero := '';

-- *****
-- Verifica a validade do CNPJ
-- *****

IF (char_length(trim(pNumero)) = 14) AND (pTipo = 0) THEN

-- primeiro digito
sNumero := substring(pNumero from 1 for 12);
iMult := 2;
iProd := 0;

FOR i IN REVERSE 12..1 LOOP
    iProd := iProd + to_number(substring(sNumero from i for 1), '9') * iMult;
    IF iMult = 9 THEN
        iMult := 2;
    ELSE
        iMult := iMult + 1;
    END IF;
END LOOP;

iDigito := 11 - (iProd % 11);
IF iDigito >= 10 THEN
    iDigito := 0;
END IF;

sNumero := substring(pNumero from 1 for 12) || trim(to_char(iDigito, '9')) || '0';

-- segundo digito
iMult := 2;
iProd := 0;

FOR i IN REVERSE 13..1 LOOP
    iProd := iProd + to_number(substring(sNumero from i for 1), '9') * iMult;
    IF iMult = 9 THEN
        iMult := 2;
    ELSE
        iMult := iMult + 1;
    END IF;
END LOOP;
```

```

ELSE
    iMult := iMult + 1;
END IF;
END LOOP;

iDigito := 11 - (iProd % 11);
IF iDigito >= 10 THEN
    iDigito := 0;
END IF;

sNumero := substring(sNumero from 1 for 13) || trim(to_char(iDigito,"9"));
END IF;

-- *****
-- Verifica a validade do CPF
-- *****

IF (char_length(trim(pNumero)) = 11) AND (pTipo = 1) THEN

-- primeiro digito
    iDigito := 0;
    iProd := 0;
    sNumero := substring(pNumero from 1 for 9);

    FOR i IN 1..9 LOOP
        iProd := iProd + (to_number(substring(sNumero from i for 1),"9") * (11 - i));
    END LOOP;
    iDigito := 11 - (iProd % 11);
    IF (iDigito) >= 10 THEN
        iDigito := 0;
    END IF;
    sNumero := substring(pNumero from 1 for 9) || trim(to_char(iDigito,"9")) || "0";

-- segundo digito
    iProd := 0;
    FOR i IN 1..10 LOOP
        iProd := iProd + (to_number(substring(sNumero from i for 1),"9") * (12 - i));
    END LOOP;
    iDigito := 11 - (iProd % 11);
    IF (iDigito) >= 10 THEN
        iDigito := 0;
    END IF;
    sNumero := substring(sNumero from 1 for 10) || trim(to_char(iDigito,"9"));

END IF;

-- faz a verificacao do digito verificador calculado
IF pNumero = sNumero::bpchar THEN

```

```
RETURN 1;
ELSE
RETURN -3;
END IF;
END;
'LANGUAGE 'plpgsql';
```

Rode este script no seu banco, caso retorne um erro *"ERROR: language 'plpgsql' does not exist"* ou qualquer coisa parecida, é preciso dizer ao banco que esta base deve aceitar funções escritas em plpgsql. O PostgreSQL tem diversas linguagens PL, em uma das minhas colunas, menciona a maioria delas, dêem um olhada.

Bom, para habilitar a base a aceitar o **plpgsql** execute o comando no prompt bash (\$):

```
createlang -U postgres pgplsql nomedabase
```

Em seguida, rode o script novamente.

Para executá-lo, digite no prompt da base (=#):

```
SELECT f_cnpjcpf( 1, '12312312345' );
```

Neste caso retorna um erro (-3) definido com documento inválido na função.

```
SELECT f_cnpjcpf( 2, '12312312345' );
```

Neste caso retorna (1) que significa que a operação foi bem sucedida! Porquê?!

Lembre-se, o argumento Pessoa tipo 2 não faz a validação do documento digitado.

Você também pode utilizar a função **f_cnpjcpf** na validação de um campo, por exemplo:

```
CREATE TABLE cadastro (
  nome    VARCHAR(50) NOT NULL,
  tipopessoa INT2 NOT NULL
           CHECK (tipopessoa IN (0,1)),
  cpfcnpj CHAR(20) NOT NULL
           CHECK (f_cnpjcpf(tipopessoa, cpfcnpj)=1)
);
```

Ao tentar inserir um registro com um número de **cpf** ou **cnpj** inválido, volta um erro retornado pela *Check Constraint* responsável pela validação. Tente:

```
INSERT INTO cadastro (nome, tipopessoa, cpfcnpj)
VALUES ( 'Juliano S. Ignacio', 1, '12312312345');
```

Coloque o seu **nome** e **cpf** e verá que o registro será inserido.

Outra aplicação é na validação de **cnpj** (por exemplo) de uma tabela importada de uma origem qualquer:

```
SELECT * FROM nomedatabela
```

WHERE f_cnpjcpf(0, campocnpjimportado) < 1;

Dessa maneira, irá selecionar todos os registros onde o número do **cnpj** estiver errado.

Observe que as atribuições são com := e que o IF usa apenas = para seus testes.

Mais exemplos (do livro PostgreSQL a Comprehensive Guide)

```
drop table customers;  
drop table tapes;  
drop table rentals;
```

```
CREATE TABLE "customers" (  
    "customer_id" integer unique not null,  
    "customer_name" character varying(50) not null,  
    "phone" character(8) null,  
    "birth_date" date null,  
    "balance" decimal(7,2)  
);
```

```
CREATE TABLE "tapes" (  
    "tape_id" character(8) not null,  
    "title" character varying(80) not null,  
    "duration" interval  
);
```

```
CREATE TABLE "rentals" (  
    "tape_id" character(8) not null,  
    "rental_date" date not null,  
    "customer_id" integer not null  
);
```

```
INSERT INTO customers VALUES (3, 'Panky, Henry', '555-1221', '1968-01-21', 0.00);  
INSERT INTO customers VALUES (1, 'Jones, Henry', '555-1212', '1970-10-10', 0.00);  
INSERT INTO customers VALUES (4, 'Wonderland, Alice N.', '555-1122', '1969-03-05', 3.00);  
INSERT INTO customers VALUES (2, 'Rubin, William', '555-2211', '1972-07-10', 15.00);
```

```
INSERT INTO tapes VALUES ('AB-12345', 'The Godfather');  
INSERT INTO tapes VALUES ('AB-67472', 'The Godfather');  
INSERT INTO tapes VALUES ('MC-68873', 'Casablanca');  
INSERT INTO tapes VALUES ('OW-41221', 'Citizen Kane');  
INSERT INTO tapes VALUES ('AH-54706', 'Rear Window');
```

```
INSERT INTO rentals VALUES ('AB-12345', '2001-11-25', 1);
```



```
INSERT INTO rentals VALUES ('AB-67472', '2001-11-25', 3);
INSERT INTO rentals VALUES ('OW-41221', '2001-11-25', 1);
INSERT INTO rentals VALUES ('MC-68873', '2001-11-20', 3);

\echo '*****'
\echo 'You may see a few error messages:'
\echo ' table customers does not exist'
\echo ' table tapes does not exist'
\echo ' table rentals does not exist'
\echo ' CREATE TABLE / UNIQUE will create implicit index customers_id_key for table
customers'
\echo "
\echo 'This is normal and you can ignore those messages'
\echo '*****'

-- exchange_index.sql
--
CREATE OR REPLACE FUNCTION get_exchange( CHARACTER )
RETURNS CHARACTER AS '

DECLARE
    result          CHARACTER(3);
BEGIN

    result := SUBSTR( $1, 1, 3 );

    return( result );
END;
' LANGUAGE 'plpgsql' WITH ( ISCACHEABLE );

-- File: dirtree.sql

CREATE OR REPLACE FUNCTION dirtree( TEXT ) RETURNS SETOF _fileinfo AS $$
DECLARE
    file _fileinfo%rowtype;
    child _fileinfo%rowtype;
BEGIN

FOR file IN SELECT * FROM fileinfo( $1 ) LOOP
    IF file.filename != '.' and file.filename != '..' THEN
        file.filename = $1 || '/' || file.filename;

    IF file.filetype = 'd' THEN
        FOR child in SELECT * FROM dirtree( file.filename ) LOOP
            RETURN NEXT child;
        END LOOP;
    END IF;
END IF;
```

```
    RETURN NEXT file;
END IF;
END LOOP;
```

```
RETURN;
```

```
END
$$ LANGUAGE 'PLPGSQL';
```

```
-----
-- my_factorial
-----
```

```
CREATE OR REPLACE FUNCTION my_factorial( value INTEGER ) RETURNS INTEGER AS
$$
```

```
    DECLARE
        arg INTEGER;
    BEGIN
```

```
        arg := value;
```

```
    IF arg IS NULL OR arg < 0 THEN
        RAISE NOTICE 'Invalid Number';
        RETURN NULL;
```

```
    ELSE
```

```
        IF arg = 1 THEN
            RETURN 1;
```

```
        ELSE
```

```
            DECLARE
                next_value INTEGER;
```

```
            BEGIN
```

```
                next_value := my_factorial(arg - 1) * arg;
```

```
                RETURN next_value;
```

```
            END;
```

```
        END IF;
```

```
    END IF;
```

```
END;
```

```
$$ LANGUAGE 'plpgsql';
```

```
-----
-- my_factorial
-----
```

```
CREATE FUNCTION my_factorial( value INTEGER ) RETURNS INTEGER AS $$
```

```
    DECLARE
```

```
        arg INTEGER;
```

```
    BEGIN
```

```

arg := value;

IF arg IS NULL OR arg < 0 THEN
BEGIN
    RAISE NOTICE 'Invalid Number';
    RETURN NULL;
END;
ELSE
    IF arg = 1 THEN
        BEGIN
            RETURN 1;
        END;
    ELSE
        DECLARE
            next_value INTEGER;
        BEGIN
            next_value := my_factorial(arg - 1) * arg;
            RETURN next_value;
        END;
    END IF;
END IF;
END;
$$ LANGUAGE 'plpgsql';

-----
-- compute_due_date
-----

CREATE FUNCTION compute_due_date( DATE ) RETURNS DATE AS $$
DECLARE
    due_date  DATE;
    rental_period  INTERVAL := '7 days';

BEGIN

    due_date := $1 + rental_period;

    RETURN( due_date );

END;
$$ LANGUAGE 'plpgsql';

-----
-- compute_due_date
-----

CREATE FUNCTION compute_due_date( DATE, INTERVAL ) RETURNS DATE AS $$
DECLARE

```

```
rental_date ALIAS FOR $1;  
rental_period ALIAS FOR $2;  
BEGIN
```

```
    RETURN( rental_date + rental_period );
```

```
END;  
$$ LANGUAGE 'plpgsql';
```

```
-----  
-- getBalances  
-----
```

```
CREATE OR REPLACE FUNCTION getBalances( id INTEGER ) RETURNS SETOF NUMERIC  
AS $$
```

```
    DECLARE  
        customer customers%ROWTYPE;  
    BEGIN
```

```
        SELECT * FROM customers INTO customer WHERE customer_id = id;
```

```
        FOR month IN 1..12 LOOP
```

```
            IF customer.monthly_balances[month] IS NOT NULL THEN  
                RETURN NEXT customer.monthly_balances[month];  
            END IF;
```

```
        END LOOP;
```

```
        RETURN;
```

```
    END;  
$$ LANGUAGE 'plpgsql';
```

```
-----  
-- max  
-----
```

```
CREATE OR REPLACE FUNCTION max( arg1 ANYELEMENT, arg2 ANYELEMENT )  
RETURNS ANYELEMENT AS $$
```

```
    BEGIN  
  
        IF( arg1 > arg2 ) THEN  
            RETURN( arg1 );  
        ELSE  
            RETURN( arg2 );  
        END IF;
```

```
END;  
$$ LANGUAGE 'plpgsql';
```

```
-----  
-- firstSmaller  
-----
```

```
CREATE OR REPLACE FUNCTION firstSmaller( arg1 ANYELEMENT, arg2 ANYARRAY )  
RETURNS ANYELEMENT AS $$  
BEGIN
```

```
    FOR i IN array_lower( arg2, 1 ) .. array_upper( arg2, 1 ) LOOP
```

```
        IF arg2[i] < arg1 THEN  
            RETURN( arg2[i] );  
        END IF;
```

```
    END LOOP;
```

```
    RETURN NULL;
```

```
END;  
$$ LANGUAGE 'plpgsql';
```

```
-----  
-- sum  
-----
```

```
CREATE OR REPLACE FUNCTION sum( arg1 ANYARRAY ) RETURNS ANYELEMENT AS $  
$
```

```
    DECLARE  
        result ALIAS FOR $0;  
    BEGIN
```

```
        result := 0;
```

```
    FOR i IN array_lower( arg1, 1 ) .. array_upper( arg1, 1 ) LOOP
```

```
        IF arg1[i] IS NOT NULL THEN  
            result := result + arg1[i];  
        END IF;
```

```
    END LOOP;
```

```
    RETURN( result );
```

```
END;
```

```
$$ LANGUAGE 'plpgsql';
```

Código fonte em:

<http://www.conjectrix.com/pgbook/index.html>

Mais exemplos:

```
CREATE OR REPLACE FUNCTION lacos(tipo_laco int4) RETURNS VOID AS
$body$
DECLARE
    contador int4 NOT NULL DEFAULT 0;
BEGIN
    IF tipo_laco = 1 THEN
        --Loop usando WHILE
        WHILE contador < 10 LOOP
            contador := contador + 1;
            RAISE NOTICE 'Contador: %', contador;
        END LOOP;
    ELSIF tipo_laco = 2 THEN
        --Loop usando LOOP
        LOOP
            contador := contador + 1;
            RAISE NOTICE 'Contador: %', contador;
            EXIT WHEN contador > 9;
        END LOOP;
    ELSE
        --Loop usando FOR
        FOR contador IN 1..10 LOOP
            RAISE NOTICE 'Contador: %', contador;
        END LOOP;
    END IF;
    RETURN;
END;
$body$ LANGUAGE 'plpgsql';
```

Função para remover registros de uma tabela:

```
CREATE OR REPLACE FUNCTION exclui_cliente(pid_cliente int4) RETURNS int4
AS
$body$
DECLARE
    vLinhas int4 DEFAULT 0;
BEGIN
    DELETE FROM clientes WHERE id_cliente = pid_cliente;
    GET DIAGNOSTICS vLinhas = ROW_COUNT;
    RETURN vLinhas;
END;
```

```
$body$ LANGUAGE 'plpgsql';
```

Receber como parâmetro o identificador de um cliente e devolver o seu volume de compras médio:

```
CREATE OR REPLACE FUNCTION media_compras(pid_cliente int4) RETURNS
numeric AS
$body$
DECLARE
    linhaCliente clientes%ROWTYPE;
    mediaCompras numeric(9,2);
    totalCompras numeric(9,2);
    periodo int4;
BEGIN
    SELECT * INTO linhaCliente FROM clientes
        WHERE id_cliente = pid_cliente;
    -- Calcula o período em dias que trabalhamos com o cliente subtraindo da
    -- data atual a data de inclusão do cliente
    periodo := (current_date linhaCliente.data_inclusao);
    -- Coloca na variável totalCompras o somatório de todos os pedidos do
    -- cliente
    SELECT SUM(valor_total) INTO totalCompras FROM pedidos
        WHERE id_cliente = pid_cliente;
    -- Faz a divisão e retorna o resultado
    mediaCompras := totalCompras / periodo;
    RETURN mediaCompras;
END;
$body$ LANGUAGE 'plpgsql';
```

Exemplo com cursores:

```
CREATE OR REPLACE FUNCTION media_compras() RETURNS VOID AS
$body$
DECLARE
    linhaCliente clientes%ROWTYPE;
    mediaCompras numeric(9,2);
    totalCompras numeric(9,2);
    periodo int4;
BEGIN
    FOR linhaCliente IN SELECT * FROM clientes LOOP
        periodo := (current_date linhaCliente.data_inclusao);
        SELECT SUM(valor_total) INTO totalCompras
            FROM pedidos WHERE id_cliente = pid_cliente;
        mediaCompras := totalCompras / periodo;
        UPDATE clientes SET media_compras = mediaCompras
            WHERE id_cliente = pid_cliente;
    END LOOP;
    RETURN;
END;
```

```
$body$ LANGUAGE 'plpgsql';
```

Função baseada na `exclui_cliente` mas genérica, para excluir em qualquer tabela:

```
CREATE OR REPLACE FUNCTION exclui_registro(nome_tabela text, nome_chave
text, id int4) RETURNS int4 AS
$body$
DECLARE
    vLinhas int4 DEFAULT 0;
BEGIN
    EXECUTE 'DELETE FROM ' || nome_tabela || '
        WHERE ' || nome_chave || ' = ' || id;
    GET DIAGNOSTICS vLinhas = ROW_COUNT;
    RETURN vLinhas;
END;
$body$ LANGUAGE 'plpgsql';
```

Outro exemplo de uso da função de exclusão mas agora controlando o uso malicioso de exclusão de uma tabela inteira (excluindo apenas um registro):

```
CREATE OR REPLACE FUNCTION exclui_registro(nome_tabela text, nome_chave
text, id int4) RETURNS int4 AS
$body$
DECLARE
    vLinhas int4 DEFAULT 0;
BEGIN
    EXECUTE 'DELETE FROM ' || nome_tabela || '
        WHERE ' || nome_chave || ' = ' || id;
    GET DIAGNOSTICS vLinhas = ROW_COUNT;
    IF vLinhas > 1 THEN
        RAISE EXCEPTION 'A exclusão de mais de uma linha não é permitida.';
    END IF;
    RETURN vLinhas;
END;
$body$ LANGUAGE 'plpgsql' SECURITY DEFINER;
```

Exemplo com Arrays:

```
CREATE OR REPLACE FUNCTION sp_teste(teste_array integer[]) returns integer[] as
$$
begin

return teste_array;
end;
$$
language plpgsql;
```



```
select sp_teste('{123}');  
  
SELECT sp_teste(ARRAY[1,2]);  
OU  
SELECT sp_teste('{1,2}');
```

-Leo

Tente:
SELECT sp_teste('{1234, 4321}'::int[]);

Oswaldo

```
SELECT sp_teste('{1234,4321}'::integer[]);  
ou  
SELECT sp_teste('{1234,4321}');
```

Como posso retornar diversos campos em uma store procedure?

Tenho uma SP que funciona como uma função para outras SP's, e o processamento dessa SP resulta em 4 resultados que a SP que tiver chamado essa terá que utilizar. Pode ser parametros out ou return array? como funciona isso em Postgres?
Se alguém tiver algum link para indicar também ajuda.

Rúben Lício Reis

```
CREATE FUNCTION r(c1 out text,c2 out text)  
LANGUAGE plpgsql;  
AS $$  
BEGIN  
SELECT 'teste1' AS foo,'teste2' AS bar  
    INTO $1,$2;  
END;  
$$;  
SELECT * FROM r();
```

Leo

Validação de CPF com PL/ PostgreSQL

A partir de hoje passamos a divulgar algoritmos de funções e consultas que sejam de utilidade pública. A validação de CPF com PL/ PostgreSQL foi escolhida em primeiro lugar por ser um algoritmo simples mas bastante útil (além disto, procurei em vários sites e não encontrei um exemplo em PL/ PGSQL).

O CPF é utilizado por muitos sistemas brasileiros como identificação dos indivíduos. Validar o CPF é fazer a verificação dos dois últimos dígitos que são gerados a partir dos nove primeiros. O código

abaixo foi uma tradução mais ou menos literal do código em javascript [deste site](#).

Talvez possa ser feita otimização ou melhoria neste algoritmo, mas a idéia é que vocês o melhorem e atualizem neste site. Estejam à vontade para utilizar e compartilhar este código.

```
CREATE OR REPLACE FUNCTION CPF_Validar(par_cpf varchar(11)) RETURNS integer AS $$
-- ROTINA DE VALIDAÇÃO DE CPF
-- Conversão para o PL/ PGSQL: Cláudio Leopoldino - http://postgresqlbr.blogspot.com/
-- Algoritmo original: http://webmasters.neting.com/msg07743.html
-- Retorna 1 para CPF correto.
DECLARE
x real;
y real; --Variável temporária
soma integer;
dig1 integer; --Primeiro dígito do CPF
dig2 integer; --Segundo dígito do CPF
len integer; -- Tamanho do CPF
contloop integer; --Contador para loop
val_par_cpf varchar(11); --Valor do parâmetro
BEGIN
-- Teste do tamanho da string de entrada
IF char_length(par_cpf) = 11 THEN
ELSE
RAISE NOTICE 'Formato inválido: %',$1;
RETURN 0;
END IF;
-- Inicialização
x := 0;
soma := 0;
dig1 := 0;
dig2 := 0;
contloop := 0;
val_par_cpf := $1; --Atribuição do parâmetro a uma variável interna
len := char_length(val_par_cpf);
x := len - 1;
--Loop de multiplicação - dígito 1
contloop := 1;
WHILE contloop <= (len - 2) LOOP
y := CAST(substring(val_par_cpf from contloop for 1) AS NUMERIC);
soma := soma + ( y * x);
x := x - 1;
contloop := contloop + 1;
END LOOP;
dig1 := 11 - CAST((soma % 11) AS INTEGER);
if (dig1 = 10) THEN dig1 := 0 ; END IF;
if (dig1 = 11) THEN dig1 := 0 ; END IF;

-- Dígito 2
```

```
x := 11; soma :=0;
contloop :=1;
WHILE contloop <= (len -1) LOOP
soma := soma + CAST((substring(val_par_cpf FROM contloop FOR 1)) AS REAL) * x;
x := x - 1;
contloop := contloop +1;
END LOOP;
dig2 := 11 - CAST ((soma % 11) AS INTEGER);
IF (dig2 = 10) THEN dig2 := 0; END IF;
IF (dig2 = 11) THEN dig2 := 0; END IF;
--Teste do CPF
IF ((dig1 || " || dig2) = substring(val_par_cpf FROM len-1 FOR 2)) THEN
RETURN 1;
ELSE
RAISE NOTICE 'DV do CPF Inválido: %', $1;
RETURN 0;
END IF;
END;
$$ LANGUAGE PLPGSQL;
```

Fonte: <http://postgresqlbr.blogspot.com/2008/06/validao-de-cpf-com-pl-pgsql.html>

Mais informações:

<http://www.postgresql.org/docs/current/static/PL/pgSQL.html>
<http://pgdocptbr.sourceforge.net/pg80/plpgsql.html>
<http://www.devmedia.com.br/articles/viewcomp.asp?comp=6906>
<http://www.tek-tips.com/faqs.cfm?fid=3463>
http://imasters.uol.com.br/artigo/1308/stored_procedures_triggers_functions
<http://postgresql.org.br/Documenta%C3%A7%C3%A3o?action=AttachFile&do=get&target=procedures.pdf>

Gatilhos

Sumário

32.1. [Visão geral do comportamento dos gatilhos](#)

32.2. [Visibilidade das mudanças nos dados](#)

32.3. [Gatilhos escritos em C](#)

32.4. [Um exemplo completo](#)

Este capítulo descreve como escrever funções de gatilho. As funções de gatilho podem ser escritas na linguagem C, ou em uma das várias linguagens procedurais disponíveis. No momento não é possível escrever funções de gatilho na linguagem SQL.

Visão geral do comportamento dos gatilhos

O gatilho pode ser definido para executar antes ou depois de uma operação de INSERT, UPDATE ou DELETE, tanto uma vez para cada linha modificada quanto uma vez por instrução SQL. Quando ocorre o evento do gatilho, a função de gatilho é chamada no momento apropriado para tratar o evento.

A função de gatilho deve ser definida antes do gatilho ser criado. A função de gatilho deve ser declarada como uma função que não recebe argumentos e que retorna o tipo trigger (A função de gatilho recebe sua entrada através de estruturas TriggerData passadas especialmente para estas funções, e não na forma comum de argumentos de função).

Tendo sido criada a função de gatilho adequada, o gatilho é estabelecido através do comando [CREATE TRIGGER](#). A mesma função de gatilho pode ser utilizada por vários gatilhos.

Existem dois tipos de gatilhos: gatilhos-por-linha e gatilhos-por-instrução. Em um gatilho-por-linha, a função é chamada uma vez para cada linha afetada pela instrução que disparou o gatilho. Em contraste, um gatilho-por-instrução é chamado somente uma vez quando a instrução apropriada é executada, a despeito do número de linhas afetadas pela instrução. Em particular, uma instrução que não afeta nenhuma linha ainda assim resulta na execução dos gatilhos-por-instrução aplicáveis. Este dois tipos de gatilho são algumas vezes chamados de "gatilhos no nível-de-linha" e "gatilhos no nível-de-instrução", respectivamente.

Os gatilhos no nível-de-instrução "BEFORE" (antes) naturalmente disparam antes da instrução começar a fazer alguma coisa, enquanto os gatilhos no nível-de-instrução "AFTER" (após) disparam bem no final da instrução. Os gatilhos no nível-de-linha "BEFORE" (antes) disparam logo antes da operação em uma determinada linha, enquanto os gatilhos no nível-de-linha "AFTER" (após) disparam no fim da instrução (mas antes dos gatilhos no nível-de-instrução "AFTER").

As funções de gatilho chamadas por gatilhos-por-instrução devem sempre retornar NULL. As funções de gatilho chamadas por gatilhos-por-linha podem retornar uma linha da tabela (um valor do tipo HeapTuple) para o executor da chamada, se assim o decidirem. Os gatilhos no nível-de-linha disparados antes de uma operação possuem as seguintes escolhas:

- Podem retornar NULL para saltar a operação para a linha corrente. Isto instrui ao executor a não realizar a operação no nível-de-linha que chamou o gatilho (a inserção ou a modificação de uma determinada linha da tabela).

- Para os gatilhos de INSERT e UPDATE, no nível-de-linha apenas, a linha retornada se torna a linha que será inserida ou que substituirá a linha sendo atualizada. Isto permite à função de gatilho modificar a linha sendo inserida ou atualizada.

Um gatilho no nível-de-linha, que não pretenda causar nenhum destes comportamentos, deve ter o cuidado de retornar como resultado a mesma linha que recebeu (ou seja, a linha NEW para os gatilhos de INSERT e UPDATE, e a linha OLD para os gatilhos de DELETE).

O valor retornado é ignorado nos gatilhos no nível-de-linha disparados após a operação e, portanto, podem muito bem retornar NULL.

Se for definido mais de um gatilho para o mesmo evento na mesma relação, os gatilhos são disparados pela ordem alfabética de seus nomes. No caso dos gatilhos para antes, a linha possivelmente modificada retornada por cada gatilho se torna a entrada do próximo gatilho. Se algum dos gatilhos para antes retornar NULL, a operação é abandonada e os gatilhos seguintes não são disparados.

Tipicamente, os gatilhos no nível-de-linha que disparam antes são utilizados para verificar ou modificar os dados que serão inseridos ou atualizados. Por exemplo, um gatilho que dispara antes pode ser utilizado para inserir a hora corrente em uma coluna do tipo timestamp, ou para verificar se dois elementos da linha são consistentes. Os gatilhos no nível-de-linha que disparam depois fazem mais sentido para propagar as atualizações para outras tabelas, ou fazer verificação de consistência com relação a outras tabelas. O motivo desta divisão de trabalho é porque um gatilho que dispara depois pode ter certeza de estar vendo o valor final da linha, enquanto um gatilho que dispara antes não pode ter esta certeza; podem haver outros gatilhos que disparam antes disparando após o mesmo. Se não houver nenhum motivo específico para fazer um gatilho disparar antes ou depois, o gatilho para antes é mais eficiente, uma vez que a informação sobre a operação não precisa ser salva até o fim da instrução.

Se a função de gatilho executar comandos SQL, então estes comandos podem disparar gatilhos novamente. Isto é conhecido como cascadear gatilhos. Não existe limitação direta do número de níveis de cascadeamento. É possível que o cascadeamento cause chamadas recursivas do mesmo gatilho; por exemplo, um gatilho para INSERT pode executar um comando que insere uma linha adicional na mesma tabela, fazendo com que o gatilho para INSERT seja disparado novamente. É responsabilidade do programador do gatilho evitar recursões infinitas nestes casos.

Ao se definir um gatilho, podem ser especificados argumentos para o mesmo. A finalidade de se incluir argumentos na definição do gatilho é permitir que gatilhos diferentes com requisitos semelhantes chamem a mesma função. Por exemplo, pode existir uma função de gatilho generalizada que recebe como argumentos dois nomes de colunas e coloca o usuário corrente em uma e a data corrente em outra. Escrita de maneira apropriada, esta função de gatilho se torna independente da tabela específica para a qual está sendo utilizada. Portanto, a mesma função pode ser utilizada para eventos de INSERT em qualquer tabela com colunas apropriadas, para acompanhar automaticamente a criação de registros na tabela de transação, por exemplo. Também pode ser utilizada para acompanhar os eventos de última atualização, se for definida em um gatilho de UPDATE.

Cada linguagem de programação que suporta gatilhos possui o seu próprio método para tornar os dados de entrada do gatilho disponíveis para a função de gatilho. Estes dados de entrada incluem o tipo de evento do gatilho (ou seja, INSERT ou UPDATE), assim como os argumentos listados em CREATE TRIGGER. Para um gatilho no nível-de-linha, os dados de entrada também incluem as linhas NEW para os gatilhos de INSERT e UPDATE, e/ou a linha OLD para os gatilhos de UPDATE e DELETE. Atualmente não há maneira de examinar individualmente as linhas

modificadas pela instrução nos gatilhos no nível-de-instrução.

Visibilidade das mudanças nos dados

Se forem executados comandos SQL na função de gatilho, e estes comandos acessarem a tabela para a qual o gatilho se destina, então deve-se estar ciente das regras de visibilidade dos dados, porque estas determinam se estes comandos SQL enxergam as mudanças nos dados para os quais o gatilho foi disparado. Em resumo:

- Os gatilhos no nível-de-instrução seguem regras simples de visibilidade: nenhuma das modificações feitas pela instrução é enxergada pelos gatilhos no nível-de-instrução chamados antes da instrução, enquanto todas as modificações são enxergadas pelos gatilhos no nível-de-instrução que disparam depois da instrução.
- As modificações nos dados (inserção, atualização e exclusão) causadoras do disparo do gatilho, naturalmente *não* são enxergadas pelos comandos SQL executados em um gatilho no nível-de-linha que dispara antes, porque ainda não ocorreram.
- Entretanto, os comandos SQL executados em um gatilho no nível-de-linha para antes *enxergam* os efeitos das modificações nos dados das linhas processadas anteriormente no mesmo comando externo. Isto requer cautela, uma vez que a ordem destes eventos de modificação geralmente não é previsível; um comando SQL que afeta várias linhas pode atuar sobre as linhas em qualquer ordem.
- Quando é disparado um gatilho no nível-de-linha para depois, todas as modificações nos dados feitas pelo comando externo já estão completas, sendo enxergadas pela função de gatilho chamada.

Podem ser encontradas informações adicionais sobre as regras de visibilidade dos dados na [Seção 40.4](#). O exemplo na [Seção 32.4](#) contém uma demonstração destas regras.

Até a versão atual não existe como criar funções de gatilho na linguagem SQL.

Uma função de gatilho pode ser criada para executar antes (BEFORE) ou após (AFTER) as consultas INSERT, UPDATE OU DELETE, uma vez para cada registro (linha) modificado ou por instrução SQL. Logo que ocorre um desses eventos do gatilho a função do gatilho é disparada automaticamente para tratar o evento.

A função de gatilho deve ser declarada como uma função que não recebe argumentos e que retorna o tipo TRIGGER.

Após criar a função de gatilho, estabelecemos o gatilho pelo comando CREATE TRIGGER. Uma função de gatilho pode ser utilizada por vários gatilhos.

As funções de gatilho chamadas por gatilhos-por-instrução devem sempre retornar NULL.

As funções de gatilho chamadas por gatilhos-por-linha podem retornar uma linha da tabela (um valor do tipo HeapTuple) para o executor da chamada, se assim o decidirem.

Sintaxe:

```
CREATE TRIGGER nome { BEFORE | AFTER } { evento [ OR ... ] }
```

```
ON tabela [ FOR [ EACH ] { ROW | STATEMENT } ]  
EXECUTE PROCEDURE nome_da_função ( argumentos )
```

Triggers em PostgreSQL sempre executam uma função que retorna TRIGGER.

O gatilho fica associado à tabela especificada e executa a função especificada nome_da_função quando determinados eventos ocorrerem.

O gatilho pode ser especificado para disparar antes de tentar realizar a operação na linha (antes das restrições serem verificadas e o comando INSERT, UPDATE ou DELETE ser tentado), ou após a operação estar completa (após as restrições serem verificadas e o INSERT, UPDATE ou DELETE ter completado).

evento

Um entre INSERT, UPDATE ou DELETE; especifica o evento que dispara o gatilho. Vários eventos podem ser especificados utilizando OR.

Exemplos:

```
CREATE TABLE empregados(  
  codigo int4 NOT NULL,  
  nome varchar,  
  salario int4,  
  departamento_cod int4,  
  ultima_data timestamp,  
  ultimo_usuario varchar(50),  
  CONSTRAINT empregados_pkey PRIMARY KEY (codigo) )
```

```
CREATE FUNCTION empregados_gatilho() RETURNS trigger AS $empregados_gatilho$  
BEGIN  
  -- Verificar se foi fornecido o nome e o salário do empregado  
  IF NEW.nome IS NULL THEN  
    RAISE EXCEPTION 'O nome do empregado não pode ser nulo';  
  END IF;  
  IF NEW.salario IS NULL THEN  
    RAISE EXCEPTION '% não pode ter um salário nulo', NEW.nome;  
  END IF;  
  
  -- Quem paga para trabalhar?  
  IF NEW.salario < 0 THEN  
    RAISE EXCEPTION '% não pode ter um salário negativo', NEW.nome;  
  END IF;  
  
  -- Registrar quem alterou a folha de pagamento e quando  
  NEW.ultima_data := 'now';  
  NEW.ultimo_usuario := current_user;  
  RETURN NEW;  
END;
```

```
$empregados_gatilho$ LANGUAGE plpgsql;

CREATE TRIGGER empregados_gatilho BEFORE INSERT OR UPDATE ON empregados
  FOR EACH ROW EXECUTE PROCEDURE empregados_gatilho();

INSERT INTO empregados (codigo,nome, salario) VALUES (5,'João',1000);
INSERT INTO empregados (codigo,nome, salario) VALUES (6,'José',1500);
INSERT INTO empregados (codigo,nome, salario) VALUES (7,'Maria',2500);

SELECT * FROM empregados;

INSERT INTO empregados (codigo,nome, salario) VALUES (5,NULL,1000);
```

NEW – Para INSERT e UPDATE

OLD – Para DELETE

```
CREATE TABLE empregados (
  nome varchar NOT NULL,
  salario integer
);

CREATE TABLE empregados_audit(
  operacao char(1) NOT NULL,
  usuario varchar NOT NULL,
  data timestamp NOT NULL,
  nome varchar NOT NULL,
  salario integer
);
```

```
CREATE OR REPLACE FUNCTION processa_emp_audit() RETURNS TRIGGER AS
$emp_audit$
BEGIN
  --
  -- Cria uma linha na tabela emp_audit para refletir a operação
  -- realizada na tabela emp. Utiliza a variável especial TG_OP
  -- para descobrir a operação sendo realizada.
  --
  IF (TG_OP = 'DELETE') THEN
    INSERT INTO emp_audit SELECT 'E', user, now(), OLD.*;
    RETURN OLD;
  ELSIF (TG_OP = 'UPDATE') THEN
    INSERT INTO emp_audit SELECT 'A', user, now(), NEW.*;
    RETURN NEW;
  ELSIF (TG_OP = 'INSERT') THEN
    INSERT INTO emp_audit SELECT 'I', user, now(), NEW.*;
    RETURN NEW;
  END IF;
```



```
RETURN NULL; -- o resultado é ignorado uma vez que este é um gatilho AFTER
END;
$emp_audit$ language plpgsql;
```

```
CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON empregados
FOR EACH ROW EXECUTE PROCEDURE processa_emp_audit();
```

```
INSERT INTO empregados (nome, salario) VALUES ('João',1000);
INSERT INTO empregados (nome, salario) VALUES ('José',1500);
INSERT INTO empregados (nome, salario) VALUES ('Maria',250);
UPDATE empregados SET salario = 2500 WHERE nome = 'Maria';
DELETE FROM empregados WHERE nome = 'João';
```

```
SELECT * FROM empregados;
```

```
SELECT * FROM empregados_audit;
Outro exemplo:
```

```
CREATE TABLE empregados (
    codigo      serial PRIMARY KEY,
    nome        varchar NOT NULL,
    salario     integer
);
```

```
CREATE TABLE empregados_audit(
    usuario     varchar NOT NULL,
    data        timestamp NOT NULL,
    id          integer NOT NULL,
    coluna      text NOT NULL,
    valor_antigo text NOT NULL,
    valor_novo  text NOT NULL
);
```

```
CREATE OR REPLACE FUNCTION processa_emp_audit() RETURNS TRIGGER AS
$emp_audit$
BEGIN
    --
    -- Não permitir atualizar a chave primária
    --
    IF (NEW.codigo <> OLD.codigo) THEN
        RAISE EXCEPTION 'Não é permitido atualizar o campo codigo';
    END IF;
    --
    -- Inserir linhas na tabela emp_audit para refletir as alterações
    -- realizada na tabela emp.
    --
    IF (NEW.nome <> OLD.nome) THEN
```

```

INSERT INTO emp_audit SELECT current_user, current_timestamp,
    NEW.id, 'nome', OLD.nome, NEW.nome;
END IF;
IF (NEW.salario <> OLD.salario) THEN
    INSERT INTO emp_audit SELECT current_user, current_timestamp,
        NEW.codigo, 'salario', OLD.salario, NEW.salario;
END IF;
RETURN NULL; -- o resultado é ignorado uma vez que este é um gatilho AFTER
END;
$emp_audit$ language plpgsql;

```

```

CREATE TRIGGER emp_audit
AFTER UPDATE ON empregados
FOR EACH ROW EXECUTE PROCEDURE processa_emp_audit();

```

```

INSERT INTO empregados (nome, salario) VALUES ('João',1000);
INSERT INTO empregados (nome, salario) VALUES ('José',1500);
INSERT INTO empregados (nome, salario) VALUES ('Maria',2500);
UPDATE empregados SET salario = 2500 WHERE id = 2;
UPDATE empregados SET nome = 'Maria Cecília' WHERE id = 3;
UPDATE empregados SET codigo=100 WHERE codigo=1;
ERRO: Não é permitido atualizar o campo codigo
SELECT * FROM empregados;

```

```
SELECT * FROM empregados_audit;
```

Crie a mesma função que insira o nome da empresa e o nome do cliente retornando o id de ambos

```

create or replace function empresa_cliente_id(varchar,varchar) returns _int4 as
,
declare
    nempresa alias for $1;
    ncliente alias for $2;
    empresaid integer;
    clienteid integer;
begin
    insert into empresas(nome) values(nempresa);
    insert into clientes(fkempresa,nome) values (currval ("empresas_id_seq"), ncliente);
    empresaid := currval("empresas_id_seq");
    clienteid := currval("clientes_id_seq");

    return "{"|| empresaid ||","|| clienteid ||"}";
end;
,
language 'plpgsql';

```

Crie uma função onde passamos como parâmetro o id do cliente e seja retornado o seu

nome

create or replace function id_nome_cliente(integer) returns text as

```
,
declare
    r record;
begin
    select into r * from clientes where id = $1;
    if not found then
        raise exception "Cliente não existente !";
    end if;
    return r.nome;
end;
,
language 'plpgsql';
```

Crie uma função que retorne os nome de toda a tabela clientes concatenados em um só campo

create or replace function clientes_nomes() returns text as

```
,
declare
    x text;
    r record;
begin
    x:="Inicio";
    for r in select * from clientes order by id loop
        x:= x||" : "||r.nome;
    end loop;
    return x||" : fim";
end;
,
language 'plpgsql';
```

Gatilhos escritos em PL/pgSQL

A linguagem PL/pgSQL pode ser utilizada para definir procedimentos de gatilho. O procedimento de gatilho é criado pelo comando CREATE FUNCTION, declarando o procedimento como uma função sem argumentos e que retorna o tipo trigger. Deve ser observado que a função deve ser declarada sem argumentos, mesmo que espere receber os argumentos especificados no comando CREATE TRIGGER — os argumentos do gatilho são passados através de TG_ARGV, conforme descrito abaixo.

Quando uma função escrita em PL/pgSQL é chamada como um gatilho, diversas variáveis especiais são criadas automaticamente no bloco de nível mais alto. São estas:

NEW

Tipo de dado RECORD; variável contendo a nova linha do banco de dados, para as operações de INSERT/UPDATE nos gatilhos no nível de linha. O valor desta variável é NULL nos gatilhos no nível de instrução.

OLD

Tipo de dado RECORD; variável contendo a antiga linha do banco de dados, para as operações de UPDATE/DELETE nos gatilhos no nível de linha. O valor desta variável é NULL nos gatilhos no nível de instrução.

TG_NAME

Tipo de dado name; variável contendo o nome do gatilho disparado.

TG_WHEN

Tipo de dado text; uma cadeia de caracteres contendo BEFORE ou AFTER, dependendo da definição do gatilho.

TG_LEVEL

Tipo de dado text; uma cadeia de caracteres contendo ROW ou STATEMENT, dependendo da definição do gatilho.

TG_OP

Tipo de dado text; uma cadeia de caracteres contendo INSERT, UPDATE, ou DELETE, informando para qual operação o gatilho foi disparado.

TG_RELID

Tipo de dado oid; o ID de objeto da tabela que causou o disparo do gatilho.

TG_RELNAME

Tipo de dado name; o nome da tabela que causou o disparo do gatilho.

TG_NARGS

Tipo de dado integer; o número de argumentos fornecidos ao procedimento de gatilho na instrução CREATE TRIGGER.

TG_ARGV[]

Tipo de dado matriz de text; os argumentos da instrução CREATE TRIGGER. O contador do índice começa por 0. Índices inválidos (menor que 0 ou maior ou igual a tg_nargs) resultam em um valor nulo.

Uma função de gatilho deve retornar nulo, ou um valor registro/linha possuindo a mesma estrutura da tabela para a qual o gatilho foi disparado.

Os gatilhos no nível de linha disparados BEFORE (antes) podem retornar nulo, para sinalizar ao gerenciador do gatilho para pular o restante da operação para esta linha (ou seja, os gatilhos posteriores não serão disparados, e não ocorrerá o INSERT/UPDATE/DELETE para esta linha. Se for retornado um valor diferente de nulo, então a operação prossegue com este valor de linha. Retornar um valor de linha diferente do valor original de NEW altera a linha que será inserida ou atualizada (mas não tem efeito direto no caso do DELETE). Para alterar a linha a ser armazenada, é possível substituir valores individuais diretamente em NEW e retornar o NEW modificado, ou construir um novo registro/linha completo a ser retornado.

O valor retornado por um gatilho BEFORE ou AFTER no nível de instrução, ou por um gatilho AFTER no nível de linha, é sempre ignorado; pode muito bem ser nulo. Entretanto, qualquer um destes tipos de gatilho pode interromper toda a operação gerando um erro.

O [Exemplo 35-1](#) mostra um exemplo de procedimento de gatilho escrito em PL/pgSQL.

Exemplo 35-1. Procedimento de gatilho PL/pgSQL

O gatilho deste exemplo garante que quando é inserida ou atualizada uma linha na tabela, fica sempre registrado nesta linha o usuário que efetuou a inserção ou a atualização, e quando isto ocorreu. Além disso, o gatilho verifica se é fornecido o nome do empregado, e se o valor do salário é um número positivo.

```
CREATE TABLE emp (
    nome_emp      text,
    salario       integer,
    ultima_data    timestamp,
    ultimo_usuario text
);

CREATE FUNCTION emp_gatilho() RETURNS trigger AS $emp_gatilho$
BEGIN
    -- Verificar se foi fornecido o nome e o salário do empregado
    IF NEW.nome_emp IS NULL THEN
        RAISE EXCEPTION 'O nome do empregado não pode ser nulo';
    END IF;
    IF NEW.salario IS NULL THEN
        RAISE EXCEPTION '% não pode ter um salário nulo', NEW.nome_emp;
    END IF;

    -- Quem paga para trabalhar?
    IF NEW.salario < 0 THEN
        RAISE EXCEPTION '% não pode ter um salário negativo', NEW.nome_emp;
    END IF;

    -- Registrar quem alterou a folha de pagamento e quando
    NEW.ultima_data := 'now';
    NEW.ultimo_usuario := current_user;
    RETURN NEW;
END;
$emp_gatilho$ LANGUAGE plpgsql;

CREATE TRIGGER emp_gatilho BEFORE INSERT OR UPDATE ON emp
FOR EACH ROW EXECUTE PROCEDURE emp_gatilho();
```

```
INSERT INTO emp (nome_emp, salario) VALUES ('João',1000);
INSERT INTO emp (nome_emp, salario) VALUES ('José',1500);
INSERT INTO emp (nome_emp, salario) VALUES ('Maria',2500);
```

```
SELECT * FROM emp;
```

nome_emp	salario	ultima_data	ultimo_usuario
João	1000	2005-11-25 07:07:50.59532	folha
José	1500	2005-11-25 07:07:50.691905	folha
Maria	2500	2005-11-25 07:07:50.694995	folha

(3 linhas)

Exemplo 35-2. Procedimento de gatilho PL/pgSQL para registrar inserção e atualização

O gatilho deste exemplo garante que quando é inserida ou atualizada uma linha na tabela, fica sempre registrado nesta linha o usuário que efetuou a inserção ou a atualização, e quando isto ocorreu. Porém, diferentemente do gatilho anterior, a criação e a atualização da linha são registradas em colunas diferentes. Além disso, o gatilho verifica se é fornecido o nome do empregado, e se o valor do salário é um número positivo. [1]

```
CREATE TABLE emp (
    nome_emp      text,
    salario       integer,
    usu_cria      text,          -- Usuário que criou a linha
    data_cria     timestamp,    -- Data da criação da linha
    usu_atu       text,          -- Usuário que fez a atualização
    data_atu      timestamp     -- Data da atualização
);

CREATE FUNCTION emp_gatilho() RETURNS trigger AS $emp_gatilho$
BEGIN
    -- Verificar se foi fornecido o nome do empregado
    IF NEW.nome_emp IS NULL THEN
        RAISE EXCEPTION 'O nome do empregado não pode ser nulo';
    END IF;
    IF NEW.salario IS NULL THEN
        RAISE EXCEPTION '% não pode ter um salário nulo', NEW.nome_emp;
    END IF;

    -- Quem paga para trabalhar?
    IF NEW.salario < 0 THEN
        RAISE EXCEPTION '% não pode ter um salário negativo', NEW.nome_emp;
    END IF;

    -- Registrar quem criou a linha e quando
    IF (TG_OP = 'INSERT') THEN
        NEW.data_cria := current_timestamp;
        NEW.usu_cria  := current_user;
    -- Registrar quem alterou a linha e quando
    ELSIF (TG_OP = 'UPDATE') THEN
        NEW.data_atu := current_timestamp;
        NEW.usu_atu  := current_user;
    END IF;
    RETURN NEW;
END;
$emp_gatilho$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER emp_gatilho BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_gatilho();
```

```
INSERT INTO emp (nome_emp, salario) VALUES ('João',1000);
INSERT INTO emp (nome_emp, salario) VALUES ('José',1500);
INSERT INTO emp (nome_emp, salario) VALUES ('Maria',250);
UPDATE emp SET salario = 2500 WHERE nome_emp = 'Maria';
```

```
SELECT * FROM emp;
```

nome_emp	salario	usu_cria	data_cria	usu_atu
João	1000	folha	2005-11-25 08:11:40.63868	
José	1500	folha	2005-11-25 08:11:40.674356	
Maria	2500	folha	2005-11-25 08:11:40.679592	folha

2005-11-25 08:11:40.682394
(3 linhas)

Uma outra maneira de registrar as modificações na tabela envolve a criação de uma nova tabela contendo uma linha para cada inserção, atualização ou exclusão que ocorra. Esta abordagem pode ser considerada como uma auditoria das mudanças na tabela. O [Exemplo 35-3](#) mostra um procedimento de gatilho de auditoria em PL/pgSQL.

Exemplo 35-3. Procedimento de gatilho PL/pgSQL para auditoria

Este gatilho garante que todas as inserções, atualizações e exclusões de linha na tabela emp são registradas na tabela emp_audit, para permitir auditar as operações efetuadas na tabela emp. O nome de usuário e a hora corrente são gravadas na linha, junto com o tipo de operação que foi realizada.

```
CREATE TABLE emp (
    nome_emp    text NOT NULL,
    salario     integer
);
```

```
CREATE TABLE emp_audit(
    operacao    char(1)    NOT NULL,
    usuario     text       NOT NULL,
    data        timestamp  NOT NULL,
    nome_emp    text       NOT NULL,
    salario     integer
);
```

```
CREATE OR REPLACE FUNCTION processa_emp_audit() RETURNS TRIGGER AS $emp_audit$
BEGIN
    --
    -- Cria uma linha na tabela emp_audit para refletir a operação
    -- realizada na tabela emp. Utiliza a variável especial TG_OP
    -- para descobrir a operação sendo realizada.
    --
    IF (TG_OP = 'DELETE') THEN
        INSERT INTO emp_audit SELECT 'E', user, now(), OLD.*;
        RETURN OLD;
    ELSIF (TG_OP = 'UPDATE') THEN
        INSERT INTO emp_audit SELECT 'A', user, now(), NEW.*;
        RETURN NEW;
```

```

        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit SELECT 'I', user, now(), NEW.*;
            RETURN NEW;
        END IF;
        RETURN NULL; -- o resultado é ignorado uma vez que este é um gatilho
AFTER
    END;
$emp_audit$ language plpgsql;

```

```

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE PROCEDURE processa_emp_audit();

```

```

INSERT INTO emp (nome_emp, salario) VALUES ('João',1000);
INSERT INTO emp (nome_emp, salario) VALUES ('José',1500);
INSERT INTO emp (nome_emp, salario) VALUES ('Maria',250);
UPDATE emp SET salario = 2500 WHERE nome_emp = 'Maria';
DELETE FROM emp WHERE nome_emp = 'João';

```

```
SELECT * FROM emp;
```

```

nome_emp | salario
-----+-----
José     |    1500
Maria    |    2500
(2 linhas)

```

```
SELECT * FROM emp_audit;
```

```

operacao | usuario | data | nome_emp | salario
-----+-----+-----+-----+-----
I        | folha   | 2005-11-25 09:06:03.008735 | João   |    1000
I        | folha   | 2005-11-25 09:06:03.014245 | José   |    1500
I        | folha   | 2005-11-25 09:06:03.049443 | Maria  |     250
A        | folha   | 2005-11-25 09:06:03.052972 | Maria  |    2500
E        | folha   | 2005-11-25 09:06:03.056774 | João   |    1000
(5 linhas)

```

Exemplo 35-4. Procedimento de gatilho PL/pgSQL para auditoria no nível de coluna

Este gatilho registra todas as atualizações realizadas nas colunas nome_emp e salario da tabela emp na tabela emp_audit (isto é, as colunas são auditadas). O nome de usuário e a hora corrente são registrados junto com a chave da linha (id) e a informação atualizada. Não é permitido atualizar a chave da linha. Este exemplo difere do anterior pela auditoria ser no nível de coluna, e não no nível de linha. [2]

```

CREATE TABLE emp (
    id          serial PRIMARY KEY,
    nome_emp    text NOT NULL,
    salario     integer
);

CREATE TABLE emp_audit (
    usuario     text NOT NULL,
    data        timestamp NOT NULL,
    id          integer NOT NULL,
    coluna      text NOT NULL,
    valor_antigo text NOT NULL,

```



```

        valor_novo        text        NOT NULL
    );

CREATE OR REPLACE FUNCTION processa_emp_audit() RETURNS TRIGGER AS $emp_audit$
BEGIN
    --
    -- Não permitir atualizar a chave primária
    --
    IF (NEW.id <> OLD.id) THEN
        RAISE EXCEPTION 'Não é permitido atualizar o campo ID';
    END IF;
    --
    -- Inserir linhas na tabela emp_audit para refletir as alterações
    -- realizada na tabela emp.
    --
    IF (NEW.nome_emp <> OLD.nome_emp) THEN
        INSERT INTO emp_audit SELECT current_user, current_timestamp,
                                    NEW.id, 'nome_emp', OLD.nome_emp, NEW.nome_emp;
    END IF;
    IF (NEW.salario <> OLD.salario) THEN
        INSERT INTO emp_audit SELECT current_user, current_timestamp,
                                    NEW.id, 'salario', OLD.salario, NEW.salario;
    END IF;
    RETURN NULL; -- o resultado é ignorado uma vez que este é um gatilho
AFTER
    END;
$emp_audit$ language plpgsql;

CREATE TRIGGER emp_audit
AFTER UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE processa_emp_audit();

INSERT INTO emp (nome_emp, salario) VALUES ('João',1000);
INSERT INTO emp (nome_emp, salario) VALUES ('José',1500);
INSERT INTO emp (nome_emp, salario) VALUES ('Maria',2500);
UPDATE emp SET salario = 2500 WHERE id = 2;
UPDATE emp SET nome_emp = 'Maria Cecília' WHERE id = 3;
UPDATE emp SET id=100 WHERE id=1;
ERRO: Não é permitido atualizar o campo ID

SELECT * FROM emp;

 id | nome_emp | salario
---+-----+-----
  1 | João     |   1000
  2 | José     |   2500
  3 | Maria Cecília |   2500
(3 linhas)

SELECT * FROM emp_audit;

 usuario | data | id | coluna | valor_antigo | valor_novo
-----+-----+-----+-----+-----+-----
+-----+
 folha   | 2005-11-25 12:21:08.493268 | 2 | salario | 1500          | 2500
 folha   | 2005-11-25 12:21:08.49822  | 3 | nome_emp | Maria         | Maria
Cecília
(2 linhas)

```

Uma das utilizações de gatilho é para manter uma tabela contendo o sumário de outra tabela. O sumário produzido pode ser utilizado no lugar da tabela original em diversas consultas — geralmente com um tempo de execução bem menor. Esta técnica é muito utilizada em Armazém de Dados (Data Warehousing), onde as tabelas dos dados medidos ou observados (chamadas de tabelas fato) podem ser muito grandes. O [Exemplo 35-5](#) mostra um procedimento de gatilho em PL/pgSQL para manter uma tabela de sumário de uma tabela fato em um armazém de dados.

Exemplo 35-5. Procedimento de gatilho PL/pgSQL para manter uma tabela sumário

O esquema que está detalhado a seguir é parcialmente baseado no exemplo *Grocery Store* do livro *The Data Warehouse Toolkit* de Ralph Kimball.

```
--
-- Main tables - time dimension and sales fact.
--
CREATE TABLE time_dimension (
    time_key          integer NOT NULL,
    day_of_week       integer NOT NULL,
    day_of_month       integer NOT NULL,
    month             integer NOT NULL,
    quarter           integer NOT NULL,
    year              integer NOT NULL
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);

CREATE TABLE sales_fact (
    time_key          integer NOT NULL,
    product_key       integer NOT NULL,
    store_key         integer NOT NULL,
    amount_sold       numeric(12,2) NOT NULL,
    units_sold        integer NOT NULL,
    amount_cost       numeric(12,2) NOT NULL
);
CREATE INDEX sales_fact_time ON sales_fact(time_key);

--
-- Summary table - sales by time.
--
CREATE TABLE sales_summary_bytime (
    time_key          integer NOT NULL,
    amount_sold       numeric(15,2) NOT NULL,
    units_sold        numeric(12) NOT NULL,
    amount_cost       numeric(15,2) NOT NULL
);
CREATE UNIQUE INDEX sales_summary_bytime_key ON sales_summary_bytime(time_key);

--
-- Function and trigger to amend summarized column(s) on UPDATE, INSERT, DELETE.
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime() RETURNS TRIGGER AS
$maint_sales_summary_bytime$
    DECLARE
        delta_time_key          integer;
        delta_amount_sold       numeric(15,2);
        delta_units_sold        numeric(12);
        delta_amount_cost       numeric(15,2);
    BEGIN
```

```

-- Work out the increment/decrement amount(s).
IF (TG_OP = 'DELETE') THEN

    delta_time_key = OLD.time_key;
    delta_amount_sold = -1 * OLD.amount_sold;
    delta_units_sold = -1 * OLD.units_sold;
    delta_amount_cost = -1 * OLD.amount_cost;

ELSIF (TG_OP = 'UPDATE') THEN

    -- forbid updates that change the time_key -
    -- (probably not too onerous, as DELETE + INSERT is how most
    -- changes will be made).
    IF ( OLD.time_key != NEW.time_key) THEN
        RAISE EXCEPTION 'Update of time_key : % -> % not allowed',
OLD.time_key, NEW.time_key;
    END IF;

    delta_time_key = OLD.time_key;
    delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
    delta_units_sold = NEW.units_sold - OLD.units_sold;
    delta_amount_cost = NEW.amount_cost - OLD.amount_cost;

ELSIF (TG_OP = 'INSERT') THEN

    delta_time_key = NEW.time_key;
    delta_amount_sold = NEW.amount_sold;
    delta_units_sold = NEW.units_sold;
    delta_amount_cost = NEW.amount_cost;

END IF;

-- Update the summary row with the new values.
UPDATE sales_summary_bytime
    SET amount_sold = amount_sold + delta_amount_sold,
        units_sold = units_sold + delta_units_sold,
        amount_cost = amount_cost + delta_amount_cost
    WHERE time_key = delta_time_key;

-- There might have been no row with this time_key (e.g new data!).
IF (NOT FOUND) THEN
    BEGIN
        INSERT INTO sales_summary_bytime (
            time_key,
            amount_sold,
            units_sold,
            amount_cost)
        VALUES (
            delta_time_key,
            delta_amount_sold,
            delta_units_sold,
            delta_amount_cost
        );
    EXCEPTION
        --
        -- Catch race condition when two transactions are adding data
        -- for a new time_key.

```

```

--
WHEN UNIQUE_VIOLATION THEN
    UPDATE sales_summary_bytime
        SET amount_sold = amount_sold + delta_amount_sold,
            units_sold = units_sold + delta_units_sold,
            amount_cost = amount_cost + delta_amount_cost
        WHERE time_key = delta_time_key;

    END;
END IF;
RETURN NULL;

END;
$maint_sales_summary_bytime$ LANGUAGE plpgsql;

CREATE TRIGGER maint_sales_summary_bytime
AFTER INSERT OR UPDATE OR DELETE ON sales_fact
    FOR EACH ROW EXECUTE PROCEDURE maint_sales_summary_bytime();

```

Exemplo 35-6. Procedimento de gatilho para controlar sobreposição de datas

O gatilho deste exemplo verifica se o compromisso sendo agendado ou modificado se sobrepõe a outro compromisso já agendado. Se houver sobreposição, emite mensagem de erro e não permite a operação. [3]

Abaixo está mostrado o script utilizado para criar a tabela, a função de gatilho e os gatilhos de inserção e atualização.

```

CREATE TABLE agendamentos (
    id          SERIAL PRIMARY KEY,
    nome        TEXT,
    evento      TEXT,
    data_inicio TIMESTAMP,
    data_fim    TIMESTAMP
);

CREATE FUNCTION fun_verifica_agendamentos() RETURNS "trigger" AS
$fun_verifica_agendamentos$
BEGIN
    /* Verificar se a data de início é maior que a data de fim */
    IF NEW.data_inicio > NEW.data_fim THEN
        RAISE EXCEPTION 'A data de início não pode ser maior que a data de
fim';
    END IF;
    /* Verificar se há sobreposição com agendamentos existentes */
    IF EXISTS (
        SELECT 1
        FROM agendamentos
        WHERE nome = NEW.nome
            AND ((data_inicio, data_fim) OVERLAPS
                (NEW.data_inicio, NEW.data_fim))
    )
    THEN
        RAISE EXCEPTION 'impossível agendar - existe outro compromisso';
    END IF;
    RETURN NEW;
END;
$fun_verifica_agendamentos$ LANGUAGE plpgsql;

```

```
COMMENT ON FUNCTION fun_verifica_agendamentos() IS
    'Verifica se o agendamento é possível';
```

```
CREATE TRIGGER trg_agendamentos_ins
    BEFORE INSERT ON agendamentos
    FOR EACH ROW
    EXECUTE PROCEDURE fun_verifica_agendamentos();
```

```
CREATE TRIGGER trg_agendamentos_upd
    BEFORE UPDATE ON agendamentos
    FOR EACH ROW
    EXECUTE PROCEDURE fun_verifica_agendamentos();
```

Abaixo está mostrado um exemplo de utilização do gatilho. Deve ser observado que os intervalos ('2005-08-23 14:00:00', '2005-08-23 15:00:00') e ('2005-08-23 15:00:00', '2005-08-23 16:00:00') não se sobrepõem, uma vez que o primeiro intervalo termina às quinze horas, enquanto o segundo intervalo inicia às quinze horas, estando, portanto, o segundo intervalo imediatamente após o primeiro.

```
=> INSERT INTO agendamentos VALUES
(DEFAULT, 'Joana', 'Congresso', '2005-08-23', '2005-08-24');
=> INSERT INTO agendamentos VALUES
(DEFAULT, 'Joana', 'Viagem', '2005-08-24', '2005-08-26');
=> INSERT INTO agendamentos VALUES
(DEFAULT, 'Joana', 'Palestra', '2005-08-23', '2005-08-26');
ERRO: impossível agendar - existe outro compromisso
=> INSERT INTO agendamentos VALUES (DEFAULT, 'Maria', 'Cabeleireiro', '2005-08-23
14:00:00', '2005-08-23 15:00:00');
=> INSERT INTO agendamentos VALUES (DEFAULT, 'Maria', 'Manicure', '2005-08-23
15:00:00', '2005-08-23 16:00:00');
=> INSERT INTO agendamentos VALUES (DEFAULT, 'Maria', 'Médico', '2005-08-23
14:30:00', '2005-08-23 15:00:00');
ERRO: impossível agendar - existe outro compromisso
=> UPDATE agendamentos SET data_inicio='2005-08-24' WHERE id=2;
ERRO: impossível agendar - existe outro compromisso
=> SELECT * FROM agendamentos;
```

id	nome	evento	data_inicio	data_fim
1	Joana	Congresso	2005-08-23 00:00:00	2005-08-24 00:00:00
2	Joana	Viagem	2005-08-24 00:00:00	2005-08-26 00:00:00
4	Maria	Cabeleireiro	2005-08-23 14:00:00	2005-08-23 15:00:00
5	Maria	Manicure	2005-08-23 15:00:00	2005-08-23 16:00:00

(4 linhas)

Notas

- [1] Exemplo escrito pelo tradutor, não fazendo parte do manual original.
- [2] Exemplo escrito pelo tradutor, não fazendo parte do manual original.
- [3] Exemplo escrito pelo tradutor, não fazendo parte do manual original, baseado em exemplo da lista de discussão pgsql-sql.

Exemplo do artigo do Kauai Aires em:

http://imasters.uol.com.br/artigo/5033/postgresql/auditoria_em_banco_de_dados_postgresql/imprimir/

/*

OBJETIVO: INSTALAR A LINGUAGEM PL/pgSQL NO BANCO DE DADOS E PARTE BÁSICA DO SISTEMA.

AUTOR: KAUI AIRES OLIVEIRA - ARQUITETO DE BANCO DE DADOS

CONTATO: KAUIAIRES@GMAIL.COM

CRIAÇÃO: 20/10/2006

ATUALIZAÇÃO: 23/10/2006

BANCO: POSTGRESQL

*/

-- PRÉ-REQUISITO:

-- -----

-- Caso a linguagem plpgsql ainda não tenha sido definida em seu banco de dados, DEVE ENTÃO CONTINUAR COM TODO ESTE SCRIPT.

-- O terminal interativo do PostgreSQL (psql) poderá ser utilizado para essa finalidade, lembre-se que o usuário deverá possuir privilégio de "administrador" do banco de dados.

-- A função create_trigger(), assim como todas as demais funções criadas pela mesma, são escritas nessa "linguagem procedural".

-- A infra-estrutura é criada no esquema (espaço de tabela) corrente.

-- CRIA A LINGUAGEM PL/pgSQL

CREATE TRUSTED PROCEDURAL LANGUAGE plpgsql

HANDLER plpgsql_call_handler;

COMMIT;

-- CRIA O USUARIO PARA AUDITORIA NO POSTGRES

-- Role: "auditoria_banco"

-- DROP ROLE auditoria_banco;

CREATE ROLE auditoria_banco LOGIN

ENCRYPTED PASSWORD 'md58662fb9d00ee6acc21190bfce1f93fda'

NOSUPERUSER INHERIT CREATEDB NOCREATEROLE;

-- OBS.: ESTE PASSWORD É: "auditoria_banco" SEM ASPAS

COMMIT;

-- CRIA O SCHEMA NO BANCO DE AUDITORIA

CREATE SCHEMA auditoria_banco

AUTHORIZATION auditoria_banco;

```
COMMENT ON SCHEMA auditoria_banco IS 'SCHEMA DE AUTIDORIA DO BANCO DE
DADOS POSTGRESQL - POR KAUI AIRES - KAUIAIRES@GMAIL.COM';
COMMIT;
```

```
-- REGISTRA A FUNÇÃO TRATADORA DE CHAMADAS
CREATE OR REPLACE FUNCTION "auditoria_banco"."plpgsql_call_handler"() RETURNS
language_handler
    AS '$libdir/plpgsql'
LANGUAGE C;
```

```
COMMENT ON FUNCTION "auditoria_banco"."plpgsql_call_handler"()
    IS 'Registra o tratador de chamadas associado linguagem plpgsql';
COMMIT;
ALTER FUNCTION "auditoria_banco"."plpgsql_call_handler"() OWNER TO auditoria_banco;
GRANT ALL ON SCHEMA auditoria_banco TO auditoria_banco WITH GRANT OPTION;
GRANT USAGE ON SCHEMA auditoria_banco TO public;
COMMIT;
```

```
-- FIM ESTRUTURA BÁSICA
```

```
/*
    OBJETIVO:  INSTALAR A LINGUAGEM PL/pgSQL NO BANCO DE DADOS E PARTE
    BÁSICA DO SISTEMA.
```

```
    AUTOR:    KAUI AIRES OLIVEIRA - ARQUITETO DE BANCO DE DADOS
```

```
    CONTATO: KAUIAIRES@GMAIL.COM
```

```
    CRIAÇÃO:   20/10/2006
```

```
    ATUALIZAÇÃO: 23/10/2006
```

```
    BANCO: POSTGRESQL
```

```
*/
```

```
create table "auditoria_banco"."tab_dado_auditado"
(
    "nome_usuario_banco" varchar(200) not null,
    "data_hora_acesso" timestamp not null,
    "nome_tabela" varchar(200) not null,
    "operacao_tabela" varchar(20) not null,
    "ip_maquina_acesso" cidr not null,
    "sessao_usuario" varchar(255) not null
) with oids;
```

```
/* create indexes */
```

```
create index "idx_tab_dado_auditado_01" on "auditoria_banco"."tab_dado_auditado" using btree
("operacao_tabela");
create index "idx_tab_dado_auditado_02" on "auditoria_banco"."tab_dado_auditado" using btree
("ip_maquina_acesso");
```

```
/* create role permissions */
/* role permissions on tables */
grant select on "auditoria_banco"."tab_dado_auditado" to "auditoria_banco";
grant update on "auditoria_banco"."tab_dado_auditado" to "auditoria_banco";
grant delete on "auditoria_banco"."tab_dado_auditado" to "auditoria_banco";
grant insert on "auditoria_banco"."tab_dado_auditado" to "auditoria_banco";
grant references on "auditoria_banco"."tab_dado_auditado" to "auditoria_banco";
```

```
/* role permissions on views */
```

```
/* role permissions on procedures */
```

```
/* create comment on tables */
comment on table "auditoria_banco"."tab_dado_auditado" is 'tabela contendo os dados auditados';
```

```
/* create comment on columns */
comment on column "auditoria_banco"."tab_dado_auditado"."nome_usuario_banco" is 'nome do
usuario do banco de dados que executou tal operacao';
comment on column "auditoria_banco"."tab_dado_auditado"."data_hora_acesso" is 'data de acesso';
comment on column "auditoria_banco"."tab_dado_auditado"."nome_tabela" is 'nome da tabela em
que foi realizado a alteracao';
comment on column "auditoria_banco"."tab_dado_auditado"."operacao_tabela" is 'operacoes
realizadas em tabelas tipo insert, update ou delete';
comment on column "auditoria_banco"."tab_dado_auditado"."ip_maquina_acesso" is 'ip maquina
acesso sistema';
comment on column "auditoria_banco"."tab_dado_auditado"."sessao_usuario" is 'sessao do
usuario';
```

```
/* create comment on domains and types */
```

```
/* create comment on indexes */
comment on index "auditoria_banco"."idx_tab_dado_auditado_01" is null;
comment on index "auditoria_banco"."idx_tab_dado_auditado_02" is null;
```

```
commit;
ALTER TABLE "auditoria_banco"."tab_dado_auditado" OWNER TO auditoria_banco;
```



```
GRANT INSERT ON TABLE auditoria_banco.tab_dado_auditado TO public;
```

```
/*
```

```
    OBJETIVO:  INSTALAR A LINGUAGEM PL/pgSQL NO BANCO DE DADOS E PARTE
BÁSICA DO SISTEMA.
```

```
    AUTOR:    KAUI AIRES OLIVEIRA - ARQUITETO DE BANCO DE DADOS
```

```
    CONTATO: KAUIAIRES@GMAIL.COM
```

```
    CRIAÇÃO:   20/10/2006
```

```
    ATUALIZAÇÃO: 23/10/2006
```

```
    BANCO: POSTGRESQL
```

```
*/
```

```
-- SOMENTE PARA TESTE VAMOS CRIAR UMA TABELA "EMPREGADO" E CONFERIR
NOSSA AUDITORIA
```

```
CREATE TABLE "public"."empregado" (
    "nome_empregado"  VARCHAR(150) NOT NULL,
    "salario"         integer
) with oids;
```

```
--FEITO ISTO VAMOS AO GATILHO
```

```
/*
```

```
    OBJETIVO:  INSTALAR A LINGUAGEM PL/pgSQL NO BANCO DE DADOS E PARTE
BÁSICA DO SISTEMA.
```

```
    AUTOR:    KAUI AIRES OLIVEIRA - ARQUITETO DE BANCO DE DADOS
```

```
    CONTATO: KAUIAIRES@GMAIL.COM
```

```
    CRIAÇÃO:   20/10/2006
```

```
    ATUALIZAÇÃO: 23/10/2006
```

```
    BANCO: POSTGRESQL
```

```
*/
```

```
--Procedimento de gatilho PL/pgSQL para auditoria
```

```
-- Este gatilho garante que todas as inserções, atualizações e exclusões de linha na tabela e são
registradas na tabela
```

```
-- tab_dado_auditado, para permitir auditar as operações efetuadas em uma tabela qualquer.
```

```
-- atualizar para o seu uso...
```

```
-- PARA O USO EM OUTRAS TABELAS TROCAR ONDE TEM O PALAVRA
```

```
"EMPREGADO" SUGIRO TROCAR PELO NOME DA TABELA QUE SERÁ AUDITADA
PARA FACILITAR DEPOIS A VISUALIZAÇÃO
```

```
CREATE OR REPLACE FUNCTION auditoria_banco.processa_empregado_audit() RETURNS
TRIGGER AS $empregado_audit$ -- AQUI TAMBÉM ALTERAR O NOME EMPREGADO
```

PARA O NOME CORRETO DA SUA FUNÇÃO

DECLARE

-- PARAMETROS DE VERIFICAÇÃO E VARIÁVEIS

iQtd integer; -- valor de retorno

auditar_delete integer = 1; -- se voce deseja auditar DELETE = 1 se não deseja = 0

auditar_update integer = 1; -- se voce deseja auditar UPDATE = 1 se não deseja = 0

auditar_insert integer = 1; -- se voce deseja auditar INSERT = 1 se não deseja = 0

SchemaName text = 'public'; -- Nome do Schema que está a tabela que Será

Auditada

TabName text = 'empregado'; -- Nome da Tabela que Será Auditada

BEGIN

--

-- VERIFICA A EXISTÊNCIA DA DEFINIÇÃO DA LINGUAGEM

--

SELECT count(*) FROM pg_catalog.pg_language INTO iQtd

WHERE lanname = 'plpgsql';

IF iQtd = 0 THEN

RAISE EXCEPTION 'NA AUDITORIA - A linguagem PLPGSQL ainda

não foi definida.';

END IF;

--

-- VERIFICA A EXISTÊNCIA DA TABELA TAB_DADO_AUDITADO

--

SELECT count(*) FROM pg_catalog.pg_tables INTO iQtd

WHERE tablename = 'tab_dado_auditado';

IF iQtd = 0 THEN

RAISE EXCEPTION 'NA AUDITORIA - A tabela tab_dado_auditado não

existe.';

END IF;

--

-- VERIFICA SE PARÂMETROS SÃO DIFERENTES

--

IF SchemaName = TabName THEN

RAISE EXCEPTION 'NA AUDITORIA - Deverá ser especificados parâmetros

diferentes.';

END IF;

--

-- VERIFICA A EXISTÊNCIA DA TABELA A SER AUDITADA

--

IF SchemaName IS NULL THEN

SELECT count(*) FROM pg_tables INTO iQtd

WHERE tablename = TabName;

ELSE

SELECT count(*) FROM pg_tables INTO iQtd

WHERE schemaname = SchemaName

AND tablename = TabName;

```

END IF;
IF iQtd = 0 THEN
    RAISE EXCEPTION 'NA AUDITORIA - A tabela % não existe.', SchemaName ||
TabName;
END IF;
--
-- SE PASSAR POR TODOS OS TESTES ACIMA CRIA NOSSO GATILHO PARA
MONITORAR
--
--
IF (TG_OP = 'DELETE') and auditar_delete = 1 THEN

    INSERT INTO auditoria_banco.tab_dado_auditado
    SELECT
        user, -- nome_usuario_banco
        now(), -- data_hora_acesso
        SchemaName || '.' || TabName, -- nome_tabela
        'DELETE', -- operacao_tabela
        inet_client_addr(), -- ip_maquina_acesso
        session_user; -- sessao_usuario

    RETURN OLD;
ELSIF (TG_OP = 'UPDATE') and auditar_update = 1 THEN

    INSERT INTO auditoria_banco.tab_dado_auditado
    SELECT
        user, -- nome_usuario_banco
        now(), -- data_hora_acesso
        SchemaName || '.' || TabName, -- nome_tabela
        'UPDATE', -- operacao_tabela
        inet_client_addr(), -- ip_maquina_acesso
        session_user; -- sessao_usuario

    RETURN NEW;
ELSIF (TG_OP = 'INSERT') and auditar_insert = 1 THEN

    INSERT INTO auditoria_banco.tab_dado_auditado
    SELECT
        user, -- nome_usuario_banco
        now(), -- data_hora_acesso
        SchemaName || '.' || TabName, -- nome_tabela
        'INSERT', -- operacao_tabela
        inet_client_addr(), -- ip_maquina_acesso
        session_user; -- sessao_usuario

    RETURN NEW;
END IF;
RETURN NULL; -- o resultado é ignorado uma vez que este é um gatilho AFTER
END;
Sempregado_audit$ language plpgsql; -- AQUI TAMBÉM ALTERAR O NOME EMPREGADO
PARA O NOME CORRETO DA SUA FUNÇÃO

```

```
CREATE TRIGGER empregado_audit -- AQUI TAMBÉM ALTERAR O NOME EMPREGADO
PARA O NOME CORRETO DA SUA FUNÇÃO
AFTER INSERT OR UPDATE OR DELETE ON public.empregado -- COLOQUE AQUI
TAMBÉM O NOME DO SCHEMA E A TABELA QUE SERÁ AUDITADA SEPARADO
SOMENTE PELO PONTO
    FOR EACH ROW EXECUTE PROCEDURE auditoria_banco.processa_empregado_audit(); --
NOME DA FUNÇÃO A EXECUTAR NO LUGAR DO EMPREGADO COLOCAR O NOME DA
FUNÇÃO
COMMIT;
ALTER FUNCTION auditoria_banco.processa_empregado_audit() OWNER TO auditoria_banco;
    -- AQUI TAMBÉM ALTERAR O NOME EMPREGADO PARA O NOME CORRETO DA
SUA FUNÇÃO

-- FIM
```

```
/*
    OBJETIVO:  INSTALAR A LINGUAGEM PL/pgSQL NO BANCO DE DADOS E PARTE
BÁSICA DO SISTEMA.
    AUTOR:    KAUI AIRES OLIVEIRA - ARQUITETO DE BANCO DE DADOS
    CONTATO:  KAUIAIRES@GMAIL.COM
    CRIAÇÃO:  20/10/2006
    ATUALIZAÇÃO: 23/10/2006
    BANCO:  POSTGRESQL
```

```
*/
-- PARA TESTARMOS COMO FICOU NOSSA AUDITORIA FAREMOS ALGUNS TESTES
EXECUTE ESTE SCRIPT
```

```
INSERT INTO public.empregado (nome_empregado, salario) VALUES ('João',1000);
INSERT INTO public.empregado (nome_empregado, salario) VALUES ('José',1500);
INSERT INTO public.empregado (nome_empregado, salario) VALUES ('Maria',250);
UPDATE public.empregado SET salario = 2500 WHERE nome_empregado = 'Maria';
DELETE FROM public.empregado WHERE nome_empregado = 'João';
```

```
-- DEPOIS VEJA COMO FICOU NOSSA TABELA EMPREGADO
```

```
SELECT * FROM public.empregado;
```

nome_empregado	salario
-----	-----
José	1500
Maria	2500

```
2 record(s) selected [Fetch MetaData: 0/ms] [Fetch Data: 32/ms]
```

-- E A SUA TABELA DE AUDITORIA DEVE TER FICADO ASSIM

```
SELECT * FROM auditoria_banco.tab_dado_auditado;
```

nome_usuario_banco	data_hora_acesso	nome_tabela	operacao_tabela	ip_maquina_acesso	sessao_usuario
-----	-----	-----	-----	-----	-----
kau_i_aires	24/10/2006 15:34	public.empregado	INSERT	172.25.0.200	
kau_i_aires					
kau_i_aires	24/10/2006 15:34	public.empregado	INSERT	172.25.0.200	
kau_i_aires					
kau_i_aires	24/10/2006 15:34	public.empregado	INSERT	172.25.0.200	
kau_i_aires					
kau_i_aires	24/10/2006 15:34	public.empregado	UPDATE	172.25.0.200	
kau_i_aires					
kau_i_aires	24/10/2006 15:34	public.empregado	DELETE	172.25.0.200	
kau_i_aires					

5 record(s) selected [Fetch MetaData: 16/ms] [Fetch Data: 16/ms]

Mais um exemplo de monitoração com triggers:

De: Hesley Py

Neste artigo vamos criar um sistema de "log" no PostgreSQL. O sistema deve armazenar em uma tabela log todas as alterações (inserção, atualização e deleção), a data em que ocorreram e o autor da alteração na tabela "alterada".

Para o nosso exercício precisamos criar as seguintes tabelas no banco:
Alterada (cod serial primary key, valor varchar(50))

```
create table alterada(  
    cod serial primary key,  
    valor varchar(50)  
);
```

Log (cod serial primary key, data date, autor varchar(20), alteracao varchar(6))

```
create table log(  
    cod serial primary key,  
    data date,  
    autor varchar(20),  
    alteracao varchar(6)  
);
```

Como veremos a seguir, como um procedimento armazenado comum, um procedimento de gatilho no PostgreSQL também é tratado como uma função. A diferença está no tipo de dados que essa função deve retornar, o tipo especial trigger.

Primeiro vamos criar o nosso procedimento armazenado (function) que será executado quando o evento ocorrer. A sintaxe é a mesma já vista nos artigos anteriores, a diferença, como já dito, é o tipo de retorno trigger. Para ver mais detalhes execute o comando \h create function no psql. A linguagem utilizada será a PL/pgSQL.

Nossa função de gatilho não deve receber nenhum parâmetro e deve retornar o tipo trigger.

```
create function gera_log() returns trigger as  
$$  
Begin  
    insert into log (data, autor, alteracao) values (now(), user, TG_OP);  
    return new;  
end;  
$$ language 'plpgsql';
```

Só pra lembrar, para criarmos uma função com a linguagem plpgsql, essa linguagem deve ter sido instalada em nosso banco através do comando "create language".

Quando uma função que retorna o tipo trigger é executada, o PostgreSQL cria algumas variáveis especiais na memória. Essas variáveis podem ser usadas no corpo das nossas funções. Na função acima estamos usando uma dessas variáveis, a TG_OP que retorna a operação que foi realizada (insert, update ou delete). Existem algumas outras variáveis muito

úteis como a new e a old. Minha intenção é abordá-las em um próximo artigo.

Para completar nosso exemplo, utilizamos as funções now e user para retornar data e hora da operação e o usuário logado respectivamente.

Agora precisamos criar o gatilho propriamente dito que liga a função à um evento ocorrido em uma tabela.

```
create trigger tr_gera_log after insert or update or delete on alterada
for each row execute procedure gera_log();
```

O comando acima cria um gatilho chamado tr_gera_log que deve ser executado após (after) as operações de insert, update ou delete na tabela "alterada". Para cada linha alterada deve ser executado o procedimento ou a função gera_log().

Para mais detalhes executar \h create trigger no psql.

Inserir registros na tabela e observar os resultados.

Mais Detalhes em:

<http://www.postgresql.org/docs/current/static/plpgsql-trigger.html>

<http://pgdocptbr.sourceforge.net/pg80/plpgsql-trigger.html>

http://imasters.uol.com.br/artigo/5033/postgresql/auditoria_em_banco_de_dados_postgresql/imprimir/

<http://conteudo.imasters.com.br/5033/01.rar> (arquivo exemplo)

<http://www.devmedia.com.br/articles/viewcomp.asp?comp=7032>

```
CREATE TRIGGER name { BEFORE | AFTER } { event [OR ...] }
ON table FOR EACH { ROW | STATEMENT }
EXECUTE PROCEDURE func ( arguments )
```

```
CREATE FUNCTION timetrigger() RETURNS opaque AS '
BEGIN
    UPDATE shop SET sold="now" WHERE sold IS NULL;
    RETURN NEW;
END;
' LANGUAGE 'plpgsql';
```

```
CREATE TRIGGER inserttime AFTER INSERT
ON shop FOR EACH ROW EXECUTE
PROCEDURE timetrigger();
```

```
INSERT INTO shop (prodname,price,amount)
VALUES('Knuth's Biography',39,1);
```

```
SELECT * FROM shop ;
```

Variáveis criadas automaticamente:

NEW holds the new database row on INSERT/UPDATE statements on row-level triggers. The

datatype of NEW is record.

OLD contains the old database row on UPDATE/DELETE operations on row-level triggers. The datatype is record.

TG_NAME contains the name of the trigger that is actually fired (datatype name).

TG_WHEN contains either AFTER or BEFORE (see the syntax overview of triggers shown earlier).

TG_LEVEL tells whether the trigger is a ROW or a STATEMENT trigger (datatype text).

TG_OP tells which operation the current trigger has been fired for (INSERT, UPDATE, or DELETE; datatype text).

TG_RELID contains the object ID (datatype oid), and TG_RELNAME (datatype name) contains the name of the table the trigger is fired for.

TG_RELNAME contains the name of the table that caused the trigger invocation. The datatype of the return value is name.

TG_ARGV[] contains the arguments from the CREATE TRIGGER statement in an array of text.

TG_NARGS is used to store the number of arguments passed to the trigger in the CREATE TRIGGER statement. The arguments themselves are stored in an array called TG_ARGV[] (datatype array of text). TG_ARGV[] is indexed starting with 0.

RULES

O comando CREATE RULE cria uma regra aplicada à tabela ou view especificada.

Uma regra faz com que comandos adicionais sejam executados quando um determinado comando é executado em uma determinada tabela.

É importante perceber que a regra é, na realidade, um mecanismo de transformação de comando, ou uma macro de comando.

É possível criar a ilusão de uma view atualizável definindo regras ON INSERT, ON UPDATE e ON DELETE, ou qualquer subconjunto destas que seja suficiente para as finalidades desejadas, para substituir as ações de atualização na visão por atualizações apropriadas em outras tabelas.

Existe algo a ser lembrado quando se tenta utilizar rules condicionais para atualização de visões: é obrigatório haver uma rule incondicional INSTEAD para cada ação que se deseja permitir na visão. Se a rule for condicional, ou não for INSTEAD, então o sistema continuará a rejeitar as tentativas de realizar a ação de atualização, porque acha que poderá acabar tentando realizar a ação sobre a tabela fictícia da visão em alguns casos.

banco=# \h create rule

Comando: CREATE RULE

Descrição: define uma nova regra de reescrita

Sintaxe:

CREATE [OR REPLACE] RULE nome AS ON evento

TO tabela [WHERE condição]

DO [ALSO | INSTEAD] { NOTHING | comando | (comando ; comando ...) }

O comando CREATE RULE cria uma rule aplicada à tabela ou visão especificada.

evento

Evento é um entre SELECT, INSERT, UPDATE e DELETE.

condição

Qualquer expressão condicional SQL (retornando boolean). A expressão condicional não pode fazer referência a nenhuma tabela, exceto NEW e OLD, e não pode conter funções de agregação.

INSTEAD

INSTEAD indica que os comandos devem ser executados em vez dos (instead of) comandos originais.

ALSO

ALSO indica que os comandos devem ser executados adicionalmente aos comandos originais.

Se não for especificado nem ALSO nem INSTEAD, ALSO é o padrão.

comando

O comando ou comandos que compõem a ação da regra. Os comandos válidos são SELECT, INSERT, UPDATE, DELETE e NOTIFY.

Dentro da condição e do comando, os nomes especiais de tabela NEW e OLD podem ser usados para fazer referência aos valores na tabela referenciada. O NEW é válido nas regras ON INSERT e ON UPDATE, para fazer referência à nova linha sendo inserida ou atualizada. O OLD é válido nas regras ON UPDATE e ON DELETE, para fazer referência à linha existente sendo atualizada ou excluída.

Obs.: É necessário possuir o privilégio RULE na tabela para poder definir uma regra para a mesma.

Exemplos:

```
CREATE RULE me_notifique AS ON UPDATE TO datas DO ALSO NOTIFY datas;
```

```
CREATE RULE r1 AS ON INSERT TO TBL1 DO  
(INSERT INTO TBL2 VALUES (new.i); NOTIFY TBL2);
```

```
CREATE RULE "_RETURN" AS ON SELECT TO minha_visão DO INSTEAD  
SELECT * FROM minha_tabela;      -- Ao invés de selecionar da visão seleciona da  
tabela.
```

Exemplo de uso prático de Rule

Existe uma tabela `shoelace_data` que queremos monitorar e guardar os relatórios de alterações do campo `sl_avail` na tabela `shoelace_log`. Para isso criaremos uma rule que gravará um registro na tabela `shoelace_log` sempre que o campo `sl_avail` for alterado em `shoelace_data` (vide estrutura abaixo).

Say we want to trace changes to the `sl_avail` column in the `shoelace_data` relation. So we set up a log table and a rule that conditionally writes a log entry when an UPDATE is performed on `shoelace_data`.

```
CREATE TABLE shoelace_data (  
    sl_name    text,          -- primary key  
    sl_avail   integer,       -- available number of pairs  
    sl_color   text,          -- shoelace color  
    sl_len     real,           -- shoelace length  
    sl_unit    text           -- length unit  
);
```

```
CREATE TABLE shoelace_log (  
    sl_name    text,          -- shoelace changed  
    sl_avail   integer,       -- new available value  
    log_who    text,          -- who did it  
);
```

```

    log_when    timestamp    -- when
);

CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
    WHERE NEW.sl_avail <> OLD.sl_avail
    DO INSERT INTO shoelace_log VALUES (
        NEW.sl_name,
        NEW.sl_avail,
        current_user,
        current_timestamp
    );

```

Vejaamos o conteúdo da tabela shoelace_log:

```
SELECT * FROM shoelace_log;
```

Atualizemos a tabela shoelace_data alterando o campo sl_avail:

```
UPDATE shoelace_data SET sl_avail = 6 WHERE sl_name = 'sl7';
```

Vejaamos novamente o que ocorreu na tabela shoelace_log:

```
SELECT * FROM shoelace_log;
```

sl_name	sl_avail	log_who	log_when
sl7	6	Al	Tue Oct 20 16:14:45 1998 MET DST

(1 row)

Veja que agora ela armazena as informações que indicamos.

Crie um exemplo similar para uso de rules.

15) Usando Matrizes/Arrays no PostgreSQL

O PostgreSQL permite que colunas de uma tabela sejam definidas como matrizes (*arrays*) multidimensionais de comprimento variável. Podem ser criadas matrizes de qualquer tipo de dado, nativo ou definido pelo usuário (Entretanto, ainda não são suportadas matrizes de tipos compostos ou domínios).

Declaração do tipo matriz

Para ilustrar a utilização do tipo matriz, é criada a tabela abaixo:

```
CREATE TABLE sal_emp (  
    nome                text,  
    pagamento_semanal  integer[],  
    agenda              text[][]  
);
```

Array assim [] significa tamanho indefinido.

Conforme visto, o tipo de dado matriz é identificado anexando colchetes ([]) ao nome do tipo de dado dos elementos da matriz. O comando acima cria uma tabela chamada sal_emp, contendo uma coluna do tipo text (nome), uma matriz unidimensional do tipo integer (pagamento_semanal), que representa o salário semanal do empregado, e uma matriz bidimensional do tipo text (agenda), que representa a agenda semanal do empregado.

A sintaxe de CREATE TABLE permite especificar o tamanho exato da matriz como, por exemplo:

```
CREATE TABLE jogo_da_velha (  
    casa      integer[3][3]  
);
```

Entretanto, a implementação atual não obriga que os limites de tamanho da matriz sejam respeitados — o comportamento é o mesmo das matrizes com comprimento não especificado.

Na verdade, a implementação atual também não obriga que o número de dimensões declarado seja respeitado. As matrizes de um determinado tipo de elemento são todas consideradas como sendo do mesmo tipo, não importando o tamanho ou o número de dimensões. Portanto, a declaração do número de dimensões ou tamanhos no comando CREATE TABLE é simplesmente uma documentação, que não afeta o comportamento em tempo de execução.

Pode ser utilizada uma sintaxe alternativa para matrizes unidimensionais, em conformidade com o padrão SQL:1999. A coluna pagamento_semanal pode ser definida como:

```
pagamento_semanal  integer ARRAY[4],
```

Esta sintaxe requer uma constante inteira para designar o tamanho da matriz. Entretanto, como anteriormente, o PostgreSQL não obriga que os limites de tamanho da matriz sejam respeitados.

Entrada de valor matriz

Para escrever um valor matriz como uma constante literal, os valores dos elementos devem ser

envoltos por chaves ({} e separados por vírgulas (Quem conhece C pode ver que não é diferente da sintaxe da linguagem C para inicializar estruturas). Podem ser colocadas aspas (") em torno de qualquer valor de elemento, sendo obrigatório caso o elemento contenha vírgulas ou chaves (abaixo são mostrados mais detalhes). Portanto, o formato geral de uma constante matriz é o seguinte:

```
'{ val1 delim val2 delim ... }'
```

onde delim é o caractere delimitador para o tipo, conforme registrado na sua entrada em pg_type. Entre todos os tipos de dado padrão fornecidos na distribuição do PostgreSQL, o tipo box usa o ponto-e-vírgula (;) mas todos os demais usam a vírgula (,). Cada val é uma constante do tipo do elemento da matriz, ou uma submatriz. Um exemplo de uma constante matriz é

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

Esta constante é uma matriz bidimensional, 3 por 3, formada por três submatrizes de inteiros.

(Estes tipos de constante matriz são, na verdade, apenas um caso especial do tipo genérico de constantes mostrado na [Seção 4.1.2.5](#). A constante é inicialmente tratada como uma cadeia de caracteres e passada para a rotina de conversão de entrada de matriz. Pode ser necessária uma especificação explícita do tipo).

Agora podemos ver alguns comandos INSERT.

```
INSERT INTO sal_emp
VALUES ('Bill',
       '{10000, 10000, 10000, 10000}',
       '{{"reunião", "almoço"}, {"reunião"}}');
```

ERRO: matrizes multidimensionais devem ter expressões de matriz com dimensões correspondentes

Deve ser observado que as matrizes multidimensionais devem possuir tamanhos correspondentes para cada dimensão. Uma combinação errada causa uma mensagem de erro.

```
INSERT INTO sal_emp
VALUES ('Bill',
       '{10000, 10000, 10000, 10000}',
       '{{"reunião", "almoço"}, {"treinamento", "apresentação"}}');
```

```
INSERT INTO sal_emp
VALUES ('Carol',
       '{20000, 25000, 25000, 25000}',
       '{{"café da manhã", "consultoria"}, {"reunião", "almoço"}}');
```

Uma limitação da implementação atual de matriz, é que os elementos individuais da matriz não podem ser valores nulos SQL. Toda a matriz pode ser definida como nula, mas não pode existir uma matriz com alguns elementos nulos e outros não.

O resultado das duas inserções anteriores se parece com:

```
SELECT * FROM sal_emp;
```

nome	pagamento_semanal	agenda
Bill	{10000,10000,10000,10000}	{{reunião,almoço}, {treinamento,apresentação}}
Carol	{20000,25000,25000,25000}	{{"café da manhã",consultoria}, {"reunião", "almoço"}}

```
{reunião, almoço}}
(2 linhas)
```

Também pode ser utilizada a sintaxe de construtor de ARRAY:

```
INSERT INTO sal_emp
VALUES ('Bill',
ARRAY[10000, 10000, 10000, 10000],
ARRAY[['reunião', 'almoço'], ['treinamento', 'apresentação']]);

INSERT INTO sal_emp
VALUES ('Carol',
ARRAY[20000, 25000, 25000, 25000],
ARRAY[['café da manhã', 'consultoria'], ['reunião', 'almoço']]);
```

Deve ser observado que os elementos da matriz são constantes comuns ou expressões SQL; por exemplo, os literais cadeia de caracteres ficam entre apóstrofes, em vez de aspas como no caso de um literal matriz. A sintaxe do construtor de ARRAY é mostrada com mais detalhes na [Seção 4.2.10](#).

Acesso às matrizes

Agora podemos efetuar algumas consultas na tabela. Primeiro, será mostrado como acessar um único elemento da matriz de cada vez. Esta consulta mostra os nomes dos empregados cujo pagamento mudou na segunda semana:

```
SELECT nome FROM sal_emp WHERE pagamento_semanal[1] <> pagamento_semanal[2];

nome
-----
Carol
(1 linha)
```

Os números dos índices da matriz são escritos entre colchetes. Por padrão, o PostgreSQL utiliza a convenção de numeração baseada em um, ou seja, uma matriz de n elementos começa por array[1] e termina por array[n].

Esta consulta mostra o pagamento da terceira semana de todos os empregados:

```
SELECT pagamento_semanal[3] FROM sal_emp;

pagamento_semanal
-----
10000
25000
(2 linhas)
```

Também é possível acessar faixas retangulares arbitrárias da matriz, ou submatrizes. Uma faixa da matriz é especificada escrevendo limite-inferior:limite-superior para uma ou mais dimensões da matriz. Por exemplo, esta consulta mostra o primeiro item na agenda do Bill para os primeiros dois dias da semana:

```
SELECT agenda[1:2][1:1] FROM sal_emp WHERE nome = 'Bill';

agenda
-----
```

```
{{reunião},{treinamento}}
(1 linha)
```

Também pode ser escrito

```
SELECT agenda[1:2][1] FROM sal_emp WHERE nome = 'Bill';
```

para obter o mesmo resultado. Uma operação com índices em matriz é sempre considerada como representando uma faixa da matriz, quando qualquer um dos índices estiver escrito na forma inferior:superior. O limite inferior igual a 1 é assumido para qualquer índice quando for especificado apenas um valor, como neste exemplo:

```
SELECT agenda[1:2][2] FROM sal_emp WHERE nome = 'Bill';
```

```

              agenda
-----
{{reunião,almoço},{treinamento,apresentação}}
(1 linha)
```

Podem ser obtidas as dimensões correntes de qualquer valor matriz através da função `array_dims`:

```
SELECT array_dims(agenda) FROM sal_emp WHERE nome = 'Carol';
```

```

array_dims
-----
[1:2][1:2]
(1 linha)
```

A função `array_dims` produz um resultado do tipo text, conveniente para as pessoas lerem mas, talvez, nem tão conveniente para os programas. As dimensões também podem ser obtidas através das funções `array_upper` e `array_lower`, que retornam os limites superior e inferior da dimensão especificada da matriz, respectivamente.

```
SELECT array_upper(agenda, 1) FROM sal_emp WHERE nome = 'Carol';
```

```

array_upper
-----
                2
(1 linha)
```

Modificação de matrizes

Um valor matriz pode ser inteiramente substituído utilizando:

```
UPDATE sal_emp SET pagamento_semanal = '{25000,25000,27000,27000}'
WHERE nome = 'Carol';
```

ou utilizando a sintaxe com a expressão `ARRAY`:

```
UPDATE sal_emp SET pagamento_semanal = ARRAY[25000,25000,27000,27000]
WHERE nome = 'Carol';
```

Também pode ser atualizado um único elemento da matriz:

```
UPDATE sal_emp SET pagamento_semanal[4] = 15000
WHERE nome = 'Bill';
```

ou pode ser atualizada uma faixa da matriz:

```
UPDATE sal_emp SET pagamento_semanal[1:2] = '{27000,27000}'
WHERE nome = 'Carol';
```

Um valor matriz armazenado pode ser ampliado fazendo atribuição a um elemento adjacente aos já presentes, ou fazendo atribuição a uma faixa que é adjacente ou se sobrepõe aos dados já presentes. Por exemplo, se a matriz `minha_matriz` possui atualmente quatro elementos, esta matriz terá cinco elementos após uma atualização que faça uma atribuição a `minha_matriz[5]`. Atualmente, as ampliações desta maneira somente são permitidas para matrizes unidimensionais, não sendo permitidas em matrizes multidimensionais.

A atribuição de faixa de matriz permite a criação de matrizes que não utilizam índices baseados em um. Por exemplo, pode ser feita a atribuição `minha_matriz[-2:7]` para criar uma matriz onde os valores dos índices variam de -2 a 7.

Também podem ser construídos novos valores matriz utilizando o operador de concatenação `||`.

```
SELECT ARRAY[1,2] || ARRAY[3,4];
```

```

?column?
-----
{1,2,3,4}
(1 linha)
```

```
SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]];
```

```

?column?
-----
{{5,6},{1,2},{3,4}}
(1 linha)
```

O operador de concatenação permite colocar um único elemento no início ou no fim de uma matriz unidimensional. Aceita, também, duas matrizes N-dimensionais, ou uma matriz N-dimensional e outra N+1-dimensional.

Quando é colocado um único elemento no início de uma matriz unidimensional, o resultado é uma matriz com o limite inferior do índice igual ao limite inferior do índice do operando à direita, menos um. Quando um único elemento é colocado no final de uma matriz unidimensional, o resultado é uma matriz mantendo o limite inferior do operando à esquerda. Por exemplo:

```
SELECT ARRAY[2,3];
```

```

array
-----
{2,3}
(1 linha)
```

```
SELECT array_dims(ARRAY[2,3]);
```

```

array_dims
-----
[1:2]
(1 linha)
```

```
-- Adicionar no início da matriz
```



```
SELECT 1 || ARRAY[2,3];

?column?
-----
{1,2,3}
(1 linha)

SELECT array_dims(1 || ARRAY[2,3]);

array_dims
-----
[0:2]
(1 linha)
```

-- Adicionar no final da matriz

```
SELECT ARRAY[1,2] || 3;

?column?
-----
{1,2,3}
(1 linha)

SELECT array_dims(ARRAY[1,2] || 3);

array_dims
-----
[1:3]
(1 linha)
```

Quando duas matrizes com o mesmo número de dimensões são concatenadas, o resultado mantém o limite inferior do índice da dimensão externa do operando à esquerda. O resultado é uma matriz contendo todos os elementos do operando à esquerda seguido por todos os elementos do operando à direita. Por exemplo:

-- Concatenação de matrizes unidimensionais

```
SELECT ARRAY[1,2] || ARRAY[3,4,5];

?column?
-----
{1,2,3,4,5}
(1 linha)

SELECT array_dims(ARRAY[1,2] || ARRAY[3,4,5]);

array_dims
-----
[1:5]
(1 linha)
```

-- Concatenação de matrizes bidimensionais

```
SELECT ARRAY[[1,2],[3,4]] || ARRAY[[5,6],[7,8],[9,0]];

?column?
-----
{{1,2},{3,4},{5,6},{7,8},{9,0}}
```

```
(1 linha)
```

```
SELECT array_dims (ARRAY[[1,2],[3,4]] || ARRAY[[5,6],[7,8],[9,0]]);
```

```
array_dims
-----
[1:5][1:2]
(1 linha)
```

Quando uma matriz N-dimensional é colocada no início ou no final de uma matriz N+1-dimensional, o resultado é análogo ao caso da matriz elemento acima. Cada submatriz N-dimensional se torna essencialmente um elemento da dimensão externa da matriz N+1-dimensional. Por exemplo:

```
-- Exemplo de matriz unidimensional concatenada com matriz bidimensional
```

```
SELECT ARRAY[1,2] || ARRAY[[3,4],[5,6]];
```

```
?column?
-----
{{1,2},{3,4},{5,6}}
(1 linha)
```

```
SELECT array_dims (ARRAY[1,2] || ARRAY[[3,4],[5,6]]);
```

```
array_dims
-----
[0:2][1:2]
(1 linha)
```

```
-- Exemplo de matriz bidimensional concatenada com matriz tridimensional (N. do T.)
```

```
SELECT ARRAY[[-1,-2],[-3,-4]];
```

```
array
-----
{{-1,-2},{-3,-4}}
(1 linha)
```

```
SELECT array_dims (ARRAY[[-1,-2],[-3,-4]]);
```

```
array_dims
-----
[1:2][1:2]
(1 linha)
```

```
SELECT ARRAY[[[5,6],[7,8]],[[9,10],[11,12]],[[13,14],[15,16]]];
```

```
array
-----
{{{5,6},{7,8}},{9,10},{11,12}},{13,14},{15,16}}}
(1 linha)
```

```
SELECT array_dims (ARRAY[[[5,6],[7,8]],[[9,10],[11,12]],[[13,14],[15,16]]]);
```

```
array_dims
-----
```

```
[1:3][1:2][1:2]
(1 linha)
```

```
SELECT ARRAY[[-1,-2],[-3,-4]] ||
        ARRAY[[5,6],[7,8]],[9,10],[11,12]],[[13,14],[15,16]]];
```

```
?column?
```

```
-----
{{{[-1,-2],[-3,-4]},{[5,6],[7,8]},{[9,10],[11,12]},{[13,14],[15,16]}}}
(1 linha)
```

```
SELECT array_dims(ARRAY[[-1,-2],[-3,-4]] ||
                    ARRAY[[5,6],[7,8]],[9,10],[11,12]],[[13,14],[15,16]]);
```

```
array_dims
```

```
-----
[0:3][1:2][1:2]
(1 linha)
```

```
SELECT ARRAY[[5,6],[7,8]],[9,10],[11,12]],[[13,14],[15,16]] ||
        ARRAY[[-1,-2],[-3,-4]];
```

```
?column?
```

```
-----
{{{[5,6],[7,8]},{[9,10],[11,12]},{[13,14],[15,16]},{[-1,-2],[-3,-4]}}}
(1 linha)
```

```
SELECT array_dims(ARRAY[[5,6],[7,8]],[9,10],[11,12]],[[13,14],[15,16]] ||
                    ARRAY[[-1,-2],[-3,-4]]);
```

```
array_dims
```

```
-----
[1:4][1:2][1:2]
(1 linha)
```

Uma matriz também pode ser construída utilizando as funções `array_prepend`, `array_append` e `array_cat`. As duas primeiras suportam apenas matrizes unidimensionais, mas `array_cat` suporta matrizes multidimensionais. Deve ser observado que é preferível utilizar o operador de concatenação mostrado acima, em vez de usar diretamente estas funções. Na verdade, estas funções têm seu uso principal na implementação do operador de concatenação. Entretanto, podem ser úteis na criação de agregações definidas pelo usuário. Alguns exemplos:

```
SELECT array_prepend(1, ARRAY[2,3]);
```

```
array_prepend
```

```
-----
{1,2,3}
(1 linha)
```

```
SELECT array_append(ARRAY[1,2], 3);
```

```
array_append
```

```
-----
{1,2,3}
(1 linha)
```

```
SELECT array_cat(ARRAY[1,2], ARRAY[3,4]);
```

```

array_cat
-----
{1,2,3,4}
(1 linha)

SELECT array_cat (ARRAY[[1,2],[3,4]], ARRAY[5,6]);

array_cat
-----
{{1,2},{3,4},{5,6}}
(1 linha)

SELECT array_cat (ARRAY[5,6], ARRAY[[1,2],[3,4]]);

array_cat
-----
{{5,6},{1,2},{3,4}}

```

Procura em matrizes

Para procurar um valor em uma matriz deve ser verificado cada valor da matriz. Pode ser feito à mão, se for conhecido o tamanho da matriz. Por exemplo:

```

SELECT * FROM sal_emp WHERE pagamento_semanal[1] = 10000 OR
                             pagamento_semanal[2] = 10000 OR
                             pagamento_semanal[3] = 10000 OR
                             pagamento_semanal[4] = 10000;

```

Entretanto, em pouco tempo se torna entediante para matrizes grandes, e não servirá se a matriz for de tamanho desconhecido. Um método alternativo está descrito na [Seção 9.17](#). A consulta acima pode ser substituída por:

```

SELECT * FROM sal_emp WHERE 10000 = ANY (pagamento_semanal);

```

Além disso, podem ser encontradas as linhas onde a matriz possui todos os valores iguais a 10000 com:

```

SELECT * FROM sal_emp WHERE 10000 = ALL (pagamento_semanal);

```

Dica: Matrizes não são conjuntos; a procura por determinados elementos da matriz pode ser um sinal de um banco de dados mal projetado. Considere a utilização de uma outra tabela, com uma linha para cada item que seria um elemento da matriz. Assim é mais fácil procurar e, provavelmente, vai se comportar melhor com um número grande de elementos.

Sintaxe de entrada e de saída das matrizes

A representação textual externa de um valor matriz é formada por itens que são interpretados de acordo com as regras de conversão de I/O para o tipo do elemento da matriz, mais os adornos que indicam a estrutura da matriz. Estes adornos consistem em chaves ({ e }) em torno do valor matriz, mais os caracteres delimitadores entre os itens adjacentes. O caractere delimitador geralmente é a vírgula (,), mas pode ser outro: é determinado pela definição de typdelim para o tipo do elemento da

matriz (Entre os tipos de dado padrão fornecidos na distribuição do PostgreSQL o tipo box utiliza o ponto-e-vírgula (;), mas todos os outros utilizam a vírgula). Em uma matriz multidimensional cada dimensão (linha, plano, cubo, etc.) recebe seu nível próprio de chaves, e os delimitadores devem ser escritos entre entidades de chaves adjacentes do mesmo nível.

A rotina de saída de matriz coloca aspas em torno dos valores dos elementos caso estes sejam cadeias de caracteres vazias, ou se contenham chaves, caracteres delimitadores, aspas, contrabarras, ou espaços em branco. Aspas e contrabarras incorporadas aos valores dos elementos recebem o escape de contrabarra. No caso dos tipos de dado numéricos é seguro assumir que as aspas nunca vão estar presentes, mas para tipos de dado textuais deve-se estar preparado para lidar tanto com a presença quanto com a ausência das aspas (Esta é uma mudança de comportamento com relação às versões do PostgreSQL anteriores a 7.2).

Por padrão, o limite inferior do valor do índice de cada dimensão da matriz é definido como um. Se alguma das dimensões da matriz tiver um limite inferior diferente de um, um adorno adicional indicando as verdadeiras dimensões da matriz precede o adorno da estrutura da matriz. Este adorno é composto por colchetes ([]) em torno de cada limite inferior e superior da dimensão da matriz, com o caractere delimitador dois-pontos (:) entre estes. O adorno de dimensão da matriz é seguido pelo sinal de igual (=). Por exemplo:

```
SELECT 1 || ARRAY[2,3] AS array;
```

```

      array
-----
 [0:2]={1,2,3}
(1 linha)
```

```
SELECT ARRAY[1,2] || ARRAY[[3,4]] AS array;
```

```

      array
-----
 [0:1][1:2]={ {1,2}, {3,4} }
(1 linha)
```

Esta sintaxe também pode ser utilizada para especificar índices de matriz não padrão em um literal matriz. Por exemplo:

```
SELECT f1[1][-2][3] AS e1, f1[1][-1][5] AS e2
FROM (SELECT '[1:1][-2:-1][3:5]={{{1,2,3},{4,5,6}}}'::int[] AS f1) AS ss;
```

```

e1 | e2
---+---
 1 |  6
(1 linha)
```

Conforme mostrado anteriormente, ao escrever um valor matriz pode-se colocar aspas em torno de qualquer elemento individual da matriz. Isto *deve* ser feito se o valor do elemento puder, de alguma forma, confundir o analisador de valor matriz. Por exemplo, os elementos contendo chaves, vírgulas (ou qualquer que seja o caractere delimitador), aspas, contrabarras ou espaços em branco na frente ou atrás devem estar entre aspas. Para colocar aspas ou contrabarras no valor entre aspas do elemento da matriz, estes devem ser precedidos por uma contrabarra. Como alternativa, pode ser utilizado o escape de contrabarra para proteger qualquer caractere de dado que seria de outra forma considerado como sintaxe da matriz.

Podem ser escritos espaços em branco antes do abre chaves ou após o fecha chaves. Também

podem ser escritos espaços em branco antes ou depois de qualquer item individual cadeia de caracteres. Em todos estes casos os espaços em branco são ignorado. Entretanto, espaços em branco dentro de elementos entre aspas, ou envoltos nos dois lados por caracteres de um elemento que não são espaços em branco, não são ignorados.

Nota: Lembre-se que o que se escreve em um comando SQL é interpretado primeiro como um literal cadeia de caracteres e, depois, como uma matriz. Isto duplica o número de contrabarras necessárias. Por exemplo, para inserir um valor matriz do tipo text contendo uma contrabarra e uma aspa, deve ser escrito

```
INSERT ... VALUES ('{"\\\\"", "\\\""}');
```

O processador de literais cadeias de caracteres remove um nível de contrabarras, portanto o que chega para o analisador de valor matriz se parece com {"\\", "\\\""} . Por sua vez, as cadeias de caracteres introduzidas na rotina de entrada do tipo de dado text se tornam \ e " , respectivamente (Se estivéssemos trabalhando com um tipo de dado cuja rotina de entrada também tratasse as contrabarras de forma especial como, por exemplo, bytea, seriam necessárias oito contrabarras no comando para obter uma contrabarra armazenada no elemento da matriz). Pode ser utilizada a delimitação por cifrão (dollar quoting) (consulte a [Seção 4.1.2.2](#)) para evitar a necessidade de duplicar as contrabarras.

Dica: Ao se escrever valores matrizes nos comandos SQL, geralmente é mais fácil trabalhar com a sintaxe do construtor de ARRAY (consulte a [Seção 4.2.10](#)) do que com a sintaxe do literal cadeia de caracteres. Em ARRAY, os valores dos elementos individuais são escritos da mesma maneira como seriam escritos caso não fossem membros de uma matriz.

Fonte: <http://pgdocptbr.sourceforge.net/pg80/arrays.html>

Funções e operadores para matrizes

A [Tabela 9-37](#) mostra os operadores disponíveis para o tipo array.

Tabela 9-37. Operadores para o tipo array

Operador	Descrição	Exemplo	Resultado
=	igual	ARRAY[1.1,2.1,3.1]::int[] = ARRAY[1,2,3]	t
<>	diferente	ARRAY[1,2,3] <> ARRAY[1,2,4]	t
<	menor	ARRAY[1,2,3] < ARRAY[1,2,4]	t
>	maior	ARRAY[1,4,3] > ARRAY[1,2,4]	t
<=	menor ou igual	ARRAY[1,2,3] <= ARRAY[1,2,3]	t
>=	maior ou igual	ARRAY[1,4,3] >= ARRAY[1,4,3]	t
	concatenação de matriz com matriz	ARRAY[1,2,3] ARRAY[4,5,6]	{1,2,3,4,5,6}
	concatenação de matriz com matriz	ARRAY[1,2,3] ARRAY[[4,5,6], [7,8,9]]	{{1,2,3},{4,5,6},{7,8,9}}
	concatenação de elemento com matriz	3 ARRAY[4,5,6]	{3,4,5,6}
	concatenação de matriz com elemento	ARRAY[4,5,6] 7	{4,5,6,7}

A [Tabela 9-38](#) mostra as funções disponíveis para uso com o tipo array.

Tabela 9-38. Funções para o tipo array

Função	Tipo retornado	Descrição	Exemplo	Resultado
array_cat (anyarray, anyarray)	anyarray	concatena duas matrizes	array_cat(ARRAY[1,2,3], ARRAY[4,5])	{1,2,3,4,5}
array_append (anyarray, anyelement)	anyarray	anexa um elemento no final de uma matriz	array_append(ARRAY[1,2], 3)	{1,2,3}
array_prepend (anyelement, anyarray)	anyarray	anexa um elemento no início de uma matriz	array_prepend(1, ARRAY[2,3])	{1,2,3}

Função	Tipo retornado	Descrição	Exemplo	Resultado
array_dims (anyarray)	text	retorna a representação textual das dimensões da matriz	array_dims(array[[1,2,3], [4,5,6]])	[1:2][1:3]
array_lower (anyarray, integer)	integer	retorna o limite inferior da dimensão especificada da matriz	array_lower(array_prepend(0, ARRAY[1,2,3]), 1)	0
array_upper (anyarray, integer)	integer	retorna o limite superior da dimensão especificada da matriz	array_upper(ARRAY[1,2,3,4], 1)	4
array_to_string (anyarray, text)	text	concatena os elementos da matriz utilizando o delimitador especificado	array_to_string(array[1, 2, 3], '^~^')	1^~^2^~^3
string_to_array (text, text)	text[]	divide uma cadeia de caracteres em elementos de matriz utilizando o delimitador especificado		

Fonte: <http://pgdocptbr.sourceforge.net/pg80/functions-array.html>

16) Utilizando o Comando CASE ... WHEN

CASE

Cláusula que economiza múltiplas linhas de programação. Usada em SELECT e UPDATE. É possível utilizá-la em situações de teste (true/false).

Um único comando testa vários registros.

Select

```
select nome, preco
  case
    when preco < 10 then preco * 0.9
    when preco >= 10 and preco < 13 then preco * 0.8
  else preco * 0.7
end desconto -- desconto aqui será o rótulo
from produtos;
```

Update

```
update produtos
  set preco =
    case
      when preco < 10 then preco * 0.9
      when preco >= 10 and preco < 13 then preco * 0.8
    else preco * 0.7
  end;
```

Case compacto

Colocar o campo após o case.

```
select nome,
  case codigo_fornecedor
    when 1 then 'Laranja'
    when 2 then 'Banana'
    when 3 then 'Goiaba'
  end
from produtos;
```

Observar que toda a expressão do case retorna apenas um único valor sempre que for executada.

CASE é uma expressão condicional do SQL. Uma estrutura semelhante ao IF das linguagens de programação. Caso WHEN algo THEN isso. Vários WHEN podem vir num único CASE. Finaliza com END.

```
CASE WHEN condição THEN resultado
      WHEN condição2 THEN resultado
      [WHEN ...]
      [ELSE resultado]
END
```

Obs.: O que vem entre condeletes é opcional.

Mais detalhes em: <http://pgdocptbr.sourceforge.net/pg80/functions-conditional.html>

William Leite Araújo na lista pgbr-geral:
Ambas as formas são válidas. Você pode usar o

```
SELECT CASE WHEN [teste] THEN ... ELSE [saida] END;
ou
SELECT CASE [coluna]
      WHEN [valor1] THEN resultado
      WHEN [valor2] THEN resultado
      ...
      ELSE [saida] END
```

Quando o teste é entre 2 valores, a primeira forma é a mais aplicável. Quando quer se diferenciar mais de um valor de uma mesma coluna, a segunda é a mais apropriada. Por exemplo:

```
SELECT CASE tipo_credito WHEN 'S' THEN 'Salário' WHEN 'P' THEN 'Pró-labore'
      WHEN 'D' THEN 'Depósito' ELSE 'Outros' END as tipo_credito;
```

```
SELECT nome, CASE WHEN EXISTS (SELECT nome FROM clientes WHERE
      nome=c.nome)
      THEN 'sim'
      ELSE 'não'
      END AS cliente
FROM clientes c;
```

Trazer o nome sempre que existir o nome do cliente.

IN
SELECT nome, CASE WHEN nome IN (SELECT nome FROM clientes)
 THEN 'sim'

```
ELSE 'não'
END AS cliente
FROM clientes;
```

NOT IN e ANY/SOME

```
SELECT cpf_cliente, CASE WHEN cpf_cliente = ANY (SELECT cpf_cliente FROM
pedidos)
THEN 'sim'
ELSE 'não'
END AS cliente
FROM pedidos;
```

Trazer o CPF

Outros Exemplos:

```
create database dml;
\c dml
create table amigos(
codigo serial primary key,
nome char(45),
idade int
);
```

```
insert into amigos (nome, idade) values ('João Brito', 25);
insert into amigos (nome, idade) values ('Roberto', 35);
insert into amigos (nome, idade) values ('Antônio', 15);
insert into amigos (nome, idade) values ('Francisco Queiroz', 23);
insert into amigos (nome, idade) values ('Bernardo dos Santos', 21);
insert into amigos (nome, idade) values ('Francisca Pinto', 22);
insert into amigos (nome, idade) values ('Natanael', 55);
```

```
select nome,
idade,
case
when idade >= 21 then 'Adulto'
else 'Menor'
end as status
from amigos order by nome;
```

-- CASE WHEN cria uma coluna apenas para exibição

```
create table amigos2(
codigo serial primary key,
nome char(45),
estado char(2)
);
```

```
insert into amigos2 (nome, estado) values ('João Brito', 'CE');
insert into amigos2 (nome, estado) values ('Roberto', 'MA');
insert into amigos2 (nome, estado) values ('Antônio', 'CE');
insert into amigos2 (nome, estado) values ('Francisco Queiroz', 'PB');
insert into amigos2 (nome, estado) values ('Bernardo dos Santos', 'MA');
insert into amigos2 (nome, estado) values ('Francisca Pinto', 'SP');
insert into amigos2 (nome, estado) values ('Natanael', 'SP');
```

```
select nome,
estado,
case
when estado = 'PB' then 'Fechado'
when estado = 'CE' or estado = 'SP' then 'Funcionando'
when estado = 'MA' then 'Funcionando a todo vapor'
else 'Menor'
end as status
from amigos2 order by nome;
```

```
create table notas(
nota numeric(4,2)
);
```

```
insert into notas(nota) values (4),
(6),
(3),
(10),
(6.5),
(7.3),
(7),
(8.8),
(9);
```

-- Mostrar cada nota junto com a menor nota, a maior nota, e a média de todas as notas.

```
SELECT nota,
(SELECT MIN(nota) FROM notas) AS menor,
(SELECT MAX(nota) FROM notas) AS maior,
(SELECT ROUND(AVG(nota),2) FROM notas) AS media
FROM notas;
```

Observar que a estrutura CASE... WHEN retorna apenas um único valor para onde foi chamada e também pode ser utilizada com UPDATE, como a seguir:

```
update produtos
set preco =
case
when preco < 10 then preco * 0.9
```

```
        when preco >= 10 and preco < 13 then preco * 0.8
        else preco 0.7
    end;
```

Funciona assim:

```
update produtos
    set preco = retorno do case;
```

Exemplo de uso do Case... When

Criemos uma tabela contendo apenas os estados vinda da tabela de ceps e adicionamos um campo para a descrição, mas queremos numa única consulta adicionar a descrição de todos os estados, tipo CE e Ceará:

- 1) create table estados as select distinct(uf) from cep_full_index;
- 2) alter table estados add column descricao varchar(30);
- 3) select * from estados

```
update estados set descricao=
CASE
    WHEN uf='AC' THEN 'Acre'
    WHEN uf='AL' THEN 'Alagoas'
    WHEN uf='AM' THEN 'Amazonas'
    WHEN uf='AP' THEN 'Amapá'
END;
```

Experimente executar este select e observe a ordem alfabética:

```
select * from estados;
```

Caso tenha problemas indique assim:

```
select * from estados order by uf, descricao;
```

Dicas Extras:

Como Adicionar uma chave estrangeira à tabela cep_full_index no campo uf relacionando com a tabela estados:

```
alter table cep_full_index add constraint uf_fk foreign key (uf) references estados(uf);
```

Como excluir a FK da tabela:

```
alter table cep_full_index drop constraint uf_fk;
```

Alguns exemplos de Constraints adicionadas no ALTER TABLE:

```
ALTER TABLE clientes ADD CHECK (length(cpf)=11);
```

```
ALTER TABLE clientes ADD CHECK (data_nasc >'1900-01-01' AND data_nasc <
```

```
CURRENT_DATE);  
ALTER TABLE clientes ADD CHECK (POSITION('@' IN email)>0);  
ALTER TABLE produtos ADD CHECK (quantidade > 0);  
ALTER TABLE produtos ALTER COLUMN data_compra SET DEFAULT CURRENT_DATE;
```

17) Funções para formatar tipo de dado

As funções de formatação do PostgreSQL fornecem um poderoso conjunto de ferramentas para converter vários tipos de dado (date/time, integer, floating point, numeric) em cadeias de caracteres formatadas, e para converter cadeias de caracteres formatadas em tipos de dado específicos. A [Tabela 9-22](#) mostra estas funções, que seguem uma convenção de chamada comum: o primeiro argumento é o valor a ser formatado, e o segundo argumento é o modelo que define o formato da entrada ou da saída.

Tabela 9-22. Funções de formatação

Função	Tipo retornado	Descrição	Exemplo
<code>to_char(timestamp, text)</code>	text	converte carimbo do tempo (time stamp) em cadeia de caracteres	<code>to_char(current_timestamp, 'HH12:MI:SS')</code>
<code>to_char(interval, text)</code>	text	converte intervalo em cadeia de caracteres	<code>to_char(interval '15h 2m 12s', 'HH24:MI:SS')</code>
<code>to_char(int, text)</code>	text	converte inteiro em cadeia de caracteres	<code>to_char(125, '999')</code>
<code>to_char(double precision, text)</code>	text	converte real e precisão dupla em cadeia de caracteres	<code>to_char(125.8::real, '999D9')</code>
<code>to_char(numeric, text)</code>	text	converte numérico em cadeia de caracteres	<code>to_char(-125.8, '999D99S')</code>
<code>to_date(text, text)</code>	date	converte cadeia de caracteres em data	<code>to_date('05 Dec 2000', 'DD Mon YYYY')</code>
<code>to_timestamp(text, text)</code>	timestamp with time zone	converte cadeia de caracteres em carimbo do tempo	<code>to_timestamp('05 Dec 2000', 'DD Mon YYYY')</code>
<code>to_number(text, text)</code>	numeric	converte cadeia de caracteres em numérico	<code>to_number('12,454.8-', '99G999D9S')</code>

Advertência: `to_char(interval, text)` está obsoleta, não devendo ser utilizada nos novos aplicativos. Será removida na próxima versão.

Em uma cadeia de caracteres modelo de saída (para `to_char`), existem certos padrões que são reconhecidos e substituídos pelos dados devidamente formatados a partir do valor a ser formatado. Qualquer texto que não seja um modelo padrão é simplesmente copiado sem alteração. Da mesma forma, em uma cadeia de caracteres modelo de entrada (para qualquer coisa menos `to_char`), os modelos padrão identificam as partes da cadeia de caracteres da entrada de dados a serem procuradas, e os valores a serem encontrados nestas partes.

A [Tabela 9-23](#) mostra os modelos padrão disponíveis para formatar valores de data e de hora.

Tabela 9-23. Modelos padrão para formatação de data e hora

Modelo	Descrição
HH	hora do dia (01-12)
HH12	hora do dia (01-12)

Modelo	Descrição
HH24	hora do dia (00-23)
MI	minuto (00-59)
SS	segundo (00-59)
MS	milissegundo (000-999)
US	microsegundo (000000-999999)
SSSS	segundos após a meia-noite (0-86399)
AM ou A.M. ou PM ou P.M.	indicador de meridiano (maiúsculas)
am ou a.m. ou pm ou p.m.	indicador de meridiano (minúsculas)
Y,YYY	ano (4 e mais dígitos) com vírgula
YYYY	ano (4 e mais dígitos)
YYY	últimos 3 dígitos do ano
YY	últimos 2 dígitos do ano
Y	último dígito do ano
IYYY	ano ISO (4 ou mais dígitos)
IYY	últimos 3 dígitos do ano ISO
IY	últimos 2 dígitos do ano ISO
I	último dígito do ano ISO
BC ou B.C. ou AD ou A.D.	indicador de era (maiúscula)
bc ou b.c. ou ad ou a.d.	indicador de era (minúscula)
MONTH	nome completo do mês em maiúsculas (9 caracteres completado com espaços)
Month	nome completo do mês em maiúsculas e minúsculas (9 caracteres completado com espaços)
month	nome completo do mês em minúsculas (9 caracteres completado com espaços)
MON	nome abreviado do mês em maiúsculas (3 caracteres)
Mon	nome abreviado do mês em maiúsculas e minúsculas (3 caracteres)
mon	nome abreviado do mês em minúsculas (3 caracteres)
MM	número do mês (01-12)
DAY	nome completo do dia em maiúsculas (9 caracteres completado com espaços)
Day	nome completo do dia em maiúsculas e minúsculas (9 caracteres completado com espaços)

Modelo	Descrição
day	nome completo do dia em minúsculas (9 caracteres completado com espaços)
DY	nome abreviado do dia em maiúsculas (3 caracteres)
Dy	nome abreviado do dia em maiúsculas e minúsculas (3 caracteres)
dy	nome abreviado do dia em minúsculas (3 caracteres)
DDD	dia do ano (001-366)
DD	dia do mês (01-31)
D	dia da semana (1-7; Domingo é 1)
W	semana do mês (1-5) onde a primeira semana começa no primeiro dia do mês
WW	número da semana do ano (1-53) onde a primeira semana começa no primeiro dia do ano
IW	número da semana do ano ISO (A primeira quinta-feira do novo ano está na semana 1)
CC	século (2 dígitos)
J	Dia Juliano (dias desde 1 de janeiro de 4712 AC)
Q	trimestre
RM	mês em algarismos romanos (I-XII; I=Janeiro) - maiúsculas
rm	mês em algarismos romanos (I-XII; I=Janeiro) - minúsculas
TZ	nome da zona horária - maiúsculas
tz	nome da zona horária - minúsculas

Certos modificadores podem ser aplicados aos modelos padrão para alterar seu comportamento. Por exemplo, FMMonth é o modelo "Month" com o modificador "FM". A [Tabela 9-24](#) mostra os modificadores de modelo para formatação de data e hora.

Tabela 9-24. Modificadores de modelo padrão para formatação de data e hora

Modificador	Descrição	Exemplo
prefixo FM	modo de preenchimento (suprime completar com brancos e zeros)	FMMonth
sufixo TH	sufixo de número ordinal maiúsculo	DDTH
sufixo th	sufixo de número ordinal minúsculo	DDth
prefixo FX	opção global de formato fixo (veja nota de utilização)	FX Month DD Day
sufixo SP	modo de falar (spell mode) (ainda não implementado)	DDSP

Notas sobre a utilização da formatação de data e hora:

- O FM suprime zeros à esquerda e espaços à direita, que de outra forma seriam adicionados para fazer a saída do modelo ter comprimento fixo.
- As funções to_timestamp e to_date saltam espaços em branco múltiplos na cadeia de

caracteres de entrada quando a opção FX não é utilizada. O FX deve ser especificado como o primeiro item do modelo; por exemplo, `to_timestamp('2000 JUN','YYYY MON')` está correto, mas `to_timestamp('2000 JUN','FXYYYY MON')` retorna erro, porque `to_timestamp` espera um único espaço apenas.

- É permitida a presença de texto comum nos modelos para `to_char`, sendo mostrados literalmente na saída. Uma parte da cadeia de caracteres pode ser colocada entre aspas, para obrigar sua interpretação como um texto literal mesmo contendo palavras chave do modelo. Por exemplo, em `"Hello Year 'YYYY'"`, o YYYY será substituído pelo ano do fornecido, mas o único Y em Year não será substituído.
- Se for desejada a presença de aspas na saída, as mesmas devem ser precedidas por contrabarra. Por exemplo `"\"YYYY Month\""`. (Duas contrabarras são necessárias, porque a contrabarra possui significado especial em uma constante cadeia de caracteres).
- A conversão YYYY de cadeia de caracteres para timestamp ou para date tem restrição quando são utilizados anos com mais de 4 dígitos. Deve ser utilizado um modelo, ou algum caractere que não seja um dígito, após YYYY, senão o ano será sempre interpretado como tendo 4 dígitos. Por exemplo, (com o ano 20000): `to_date('200001121','YYYYMMDD')` é interpretado como um ano de 4 dígitos; em vez disso, deve ser utilizado um separador que não seja um dígito após o ano, como `to_date('20000-1121','YYYY-MMDD')` ou `to_date('20000Nov21','YYYYMonDD')`.
- Os valores de milissegundos MS e microssegundos US na conversão de uma cadeia de caracteres para um carimbo do tempo (*timestamp*), são interpretados como a sendo parte dos segundos após o ponto decimal. Por exemplo, `to_timestamp('12:3','SS:MS')` não são 3 milissegundos, mas 300, porque a conversão interpreta como sendo 12 + 0.3 segundos. Isto significa que, para o formato SS:MS, os valores de entrada 12:3, 12:30 e 12:300 especificam o mesmo número de milissegundos. Para especificar três milissegundos deve ser utilizado 12:003, que na conversão é interpretado como 12 + 0.003 = 12.003 segundos.
A seguir está mostrado um exemplo mais complexo:
`to_timestamp('15:12:02.020.001230','HH:MI:SS.MS.US')` é interpretado como 15 horas, 12 minutos e 2 segundos + 20 milissegundos + 1230 microssegundos = 2.021230 segundos.
- A numeração do dia da semana de `to_char` (veja o modelo padrão de formatação 'D') é diferente do dia da semana da função `extract`.

A [Tabela 9-25](#) mostra os modelos padrão disponíveis para formatar valores numéricos.

Tabela 9-25. Modelos padrão para formatação de números

Modelo	Descrição
9	valor com o número especificado de dígitos
0	valor com zeros à esquerda
. (ponto)	ponto decimal
, (vírgula)	separador de grupo (milhares)
PR	valor negativo entre < e >
S	sinal preso ao número (utiliza o idioma)
L	símbolo da moeda (utiliza o idioma)
D	ponto decimal (utiliza o idioma)
G	separador de grupo (utiliza o idioma)

Modelo	Descrição
MI	sinal de menos na posição especificada (se número < 0)
PL	sinal de mais na posição especificada (se número > 0)
SG	sinal de mais/menos na posição especificada
RN [a]	algarismos romanos (entrada entre 1 e 3999)
TH ou th	sufixo de número ordinal
V	desloca o número especificado de dígitos (veja as notas sobre utilização)
EEEE	notação científica (ainda não implementada)
Notas:	
a. RN — roman numerals.	

Notas sobre a utilização da formatação numérica:

- O sinal formatado utilizando SG, PL ou MI não está ancorado ao número; por exemplo, `to_char(-12, 'S9999')` produz ' -12', mas `to_char(-12, 'MI9999')` produz '- 12'. A implementação do Oracle não permite utilizar o MI antes do 9, requerendo que o 9 preceda o MI.
- O 9 resulta em um valor com o mesmo número de dígitos que o número de 9s. Se não houver um dígito para colocar, é colocado espaço.
- O TH não converte valores menores que zero e não converte números fracionários.
- O PL, o SG e o TH são extensões do PostgreSQL.
- O V multiplica efetivamente os valores da entrada por 10^n , onde n é o número de dígitos após o V. A função `to_char` não permite o uso de V junto com o ponto decimal (Por exemplo, `99.9V99` não é permitido).

A [Tabela 9-26](#) mostra alguns exemplos de uso da função `to_char`.

Tabela 9-26. Exemplos de utilização da função `to_char`

Expressão	PostgreSQL 8.0.0 [a]	Oracle 10g (N. do T.) [b]
<code>to_char(current_timestamp, 'Day, DD HH12:MI:SS')</code>	'Friday , 04 02:22:24'	'Friday , 04 02:22:24'
<code>to_char(current_timestamp, 'FMDay, FMDD HH12:MI:SS')</code>	'Friday, 4 02:22:24'	'Friday, 04 02:22:24'
<code>to_char(-0.1, '99.99')</code>	' -.10'	' -.10'
<code>to_char(-0.1, 'FM9.99')</code>	'-.1'	'-.1'
<code>to_char(0.1, '0.9')</code>	' 0.1'	' 0.1'
<code>to_char(12, '9990999.9')</code>	' 0012.0'	' 0012.0'
<code>to_char(12, 'FM9990999.9')</code>	'0012.'	'0012.'
<code>to_char(485, '999')</code>	' 485'	' 485'
<code>to_char(-485, '999')</code>	'-485'	'-485'
<code>to_char(485, '9 9 9')</code>	' 4 8 5'	formato inválido
<code>to_char(1485, '9,999')</code>	' 1,485'	' 1,485'

Expressão	PostgreSQL 8.0.0 [a]	Oracle 10g (N. do T.) [b]
to_char(1485, '9G999')	' 1,485'	' 1,485'
to_char(148.5, '999.999')	' 148.500'	' 148.500'
to_char(148.5, 'FM999.999')	'148.5'	'148.5'
to_char(148.5, 'FM999.990')	'148.500'	'148.500'
to_char(148.5, '999D999') -- com idioma	' 148,500'	' 148,500'
to_char(3148.5, '9G999D999')	' 3,148.500'	' 3,148.500'
to_char(-485, '999S')	'485-'	'485-'
to_char(-485, '999MI')	'485-'	'485-'
to_char(485, '999MI')	'485 '	'485 '
to_char(485, 'FM999MI')	'485'	'485'
to_char(485, 'PL999')	'+485'	formato inválido
to_char(485, 'SG999')	'+485'	formato inválido
to_char(-485, 'SG999')	'-485'	formato inválido
to_char(-485, '9SG99')	'4-85'	formato inválido
to_char(-485, '999PR')	'<485>'	'<485>'
to_char(485, 'L999') -- com idioma	'R\$ 485'	' R\$485'
to_char(485, 'RN')	' CDLXXXV'	' CDLXXXV'
to_char(485, 'FMRN')	'CDLXXXV'	'CDLXXXV'
to_char(5.2, 'FMRN')	'V'	'V'
to_char(482, '999th')	' 482nd'	formato inválido
to_char(485, '"Good number:"999')	'Good number: 485'	formato inválido
to_char(485.8, '"Pre:"999" Post:" .999')	'Pre: 485 Post: .800'	formato inválido
to_char(12, '99V999')	' 12000'	' 12000'
to_char(12.4, '99V999')	' 12400'	' 12400'
to_char(12.45, '99V9')	' 125'	' 125'
Notas:		
a. idioma — set lc_numeric to 'pt_BR'; set lc_monetary to 'pt_BR'; (N. do T.)		
b. idioma — ALTER SESSION SET NLS_TERRITORY="BRAZIL"; (N. do T.)		

Fonte: <http://pgdocptbr.sourceforge.net/pg80/functions-formatting.html>

Formatação - Juliano Ignácio

Bom,... época de carnaval é assim, todos usam máscaras! - então, vamos nos aproveitar da analogia. As saídas de dados limpos, ou seja, o valor diretamente armazenado, às vezes, não é interessante. Não só pela questão da aparência em si, mas da visualização correta da informação. O PostgreSQL

contém diversas funções para formatação dos dados.

Por exemplo, para visualizar a formatação de um valor monetário, inserido no valor de venda em nossa tabela de exercício, definido como sendo do tipo numeric(15,2), primeiro vamos inserir dois itens:

```
INSERT INTO venda (cliente_id, data, valor, produto) VALUES (1, CURRENT_DATE, 1325.50, 'Mini System');
```

```
INSERT INTO venda (cliente_id, data, valor, produto) VALUES (1, CURRENT_DATE, 12422.03, 'Kart');
```

Agora sim, vamos à formatação de saída:

<code>select produto, valor from venda where cliente_id = 1;</code>	<code>select produto, to_char(valor,'999G999D99') as valor_fmt from venda where cliente_id = 1;</code>
<pre>produto valor -----+----- Kart 12422.03 Mini System 1325.50 Fichário 32.00 Mala Viagem 110.00 Saca-rolha 10.00 Relógio 16.00 (6 rows)</pre>	<pre>produto valor_fmt -----+----- Kart 12,422.03 Mini System 1,325.50 Fichário 32.00 Mala Viagem 110.00 Saca-rolha 10.00 Relógio 16.00 (6 rows)</pre>

Como pode-se perceber, para valores maiores, a visualização oferece o entendimento mais fácil e mais rápido.

Vamos ver outros casos:

<code>select produto, to_char(valor,'999D99') as valor_fmt from venda where cliente_id = 1;</code>	<code>select produto, to_char(valor,'R\$"999G990D00') as valor_fmt from venda where cliente_id = 1;</code>
<pre>produto valor -----+----- Kart ###.## Mini System ###.## Fichário 32.00 Mala Viagem 110.00 Saca-rolha 10.00 Relógio 16.00 (6 rows)</pre>	<pre>produto valor_fmt -----+----- Kart R\$ 12,422.03 Mini System R\$ 1,325.50 Fichário R\$ 32.00 Mala Viagem R\$ 110.00 Saca-rolha R\$ 10.00 Relógio R\$ 16.00 (6 rows)</pre>

Aqui temos no primeiro caso, um cuidado que se deve ter, a máscara sempre deve possuir uma tamanho que trate o maior valor provável que será tratado, pois, caso contrário, somente os caracteres # (sustenido ou jogo-da-velha) serão mostrados. No segundo exemplo, foi inserido um texto ("R\$") para mostrar a flexibilidade da formatação.

A formatação não é exclusiva para tratamento e conversão de números, pode-se trabalhar com data e hora também, o que ajuda muito em diversas situações, por exemplo, uma das maneiras de se extrair dos dados de venda, quais foram os produtos vendidos em fevereiro de 2003, pode ser executado como:

<code>select produto from venda where to_char(data,'MMYYYY') = '022003';</code>
<pre>produto ----- Relógio Mala Viagem Saca-rolha Fichário Mini System Kart (6 rows)</pre>

Fonte: <http://imasters.uol.com.br/artigo/974/postgresql/formatacao/>

Geométricos

18) Trabalhando com pontos e linhas

De Juliano Ignácio

Não, não é nenhum artigo sobre corte e costura, ou ainda, sobre desenhos (como vemos no Paint do Windows). Trata-se de tipos de dados que constam no PostgreSQL específicos para operações geométricas.

O PostgreSQL tem condições de ser utilizado para construir poderosas soluções que exijam tratamento científico, onde, tais recursos são dificilmente encontrados em bancos de dados comerciais (Oracle e DB2 têm esses recursos). Este artigo dará uma pequena idéia de como tratar dados geográficos (Geo-Data) com o PostgreSQL, um assunto cada vez mais importante: o geoprocessamento (GIS).

Dados geométricos podem ser armazenados com o auxílio de alguns tipos de dados do PostgreSQL: point, line, box, path, polygon e circle.

point

O ponto é o ponto fundamental (desculpem, não resisti ao trocadilho) para o tratamento de objetos geométricos. Ele pode ser manuseado fácil e eficientemente pelo usuário. Um ponto (neste caso) é definido por dois valores: o primeiro é o valor da coordenada do eixo X, o segundo é o valor da coordenada do eixo Y. Ambos os valores são armazenados internamente no banco de dados PostgreSQL como valores do tipo ponto-flutuante (8 bytes), portanto, um ponto requer 16 bytes de armazenamento.

<pre>CREATE TABLE tabpontos(posicao point); INSERT INTO tabpontos(posicao) VALUES ('1,2'); INSERT INTO tabpontos(posicao) VALUES ('1,3');</pre>	<pre>SELECT * FROM tabpontos; posicao ----- (1,2) (1,3) (2 rows)</pre>
--	---

Um detalhe muito importante é que ao inserirmos os pontos, estes devem estar entre apóstrofes, caso contrário, um erro sobre tipos diferentes de dados será informado.

line

Uma linha é definida através de dois pontos (ou seja, quatro valores ou dois pares de coordenadas). O primeiro ponto determina o início da linha e, o segundo, determina seu término. Portanto, para armazenar uma linha serão utilizados 32 bytes. Existem dois tipos definidos para linha: line e lseg. O tipo line ainda não está totalmente implementado, usaremos então o lseg.

<pre>CREATE TABLE tablinhas(segmento lseg); INSERT INTO tablinhas(segmento) VALUES ('1,4,3,5'); INSERT INTO tablinhas(segmento) VALUES ('((1,4),(3,5))'); INSERT INTO tablinhas(segmento) VALUES ('[(2,6),(4,7)]');</pre>	<pre>SELECT * FROM tablinhas; segmento ----- [(1,4),(3,5)] [(1,4),(3,5)] [(2,6),(4,7)] (3 rows)</pre>
--	--

A utilização de colchetes e parênteses facilita a leitura ao inserir os pontos da linha.

box

O tipo box é utilizado para armazenar um retângulo (ou quadrado), definido por dois pontos (exatamente como a linha), porém, esta linha, define a diagonal deste retângulo. Portanto, o espaço destinado ao armazenamento de um retângulo é igual ao de uma linha, 32 bytes.

CREATE TABLE tabbarearec(area box);	SELECT * FROM tabbarearec;
INSERT INTO tabbarearec(area) VALUES ('7,5,2,3');	area
INSERT INTO tabbarearec(area) VALUES ('((1,1),(8,0))');	(7,5),(2,3)
	(8,1),(1,0)
	(2 rows)

Aqui, somente a utilização de parênteses é permitida, os colchetes não.

path

O **path** é um caminho, ou seja, uma seqüência de pontos que pode ser aberta ou fechada. Fechada significa que o último ponto do caminho retorna ao início da seqüência. O tamanho (ou espaço de armazenamento) de um **path** é dinâmico, ou seja, são utilizados 4 bytes de início e 32 bytes para cada nó que for armazenado. O PostgreSQL também fornece algumas funções especiais para verificar se o caminho é aberto ou fechado: popen() e pclose() forçam que o caminho deva ser aberto ou fechado respectivamente; isopen() eisclosed() fazem a verificação.

CREATE TABLE tabtrilha(trajeto path);	SELECT * FROM tabtrilha;
INSERT INTO tabtrilha(trajeto) VALUES ('(1,2), (5,3)');	trajeto
INSERT INTO tabtrilha(trajeto) VALUES ('(1,3), (-3,1), (7,0)');	((1,2),(5,3))
	((1,3),(-3,1),(7,0))
	(2 rows)

polygon

O polígono é um caminho fechado por definição.

CREATE TABLE tabarea(area polygon);	SELECT * FROM tabarea;
INSERT INTO tabarea(area) VALUES ('(1,3), (4,16), (0,23)');	area
INSERT INTO tabarea(area) VALUES ('(2,7), (5,-15), (1,20)');	((1,3),(4,16),(0,23))
	((2,7),(5,-15),(1,20))
	(2 rows)

circle

Um círculo consiste de um ponto central e seu raio, e precisa de 24 bytes para ser armazenado.

CREATE TABLE tabareacirc(area circle);	SELECT * FROM tabareacirc;
INSERT INTO tabareacirc(area) VALUES ('(10,12,10)');	area
INSERT INTO tabareacirc(area) VALUES ('(12,19,5)');	<(10,12),10>
	<(12,19),5>
	<(2,9),0>
	(3 rows)

O tipo **circle** não aceita o uso de parênteses para uma melhor definição visual e, além disso, o valor do raio **SEMPRE** deve ser maior ou igual a zero.

OPERADORES PARA DADOS GEOMÉTRICOS

Primeira regra: lembre-se de usar o operador ~= no lugar de = sempre, caso contrário receberá uma mensagem de erro. Por exemplo:

```
SELECT * FROM tabpontos WHERE posicao ~= '(1,2)';
```

Segundo: existem operadores para cálculo entre dados geométricos. Por exemplo:

Distância entre 2 pontos	SELECT '(1,1)::point <-> '(3,9)::point;
Adição de 2 pontos	SELECT '(2,1)::point + '(4,8)::point;
Subtração de 2 pontos	SELECT '(3,1)::point - '(5,7)::point;
Multiplicação de 2 pontos	SELECT '(4,1)::point * '(6,6)::point;
Divisão de 2 pontos	SELECT '(5,1)::point / '(7,5)::point;
Divisão de 2 pontos	SELECT '(5,1)::point / '(7,5)::point;
Intersecção de 2 segmentos	SELECT '((0,0),(10,1))::lseg # '((-4,5),(10,-3))::lseg;
O ponto mais próximo	SELECT '(10,1)::point ## '((-4,5),(10,-3))::lseg;

Terceiro: existem operadores para análise dos pontos. Por exemplo:

Pontos perpendiculares na horizontal	<code>SELECT '(6,1)::point' ?- '(8,4)::point;</code>
Pontos perpendiculares na vertical	<code>SELECT '(6,1)::point' ? '(8,4)::point;</code>
O segundo ponto está à esquerda do primeiro	<code>SELECT '(-8,0)::point' << '(0,0)::point;</code>
O segundo ponto está abaixo do primeiro	<code>SELECT '(0,-9)::point' <^ '(0,0)::point;</code>
O segundo ponto está à direita do primeiro	<code>SELECT '(6,0)::point' >> '(0,0)::point;</code>
O segundo ponto está acima do primeiro	<code>SELECT '(0,5)::point' >^ '(0,0)::point;</code>

Quarto: existem diversos outros operadores para trabalhar com dados geométricos (&&, &<, &>, ?, #, @-@, ?||, @, @@), procure maiores detalhes na documentação do PostgreSQL, ou execute `/do` em seu `psql` para ver todos os operadores.

Link do original:

http://imasters.uol.com.br/artigo/1004/postgresql/trabalhando_com_pontos_e_linhas/imprimir/

Tipos geométricos

Os tipos de dado geométricos representam objetos espaciais bidimensionais. A [Tabela 8-16](#) mostra os tipos geométricos disponíveis no PostgreSQL. O tipo mais fundamental, o ponto, forma a base para todos os outros tipos.

Tabela 8-16. Tipos geométricos

Nome	Tamanho de Armazenamento	Descrição	Representação
point	16 bytes	Ponto no plano	(x,y)
line	32 bytes	Linha infinita (não totalmente implementado)	((x1,y1),(x2,y2))
lseg	32 bytes	Segmento de linha finito	((x1,y1),(x2,y2))
box	32 bytes	Caixa retangular	((x1,y1),(x2,y2))
path	16+16n bytes	Caminho fechado (semelhante ao polígono)	((x1,y1),...)
path	16+16n bytes	Caminho aberto	[(x1,y1),...]
polygon	40+16n bytes	Polígono (semelhante ao caminho fechado)	((x1,y1),...)
circle	24 bytes	Círculo	<(x,y),r> (centro e raio)

Está disponível um amplo conjunto de funções e operadores para realizar várias operações geométricas, como escala, translação, rotação e determinar interseções, conforme explicadas na [Seção 9.10](#).

Pontos

Os pontos são os blocos de construção bidimensionais fundamentais para os tipos geométricos. Os

valores do tipo point são especificados utilizando a seguinte sintaxe:

```
( x , y )
  x , y
```

onde x e y são as respectivas coordenadas na forma de números de ponto flutuante.

Segmentos de linha

Os segmentos de linha (lseg) são representados por pares de pontos. Os valores do tipo lseg são especificado utilizando a seguinte sintaxe:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1 , x2 , y2
```

onde (x1,y1) e (x2,y2) são os pontos das extremidades do segmento de linha.

Caixas

As caixas são representadas por pares de pontos de vértices opostos da caixa. Os valores do tipo box são especificados utilizando a seguinte sintaxe:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1 , x2 , y2
```

onde (x1,y1) e (x2,y2) são quaisquer vértices opostos da caixa.

As caixas são mostradas utilizando a primeira sintaxe. Os vértices são reordenados na entrada para armazenar o vértice direito superior e, depois, o vértice esquerdo inferior. Podem ser especificados outros vértices da caixa, mas os vértices esquerdo inferior e direito superior são determinados a partir da entrada e armazenados.

Caminhos

Os caminhos são representados por listas de pontos conectados. Os caminhos podem ser *abertos*, onde o primeiro e o último ponto da lista não são considerados conectados, e *fechados*, onde o primeiro e o último ponto são considerados conectados.

Os valores do tipo path são especificados utilizando a seguinte sintaxe:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1 , ... , xn , yn )
    x1 , y1 , ... , xn , yn
```

onde os pontos são os pontos das extremidades dos segmentos de linha que compõem o caminho. Os colchetes ([]) indicam um caminho aberto, enquanto os parênteses (()) indicam um caminho fechado.

Os caminhos são mostrados utilizando a primeira sintaxe.

Polígonos

Os polígonos são representados por uma lista de pontos (os vértices do polígono). Provavelmente os polígonos deveriam ser considerados equivalentes aos caminhos fechados, mas são armazenados de forma diferente e possuem um conjunto próprio de rotinas de suporte.

Os valores do tipo polygon são especificados utilizando a seguinte sintaxe:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
( x1 , y1 ) , ... , ( xn , yn )
( x1 , y1 , ... , xn , yn )
x1 , y1 , ... , xn , yn
```

onde os pontos são os pontos das extremidades dos segmentos de linha compondo a fronteira do polígono.

Os polígonos são mostrados utilizando a primeira sintaxe.

Círculos

Os círculos são representados por um ponto central e um raio. Os valores do tipo circle são especificado utilizando a seguinte sintaxe:

```
< ( x , y ) , r >
( ( x , y ) , r )
( x , y ) , r
x , y , r
```

onde (x,y) é o centro e r é o raio do círculo.

Os círculos são mostrados utilizando a primeira sintaxe.

Fonte: <http://pgdocptbr.sourceforge.net/pg80/datatype-geometric.html>

Funções e operadores geométricos

Os tipos geométricos point, box, lseg, line, path, polygon e circle possuem um amplo conjunto de funções e operadores nativos para apoiá-los, mostrados na [Tabela 9-30](#), na [Tabela 9-31](#) e na [Tabela 9-32](#).

Tabela 9-30. Operadores geométricos

Operador	Descrição	Exemplo
+	Translação	box '((0,0),(1,1))' + point '(2.0,0)'
-	Translação	box '((0,0),(1,1))' - point '(2.0,0)'
*	Escala/rotação	box '((0,0),(1,1))' * point '(2.0,0)'
/	Escala/rotação	box '((0,0),(2,2))' / point '(2.0,0)'
#	Ponto ou caixa de interseção	'((1,-1),(-1,1))' # '((1,1),(-1,-1))'

Operador	Descrição	Exemplo
#	Número de pontos do caminho ou do polígono	# '((1,0),(0,1),(-1,0))'
@-@	Comprimento ou circunferência	@-@ path '((0,0),(1,0))'
@@	Centro	@@ circle '((0,0),10)'
##	Ponto mais próximo do primeiro operando no segundo operando	point '(0,0)' ## lseg '((2,0),(0,2))'
<->	Distância entre	circle '((0,0),1)' <-> circle '((5,0),1)'
&&	Se sobrepõem?	box '((0,0),(1,1))' && box '((0,0),(2,2))'
&<	Não se estende à direita de?	box '((0,0),(1,1))' &< box '((0,0),(2,2))'
&>	Não se estende à esquerda de?	box '((0,0),(3,3))' &> box '((0,0),(2,2))'
<<	Está à esquerda?	circle '((0,0),1)' << circle '((5,0),1)'
>>	Está à direita?	circle '((5,0),1)' >> circle '((0,0),1)'
<^	Está abaixo?	circle '((0,0),1)' <^ circle '((0,5),1)'
>^	Está acima?	circle '((0,5),1)' >^ circle '((0,0),1)'
?#	Se intersectam?	lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))'
?-	É horizontal?	?- lseg '((-1,0),(1,0))'
?-	São alinhados horizontalmente?	point '(1,0)' ?- point '(0,0)'
?	É vertical?	? lseg '((-1,0),(1,0))'
?	São alinhados verticalmente	point '(0,1)' ? point '(0,0)'
?-	São perpendiculares?	lseg '((0,0),(0,1))' ?- lseg '((0,0),(1,0))'
?	São paralelos?	lseg '((-1,0),(1,0))' ? lseg '((-1,2),(1,2))'
~	Contém?	circle '((0,0),2)' ~ point '(1,1)'
@	Está contido ou sobre?	point '(1,1)' @ circle '((0,0),2)'
~=	O mesmo que?	polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'

Tabela 9-31. Funções geométricas

Função	Tipo retornado	Descrição	Exemplo
area(object)	double precision	área	area(box '((0,0),(1,1))')
box_intersect(box	box	caixa de interseção	box_intersect(box '((0,0),

Função	Tipo retornado	Descrição	Exemplo
, box)			(1,1))', box '((0.5,0.5),(2,2))')
center(object)	point	centro	center(box '((0,0),(1,2))')
diameter(circle)	double precision	diâmetro do círculo	diameter(circle '((0,0),2.0)')
height(box)	double precision	tamanho vertical da caixa	height(box '((0,0),(1,1))')
isclosed(path)	boolean	é um caminho fechado?	isclosed(path '((0,0),(1,1),(2,0))')
isopen(path)	boolean	é um caminho aberto?	isopen(path '[(0,0),(1,1),(2,0)]')
length(object)	double precision	comprimento	length(path '((-1,0),(1,0))')
npoints(path)	integer	número de pontos	npoints(path '[(0,0),(1,1),(2,0)]')
npoints(polygon)	integer	número de pontos	npoints(polygon '((1,1),(0,0))')
pclose(path)	path	converte o caminho em caminho fechado	pclose(path '[(0,0),(1,1),(2,0)]')
popen(path)	path	converte o caminho em caminho aberto	popen(path '((0,0),(1,1),(2,0))')
radius(circle)	double precision	raio do círculo	radius(circle '((0,0),2.0)')
width(box)	double precision	tamanho horizontal da caixa	width(box '((0,0),(1,1))')

Tabela 9-32. Funções de conversão de tipo geométrico

Função	Tipo retornado	Descrição	Exemplo
box(circle)	box	círculo em caixa	box(circle '((0,0),2.0)')
box(point, point)	box	pontos em caixa	box(point '(0,0)', point '(1,1)')
box(polygon)	box	polígono em caixa	box(polygon '((0,0),(1,1),(2,0))')
circle(box)	circle	caixa em círculo	circle(box '((0,0),(1,1))')
circle(point, double precision)	circle	centro e raio em círculo	circle(point '(0,0)', 2.0)
lseg(box)	lseg	diagonal de caixa em segmento de linha	lseg(box '((-1,0),(1,0))')
lseg(point, point)	lseg	ponto em segmento de linha	lseg(point '(-1,0)', point '(1,0)')
path(polygon)	point	polígono em caminho	path(polygon '((0,0),(1,1),(2,0))')

Função	Tipo retornado	Descrição	Exemplo
point(double precision, double precision)	point	constrói ponto	point(23.4, -44.5)
point(box)	point	centro da caixa	point(box '((-1,0),(1,0))')
point(circle)	point	centro do círculo	point(circle '((0,0),2.0)')
point(lseg)	point	centro do segmento de linha	point(lseg '((-1,0),(1,0))')
point(lseg, lseg)	point	intersecção	point(lseg '((-1,0),(1,0))', lseg '((-2,-2),(2,2))')
point(polygon)	point	centro do polígono	point(polygon '((0,0),(1,1),(2,0))')
polygon(box)	polygon	caixa em polígono de 4 pontos	polygon(box '((0,0),(1,1))')
polygon(circle)	polygon	círculo em polígono de 12 pontos	polygon(circle '((0,0),2.0)')
polygon(npts, circle)	polygon	círculo em polígono de npts-pontos	polygon(12, circle '((0,0),2.0)')
polygon(path)	polygon	caminho em polígono	polygon(path '((0,0),(1,1),(2,0))')

É possível acessar os dois números que compõem um point como se este fosse uma matriz com os índices 0 e 1. Por exemplo, se t.p for uma coluna do tipo point, então `SELECT p[0] FROM t` retorna a coordenada X, e `UPDATE t SET p[1] = ...` altera a coordenada Y. Do mesmo modo, um valor do tipo box ou lseg pode ser tratado como sendo uma matriz contendo dois valores do tipo point.

As funções area operam sobre os tipos box, circle e path. A função area somente opera sobre o tipo de dado path se os pontos em path não se intersectarem. Por exemplo, não opera sobre o path `'((0,0),(0,1),(2,1),(2,2),(1,2),(1,0),(0,0))':PATH`, entretanto opera sobre o path visualmente idêntico `'((0,0),(0,1),(1,1),(1,2),(2,2),(2,1),(1,1),(1,0),(0,0))':PATH`. Se o conceito de path que intersecta e que não intersecta estiver confuso, desenhe os dois caminhos acima lado a lado em uma folha de papel gráfico.

<http://pgdocptbr.sourceforge.net/pg80/functions-geometry.html>

Capítulo de E-book Online:

Chapter: Geometric Data Types

PostgreSQL supports six data types that represent two-dimensional geometric objects. The most basic geometric data type is the `POINT`?as you might expect, a `POINT` represents a point within a two-dimensional plane.

A `POINT` is composed of an x-coordinate and a y-coordinate?each coordinate is a `DOUBLE PRECISION` number.

The `LSEG` data type represents a two-dimensional line segment. When you create a `LSEG` value, you specify two points: the starting `POINT` and the ending `POINT`.

A `BOX` value is used to define a rectangle: the two points that define a box specify opposite corners.

A `PATH` is a collection of an arbitrary number of `POINTS` that are connected. A `PATH` can specify either a closed path or an open path. In a closed path, the beginning and ending points are considered to be connected, and in an open path, the first and last points are not connected. PostgreSQL provides two functions to force a `PATH` to be either open or closed: `POPEN()` and `PCLOSE()`. You can also specify whether a `PATH` is open or closed using special literal syntax (described later).

A `POLYGON` is similar to a closed `PATH`. The difference between the two types is in the supporting functions.

A center `POINT` and a (`DOUBLE PRECISION`) floating-point radius represent a `CIRCLE`.

Table 2.18 summarizes the geometric data types.

Syntax for Literal Values

When you enter a value for geometric data type, keep in mind that you are working with a list of two-dimensional points (except in the case of a `CIRCLE`, where you are working with a `POINT` and a radius).

A single `POINT` can be entered in either of the following two forms:

```
'( x, y )'
```

```
' x, y '
```

The `LSEG` and `BOX` types are constructed from a pair of `POINTS`. You can enter a pair of `POINTS` in any of the following formats:

```
'(( x1, y1 ), ( x2, y2 ))'
```

```
'( x1, y1 ), ( x2, y2 )'
```

```
'x1, y1, x2, y2'
```

The `PATH` and `POLYGON` types are constructed from a list of one or more `POINTS`. Any of the following forms is acceptable for a `PATH` or `POLYGON` literal:

```
'(( x1, y1 ), ..., ( xn, yn ))'
```

```
'( x1, y1 ), ..., ( xn, yn )'
```

```
'( x1, y1, ..., xn, yn )'
```

```
'x1, y1, ..., xn, yn'
```

You can also use the syntax `'[(x1, y1), ..., (xn, yn)]'` to enter a `PATH` literal:

A `PATH` entered in this form is considered to be an open `PATH`.

A `CIRCLE` is described by a central point and a floating point radius. You can enter a `CIRCLE` in any of the following forms:

```
'< ( x, y ), r >'
```

```
'(( x, y ), r )'
```

```
'( x, y ), r'
```

```
'x, y, r'
```

Notice that the surrounding single quotes are required around all geometric literals?in other words, geometric literals are entered as string literals. If you want to create a geometric value from individual components, you will have to use a geometric conversion function. For example, if you want to create a `POINT` value from the results of some computation, you would use:

```
POINT( 4, 3*height )
```

The `POINT(DOUBLE PRECISION x, DOUBLE PRECISION y)` function creates a `POINT` value from two `DOUBLE PRECISION` values. There are similar functions that you can use to create any geometric type starting from individual components. Table 2.19 lists the conversion functions for geometric types.

Table 2.19. Type Conversion Operators for the Geometric Data Types

Result Type	Meaning
<code>POINT</code>	<code>POINT(DOUBLE PRECISION x, DOUBLE PRECISION y)</code>
<code>LSEG</code>	<code>LSEG(POINT p1, POINT p2)</code>
<code>BOX</code>	<code>BOX(POINT p1, POINT p2)</code>
<code>PATH</code>	<code>PATH(POLYGON poly)</code>
<code>POLYGON</code>	<code>POLYGON(PATH path)</code>
	<code>POLYGON(BOX b)</code>
	yields a 12-point polygon
	<code>POLYGON(CIRCLE c)</code>

Result Type	Meaning
-------------	---------

	yields a 12-point polygon
--	---------------------------

	POLYGON(INTEGER n, CIRCLE c)
--	--------------------------------

	yields an n point polygon
--	---------------------------

CIRCLE	CIRCLE(BOX b)
--------	-----------------

	CIRCLE(POINT radius, DOUBLE PRECISION point)
--	--

Sizes and Valid Values

Table 2.20 lists the size of each geometric data type.

Table 2.20. Geographic Data Type Storage Requirements

Type	Size (in bytes)
POINT	16 (2 <input type="text"/> sizeof DOUBLE PRECISION)
LSEG	32 (2 <input type="text"/> sizeof POINT)
BOX	32 (2 <input type="text"/> sizeof POINT)
PATH	4+(32 <input type="text"/> number of points)[4]
POLYGON	4+(32 <input type="text"/> number of points) ^[4]
CIRCLE	24 (sizeof POINT + sizeof DOUBLE PRECISION)

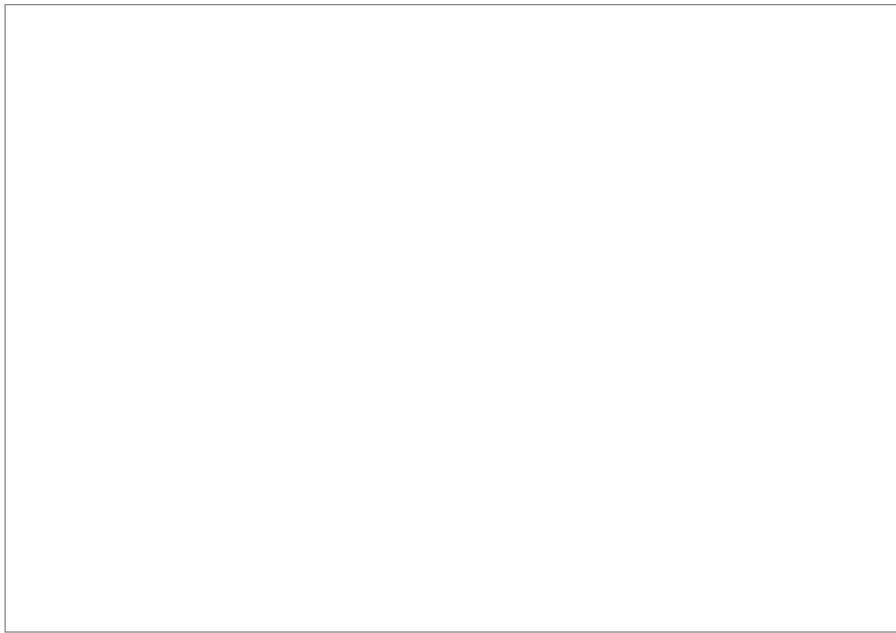
^[4] The size of a PATH or POLYGON is equal to 4 + (sizeof LSEG number of segments).

Supported Operators

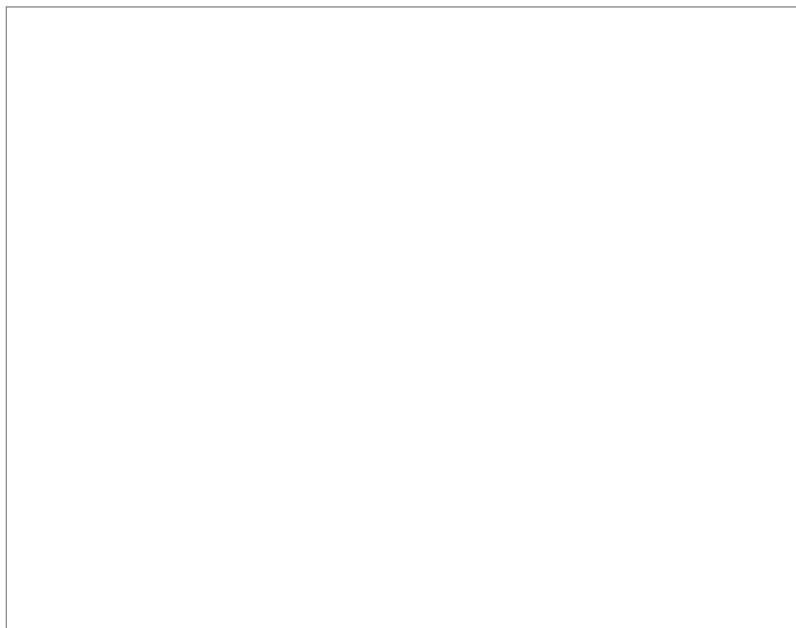
PostgreSQL features a large collection of operators that work with the geometric data types. I've divided the geometric operators into two broad categories (transformation and proximity) to make it a little easier to talk about them.

Using the transformation operators, you can translate, rotate, and scale geometric objects. The + and - operators translate a geometric object to a new location. Consider Figure 2.1, which shows a BOX defined as `BOX (POINT (3,5) , POINT (1,2))`.

Figure 2.1. `BOX(POINT(3,5), POINT(1,2))`.



If you use the + operator to add the `POINT (2,1)` to this BOX, you end up with the object shown in Figure 2.2.

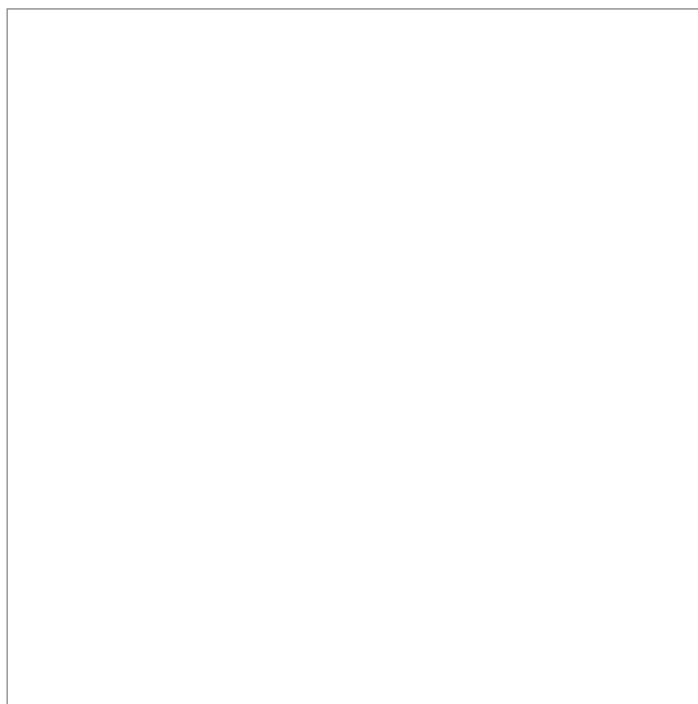
Figure 2.2. Geometric translation.

You can see that the x-coordinate of the `POINT` is added to each of the x-coordinates in the `BOX`, and the y-coordinate of the `POINT` is added to the y-coordinates in the `BOX`. The `-` operator works in a similar fashion: the x-coordinate of the `POINT` is subtracted from the x-coordinates of the `BOX`, and the y-coordinate of the `POINT` is subtracted from each y-coordinate in the `BOX`.

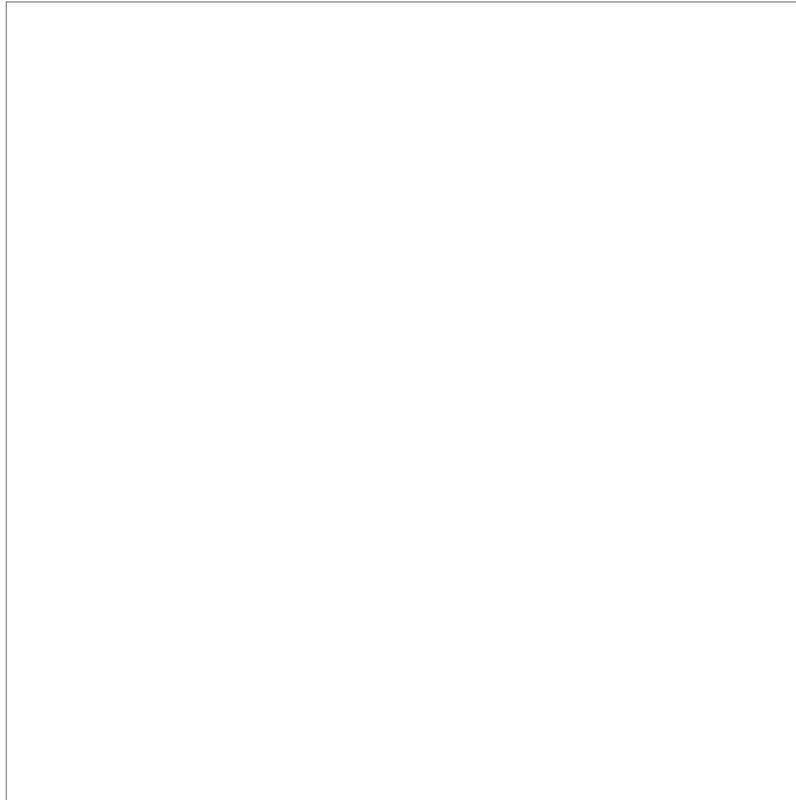
Using the `+` and `-` operators, you can move a `POINT`, `BOX`, `PATH`, or `CIRCLE` to a new location. In each case, the x-coordinate in the second operand (a `POINT`), is added or subtracted from each x-coordinate in the first operand, and the y-coordinate in the second operand is added or subtracted from each y-coordinate in the first operand.

The multiplication and division operators (`*` and `/`) are used to scale and rotate. The multiplication and division operators treat the operands as points in the complex plane. Let's look at some examples.

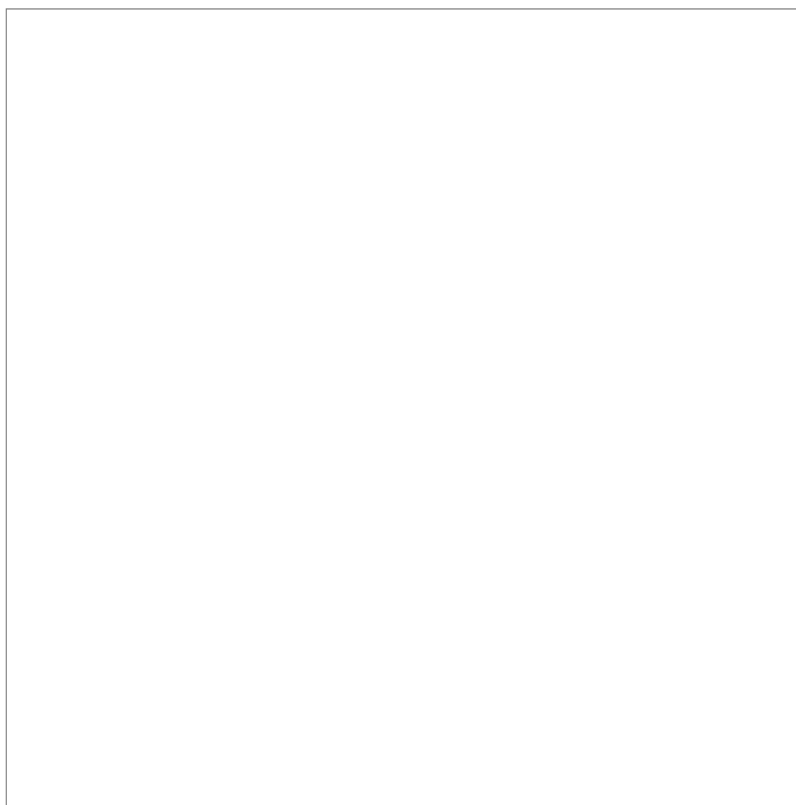
Figure 2.3 shows the result of multiplying `BOX (POINT (3, 2) , POINT (1, 1))` by `POINT (2, 0)`.

Figure 2.3. Point multiplication?scaling by a positive value.

You can see that each coordinate in the original box is multiplied by the x-coordinate of the point, resulting in `BOX (POINT (6, 4) , POINT (2, 2))`. If you had multiplied the box by `POINT (0.5, 0)`, you would have ended up with `BOX (POINT (1.5, 1) , POINT (0.5, 0.5))`. So the effect of multiplying an object by `POINT (x, 0)` is that each coordinate in the object moves away from the origin by a factor x. If x is negative, the coordinates move to the other side of the origin, as shown in Figure 2.4.

Figure 2.4. Point multiplication?scaling by a negative value.

You can see that the x-coordinate controls scaling. The y-coordinate controls rotation. When you multiply any given geometric object by `POINT (0, y)`, each point in the object is rotated around the origin. When `y` is equal to 1, each point is rotated counterclockwise by 90° about the origin. When `y` is equal to -1, each point is rotated 90° about the origin (or 270°). When you rotate a point without scaling, the length of the line segment drawn between the point and origin remains constant, as shown in Figure 2.5.

Figure 2.5. Point multiplication?rotation.

You can combine rotation and scaling into the same operation by specifying non-zero values for both the x- and y-coordinates. For more information on using complex numbers to represent geometric points, see <http://www.clarku.edu/~djoyce/complex>.

Table 2.21 shows the valid combinations for geometric types and geometric operators.

Table 2.21. Transformation Operators for the Geometric Types

Data Types	Valid Operators (θ)
POINT θ POINT	* + - /
BOX θ POINT	* + - /
PATH θ POINT	* + - /
CIRCLE θ POINT	* + - /

The proximity operators allow you to determine the spatial relationships between two geometric objects.

First, let's look at the three containment operators. The ~ operator evaluates to TRUE if the left operand contains the right operand. The @ operator evaluates to TRUE if the left operand is

contained within the right operand. The `~=` returns `TRUE` if the left operand is the same as the right operand?two geographic objects are considered identical if the points that define the objects are identical (two circles are considered identical if the radii and center points are the same).

The next two operators are used to determine the distance between two geometric objects.

The `##` operator returns the closest point between two objects. You can use the `##` operator with the following operand types shown in Table 2.22.

Table 2.22. Closest-Point Operators

Operator	Description
<code>LSEG_a ## BOX_b</code>	Returns the point in <code>BOX_b</code> that is closest to <code>LSEG_a</code>
<code>LSEG_a ## LSEG_b</code>	Returns the point in <code>LSEG_b</code> that is closest to <code>LSEG_a</code>
<code>POINT_a ## BOX_b</code>	Returns the point in <code>BOX_b</code> that is closest to <code>POINT_a</code>
<code>POINT_a ## LSEG_b</code>	Returns the point in <code>LSEG_b</code> that is closest to <code>POINT_a</code>

The distance (`<->`) operator returns (as a `DOUBLE PRECISION` number) the distance between two geometric objects. You can use the distance operator with the operand types in Table 2.23.

Table 2.23. Distance Operators

Operator	Description (or Formula)
<code>BOX_a <-> BOX_b</code>	<code>((@@ BOX_a) <-> (@@ BOX_b))</code>
<code>CIRCLE_a <-> CIRCLE_b</code>	<code>((@@ CIRCLE_a) <-> (@@ CIRCLE_b))</code>
	?
	<code>(radius_a + radius_b)</code>
<code>CIRCLE_a <-> POLYGON_b</code>	0 if any point in <code>POLYGON_b</code> is inside <code>CIRCLE_a</code> otherwise, distance between center of <code>CIRCLE_a</code> and closest point in <code>POLYGON_b</code>
<code>LSEG_a <-> BOX_b</code>	<code>(LSEG ## BOX) <-> (LSEG ## (LSEG ## BOX))</code>

Operator	Description (or Formula)
$LSEG_a <-> LSEG_b$	Distance between closest points (0 if $LSEG_a$ intersects $LSEG_b$)
$PATH_a <-> PATH_b$	Distance between closest points
$POINT_a <-> BOX_b$	$POINT_a <-> (POINT_a \ \#\# \ BOX_b)$
$POINT_a <-> CIRCLE_b$	$POINT_a <-> ((@@ \ CIRCLE_b) \ ? \ CIRCLE_b \ radius)$
$POINT_a <-> LSEG_b$	$POINT_a <-> (POINT_a \ \#\# \ LSEG_b)$
$POINT_a <-> PATH_b$	Distance between $POINT_a$ and closest points
$POINT_a <-> POINT_b$	$SQRT((\ POINT_a.x \ ? \ POINT_b.x \)^2 + (\ POINT_a.y \ ? \ POINT_b.y \)^2)$

Next, you can determine the spatial relationships between two objects using the left-of (<<), right-of (>>), below (<^), and above (>^) operators.

There are three overlap operators. && evaluates to TRUE if the left operand overlaps the right operand. The &> operator evaluates to TRUE if the leftmost point in the first operand is left of the rightmost point in the second operand. The &< evaluates to TRUE if the rightmost point in the first operand is right of the leftmost point in the second operand.

The intersection operator (#) returns the intersecting points of two objects. You can find the intersection of two BOXes, or the intersection of two LSEGs. The intersection of two BOXes evaluates to a BOX. The intersection of two LSEGs evaluates to a single POINT.

Finally, the ?# operator evaluates to TRUE if the first operand intersects with or overlaps the second operand.

The final set of geometric operators determines the relationship between a line segment and an axis, or the relationship between two line segments.

The ?- operator evaluates to TRUE if the given line segment is horizontal (that is, parallel to the x-axis). The ?| operator evaluates to TRUE if the given line segment is vertical (that is, parallel to the y-axis). When you use the ?- and ?| operators with a line segment, they function as prefix unary operators. You can also use the ?- and ?| operators as infix binary operators (meaning that the operator appears between two values), in which case they operate as if you specified two points on a line segment.

The `?-|` operator evaluates to `TRUE` if the two operands are perpendicular. The `?||` operator evaluates to `TRUE` if the two operands are parallel. The perpendicular and parallel operators can be used only with values of type `LSEG`.

The final geometric operator (`@@`) returns the center point of an `LSEG`, `PATH`, `BOX`, `POLYGON`, or `CIRCLE`.

Table 2.24. Proximity Operators for the Geometric Types

Data Types	Valid Operators (θ)
<code>POINT θ POINT</code>	<code><-> << <^ >> >^ ?- ? @</code>
<code>POINT θ LSEG</code>	<code>## <-> @</code>
<code>POINT θ BOX</code>	<code>## <-> @</code>
<code>POINT θ PATH</code>	<code><-> @</code>
<code>POINT θ POLYGON</code>	<code>@</code>
<code>POINT θ CIRCLE</code>	<code><-> @</code>
<code>LSEG θ LSEG</code>	<code># ## < <-> <= <> = > >= ?# ?- ? </code>
<code>LSEG θ BOX</code>	<code>## <-> ?# @</code>
<code>BOX θ POINT</code>	<code>* + - /</code>
<code>BOX θ BOX</code>	<code># && &< &> < <-> << <= <^ = > >= >> >^ ?# @ ~ ~=</code>
<code>PATH θ POINT</code>	<code>* + - / ~</code>
<code>PATH θ PATH</code>	<code>+ < <-> <= = > >= ?#</code>
<code>POLYGON θ POINT</code>	<code>~</code>

Data Types**Valid Operators (θ)**

POLYGON θ POLYGON && <& >& <-> >> << @ ~ ~=

CIRCLE θ POINT * + - / ~

CIRCLE θ POLYGON <->

CIRCLE θ CIRCLE && <& >& > <-> << <= <> <^ = > >= >> >^ @ ~ ~=

Table 2.25 summarizes the names of the proximity operators for geometric types.

Table 2.25. Geometric Proximity Operator Names

Data Types**Valid Operators (θ)**

Intersection or point count(for polygons)

Point of closest proximity

<-> Distance Between

<< Left of?

>> Right of?

<^ Below?

>^ Above?

&& Overlaps

&> Overlaps to left

&< Overlaps to right

?# Intersects or overlaps

Data Types	Valid Operators (θ)
------------	---------------------

@	Contained in
~	Contains
~=	Same as
?-	Horizontal
?	Vertical
?-	Perpendicular
?	Parallel
@@	Center

Original em:

<http://etutorials.org/SQL/Postgresql/Part+I+General+PostgreSQL+Use/Chapter+2.+Working+with+Data+in+PostgreSQL/Geometric+Data+Types/>

Outro capítulo de e-book online:**Using Geometric Data Types**

PostgreSQL provides a set of data types you can use for storing geometric information efficiently. These data types are included in PostgreSQL's core distribution and are widely used and admired by many people. In addition to the data types themselves, PostgreSQL provides an index structure based on R-trees, which are optimized for performing spatial searching.

PHP has a simple interface for generating graphics. To generate database-driven indexes, you can combine PostgreSQL and PHP. In this section you will see how to implement the "glue" between PHP and PostgreSQL.

The goal of the next example is to implement an application that extracts points from a database and displays them in an image. To store the points in the database, you can create a table:

```
phpbook=# CREATE TABLE coord (comment text, data point);
CREATE
```

Then you can insert some values into the table:

```
phpbook=# INSERT INTO coord VALUES ('no comment', '20,20');
INSERT 19873 1
phpbook=# INSERT INTO coord VALUES ('no comment', '120,99');
INSERT 19874 1
phpbook=# INSERT INTO coord VALUES ('no comment', '137,110');
INSERT 19875 1
phpbook=# INSERT INTO coord VALUES ('no comment', '184,178');
INSERT 19876 1
```

In this example four records have been added to the table. Now that some data is in the database, you can start working on a script that generates the dynamic image:

```
<?php

include("objects/point.php");

header ("Content-type: image/png");
$im = @ImageCreate (200, 200)
    or die ("Cannot Generate Image");

$white = ImageColorAllocate ($im, 255, 255, 255);
$black = ImageColorAllocate ($im, 0, 0, 0);
ImageFill($im, 0, 0, $white);

# connecting to the database
$dbh = pg_connect("dbname=phpbook user=postgres");
if      (! $dbh)
{
    exit ("an error has occurred<br>");
}

# drawing border
$points = array(0,0, 0,199, 199,199, 199,0);
ImagePolygon($im, $points, 4, $black);

# selecting points
$sql = "SELECT comment, data FROM coord";
$result = pg_exec($dbh, $sql);
$rows = pg_numrows($result);

# processing result
for      ($i = 0; $i < $rows; $i++)
{
    $data = pg_fetch_array ($result, $i);
    $mypoint = new point($data["data"]);
    $mypoint->draw($im, $black, 3);
}

ImagePng ($im);

?>
```

First a library is included. This library will be discussed after you have gone through the main file of the script. Then an image is generated and two colors are allocated.

The background is painted white, and after that the connection to the database is established. To see where the image starts and where it ends, a rectangle is drawn that marks the borders of the image.

In the next step all records are retrieved from the table called `coord`. All records in the table are processed using a simple loop. In the loop a constructor is called. After the object has been created, a method called `draw` is called. This method adds the point to the image. Finally the image is sent to the browser.

In the next listing you can take a look at the library you will need to run the script you have just seen:

```
<?php

# working with points
class point
{
    var $x1;
    var $y1;

    # constructor
    function point($string)
    {
        $string = ereg_replace("\\(\\)*", "", $string);
        $tmp = explode(",", $string);
        $this->x1 = $tmp[0];
        $this->y1 = $tmp[1];
    }

    # drawing a point
    function draw($image, $color, $size)
    {
        imagefilledrectangle ($image,
                               $this->x1 - $size/2,
                               $this->y1 - $size/2,
                               $this->x1 + $size/2,
                               $this->y1 + $size/2,
                               $color);
    }
}

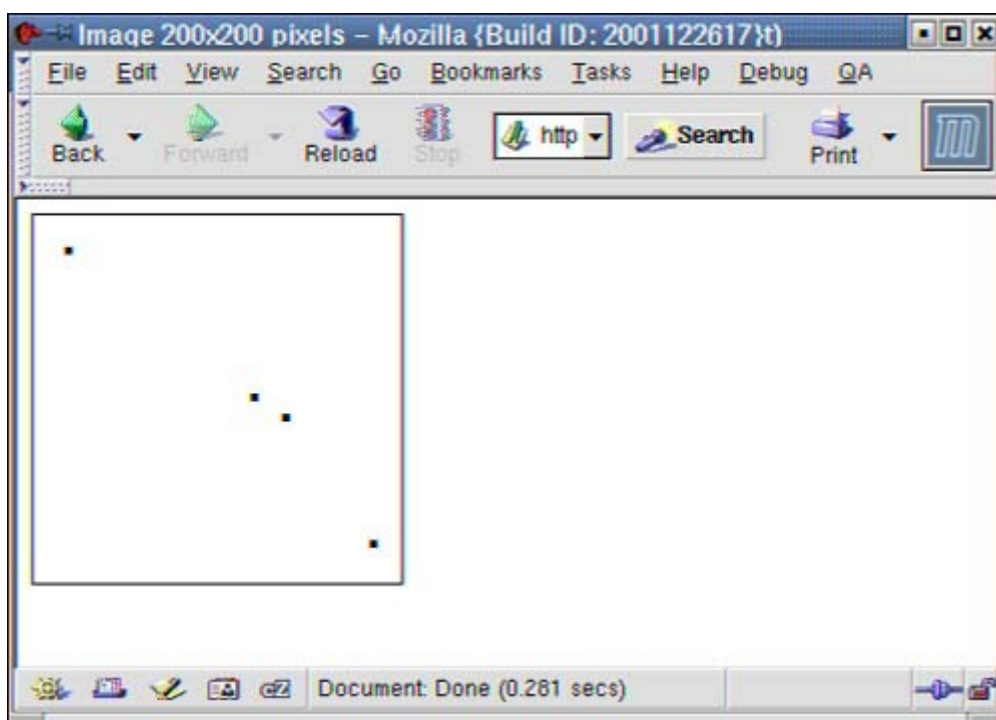
?>
```

The class called `point` contains two variables. `$x1` contains the x coordinate of the point. `$y1` contains the y coordinate of the point. Let's take a closer look at the constructor.

One parameter has to be passed to the function. This parameter contains the data returned by PostgreSQL. In the next step the data is transformed and the two coordinates of the point are extracted from the input string. Finally the two values are assigned to `$this`.

In addition to the constructor, a function for drawing a point has been implemented. Because a point is too small, a filled rectangle is drawn. The coordinates of the rectangle are computed based on the data retrieved from the database by the constructor. [Figure 16.11](#) shows what comes out when running the script.

Figure 16.11. Dynamic points.



Points are the easiest data structure provided by PostgreSQL and therefore the PHP class you have to implement is easy as well. A more complex yet important data structure is polygons. Polygons consist of a variable number of points, so the constructor as well as the drawing functions are more complex. Before taking a look at the code, you have to create a table and insert some data into it:

```
phpbook=# CREATE TABLE mypolygon (data polygon);
CREATE
```

In the next step you can add some data to the table:

```
phpbook=# INSERT INTO mypolygon VALUES ('10,10, 150,20, 120,160'::polygon);
INSERT 19902 1
phpbook=# INSERT INTO mypolygon VALUES ('20,20, 160,30, 120,160, 90,90');
INSERT 19903 1
```

Two records have been added to the table. Let's query the table to see how the data is returned by PostgreSQL:

```
phpbook=# SELECT * FROM mypolygon;
           data
-----
((10,10),(150,20),(120,160))
((20,20),(160,30),(120,160),(90,90))
(2 rows)
```

The target of the next piece of code is to transform the data returned by the database and make something useful out of it. The next listing contains a class called `polygon` used for processing polygons:

```
<?php

# working with polygons
```

```

class polygon
{
    var $x;
    var $y;
    var $number;

    # constructor
    function polygon($string)
    {
        $string = ereg_replace("\\(\\*)*", "", $string);
        $tmp = explode(",", $string);
        $this->number = count($tmp);
        for      ($i = 0; $i < $this->number; $i = $i + 2)
        {
            $this->x[$i/2] = $tmp[$i];
            $this->y[$i/2] = $tmp[$i + 1];
        }
    }

    # drawing a polygon
    function draw($image, $color)
    {
        for      ($i = 0; $i < $this->number - 1; $i++)
        {
            imageline($image, $this->x[$i], $this->y[$i],
                      $this->x[$i + 1], $this->y[$i + 1], $color);
        }
        imageline($image, $this->x[$this->number - 1],
                  $this->y[$this->number - 1], $this->x[0],
                  $this->y[0], $color);
    }
}
?>

```

\$x will be used to store a list of x coordinates and \$y will contain the list of y coordinates extracted from the points defining the polygon. \$number will contain the number of points a polygon consists of.

After defining the variables belonging to the polygon, the constructor has been implemented. With the help of regular expressions, the data returned by PostgreSQL is modified and can easily be transformed into an array of values. In the next step the values in the array are counted and the various coordinates are added to \$this->x and \$this->y.

In addition to the constructor, a function for drawing the polygon has been implemented. The function goes through all points of the polygon and adds lines to the image being generated.

Now that you have seen how to process polygons, you can take a look at the main file of the application:

```

<?php

include("objects/polygon.php");
header ("Content-type: image/png");
$im = @ImageCreate (200, 200)
    or die ("Cannot Generate Image");

$white = ImageColorAllocate ($im, 255, 255, 255);

```

```
$black = ImageColorAllocate ($im, 0, 0, 0);
ImageFill($im, 0, 0, $white);

# connecting to the database
$dbh = pg_connect("dbname=phpbook user=postgres");
if (! $dbh)
{
    exit ("an error has occurred<br>");
}
# drawing border
$points = array(0,0, 0,199, 199,199, 199,0);
ImagePolygon($im, $points, 4, $black);

# selecting points
$sql = "SELECT data FROM mypolygon";
$result = pg_exec($dbh, $sql);
$rows = pg_numrows($result);

# processing result
for ($i = 0; $i < $rows; $i++)
{
    $data = pg_fetch_array ($result, $i);
    $mypoly = new polygon($data["data"]);
    $mypoly->draw($im, $black);
}

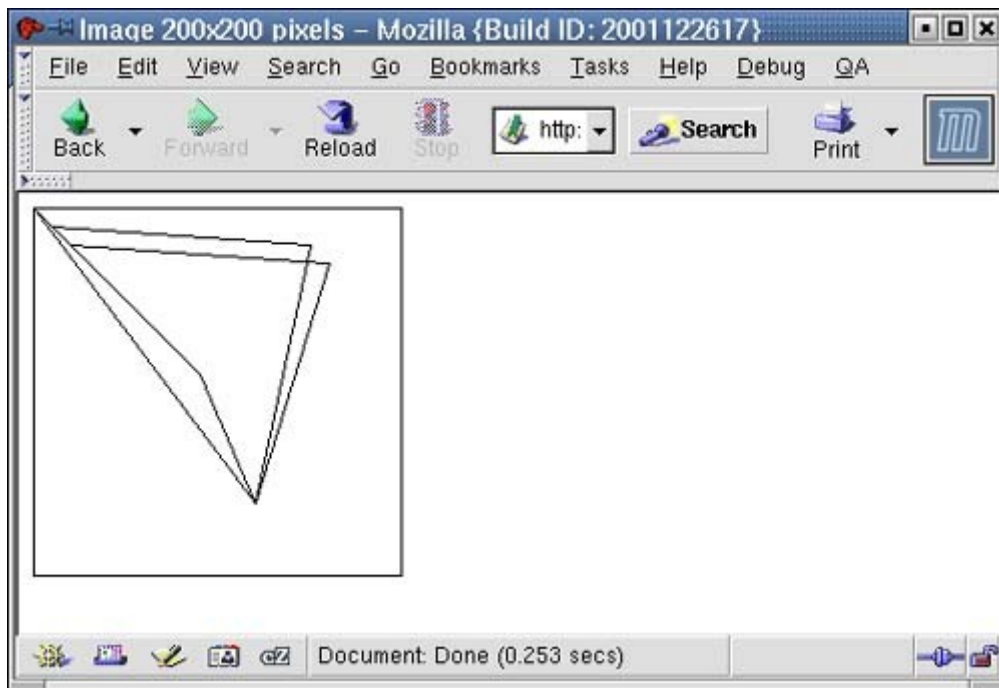
ImagePng ($im);

?>
```

First the library for processing polygons is included. In the next step an image is created, colors are allocated, the background of the image is painted white, and a connection to the database is established. After drawing the borders and retrieving all records from the database, the polygons are displayed using a loop. Inside the loop the constructor is called for every record returned by PostgreSQL. After the constructor, the `draw` function is called, which adds the polygon to the scenery.

If you execute the script, you will see the result as shown in [Figure 16.12](#).

Figure 16.12. Dynamic polygons.



Classes can be implemented for every geometric data type provided by PostgreSQL. As you have seen in this section, implementing a class for working with geometric data types is truly simple and can be done with just a few lines of code. The major part of the code consists of basic things such as creating images, allocating colors, or connecting to the database. The code that is used for doing the interaction with the database is brief and easy to understand. This should encourage you to work with PHP and PostgreSQL's geometric data types extensively.

Fonte: http://jlbtc.eduunix.cn/index/html/php/Sams%20-%20PHP%20and%20PostgreSQL%20Advanced%20Web%20Programming/0672323826_ch16lev1sec2.html