

SGBD PostgreSQL



Ribamar FS

Fortaleza, 21 de maio de 2018

Sumário

1 - Conceitos Importantes no PostgreSQL.....	6
Introdução ao PostgreSQL.....	7
1.1 - Resumo da História do PostgreSQL.....	7
1.2 - Características Avançadas do PostgreSQL.....	8
1.3 - Limites atuais do PostgreSQL.....	9
1.4 - Licença.....	9
1.5 - Quem oferece Suporte ao PostgreSQL no Brasil e quem o usa.....	9
2 - Database FAQs.....	11
3 - Um panorama das novidades previstas para o PostgreSQL 10.....	41
Novidades sobre o Postgresql.....	42
Replicação em cascata.....	47
Novos tipos de dados.....	48
Index-only scans.....	48
Novo utilitário: “pg_receivexlog”.....	49
Melhorias na contrib “pg_stat_statements”.....	50
Background Checkpointer.....	50
Melhor escalabilidade vertical.....	51
4 - Projeto, Administração e Programação.....	54
Dimensionamento de Hardware.....	72
5 - Ferramentas.....	80
Expressões Regulares para uso em Modelagem de Bancos de Dados.....	103
6 - Instalação do PostgreSQL 8.3 no Windows.....	113
Instalação do PostgreSQL no Linux.....	125
Definições.....	126
Entendendo as constraints e a integridade referencial.....	132
Funções com Data e Hora.....	141
7 - Relacionamentos entre tabelas.....	142
Entendendo a Herança de tabelas.....	146
8 - Tipos de Dados no PostgreSQL.....	150
9 - Agrupando Registros.....	160
10 - EXPRESSÃO CASE.....	161
CASTS - Conversão explícita de tipos.....	163
11 - Trabalhando com Conjuntos de Dados.....	163
12 - Trabalhando com SQL.....	165
13 - Utilizando SQL para selecionar, filtrar e agrupar registros.....	180
DDL - Linguagem de Definição de Dados.....	181
DCL - Linguagem de Controle de Dados.....	182
DTL - Linguagem de Transação de Dados.....	182
DQL - Linguagem de Consulta de Dados.....	183
Cláusula <i>DISTINCT</i>	189
14 - Desvendando o SELECT.....	195
15 - Entendendo e Utilizando sub-consultas.....	197
16 - Reuso de Código: Utilizando Funções.....	205
17 - Consultanto dados em múltiplas tabelas.....	220
Cláusula JOIN.....	223
18 - Visões (views).....	231

19 - Junção entre tabelas no PostgreSQL - Daniel Oslei.....	237
20 - Transações.....	241
Isolamento serializável versus verdadeira serialidade.....	247
Trabalhando com Nulos.....	251
Utilizando Operadores.....	263
21 - Catálogo do Sistema.....	285
22 - Tipos Geométricos.....	315
23 - Polígonos.....	316
Chapter: Geometric Data Types.....	320
Using Geometric Data Types.....	321
Figure 16.12. Dynamic polygons.....	326
24 - PostgreSQL Dicas.....	335
COALESCE: Trate decisões envolvendo campos nulos!.....	335
explain.depesz: Encontre a Causa da Lentidão em suas Consultas!.....	337
Simples e Útil: Visões Materializadas no PostgreSQL!.....	337
Beltrano: Base de Dados em Português para o PostgreSQL.....	339
Faça Você Mesmo: Diagramas de Classe com as Tabelas do Catálogo do PostgreSQL.....	340
Você REALMENTE Sabe Lidar com Valores Nulos no PostgreSQL?.....	348
Formatação de Data e Hora com as funções TO_CHAR e TO_DATE.....	354
DtSQL: Ferramenta Front-End para Banco de Dados.....	357
Edição de SQL e Funções no PSQL.....	362
Produza Sequências Com a Função Generate_Series()!.....	364
Visão Geral da Arquitetura do PostgreSQL!.....	369
PGTUNE: Otimize a configuração do PostgreSQL!.....	369
Range Types: Novo recurso do Postgres 9.2!.....	372
Tratamento de Parâmetros de Funções com PL/PgSQL.....	377
Desenvolva suas Aplicações de Bancos Postgres com Wavemaker.....	381
Unlogged Tables: Funcionalidade para Aumento de Desempenho!.....	385
25 - PostgreSQL Cheat Sheet.....	393
26 - Prepared Statements.....	393
27 - Exercícios.....	395
Entendendo e Trabalhando com Clusters.....	395
Esquemas.....	397
Otimizador de Consultas.....	400
Restaurando um Template1 Corrompido usando o Template0.....	402
28 - Exemplo de Modelagem de Banco de Dados.....	403
29 - Normalização de uma tabela de CEP sem mesmo chave primária.....	404
Modelagem do Banco de Dados Pessoa.....	408
30 - Segurança.....	418
Segurança de Dados.....	418
Inundação.....	419
Desafios da Segurança de Informação.....	436
31 - Linguagens do Lado do Servidor.....	443
Funções no PostgreSQL.....	443
RETURN NEXT.....	506
EXIT.....	509
WHILE.....	510
FOR (variação inteira).....	511
Laço através do resultado da consulta.....	511

Captura de erros.....	515
Declaração de variável cursor.....	517
Abertura de cursor.....	518
OPEN FOR SELECT.....	518
OPEN FOR EXECUTE.....	518
Abertura de cursor ligado.....	518
Utilização de cursores.....	519
FETCH.....	519
CLOSE.....	519
Retornar cursor.....	520
Validação de CPF com PL/ PgSQL.....	540
32 - Gatilhos.....	542
33 - Rules.....	570
Triggers (Gatilhos).....	574
RULES.....	579
34 - Exemplos de Funções em SQL no PostgreSQL.....	582
35 - Expressões Regulares para uso em Modelagem de Bancos de Dados.....	616
36 - Tipos e Domínios.....	625
37 - Fases de um Projeto de Banco de Dados.....	628
38 - Ferramentas.....	632
Validação e Geração de IE em JavaScript.....	632
Links Úteis para Modelagem.....	632
39 - Modelagem de Bancos de Dados.....	634
40 - Modelando um Banco Pessoa.....	640
Integridade Referencial.....	659
Algumas Demonstrações sobre Normalização.....	662
Normalizando Tabelas com Leandro Dutra.....	664
Modelo Relacional segundo C. J. Date.....	665
Fases de um Projeto de Banco de Dados.....	666
Tipos e Domínios.....	668
41 - Melhorando a Performance do PostgreSQL.....	670
Cinco Princípios de Hardware para Configurar o seu Servidor PostgreSQL.....	676
Conexão.....	678
Memória.....	678
Disco e WAL.....	680
Planejador de Consultas.....	680
Logging.....	681
Autovacuum e você.....	681
Usando o Vacuum e Analyze.....	703
42 - Gerenciando Bancos de Dados no PostgreSQL.....	708
43 - Administrando o PostgreSQL pela linha de comando (psql).....	715
44 - Administração com o cliente web Adminer.....	717
45 - Instalação do PostgreSQL através dos Fontes.....	718
Contribs.....	725
Instalando PostgreSQL no Slackware Linux.....	728
Instalação do PostgreSQL no CentOS.....	730
Instalação do PostgreSQL no OpenBSD.....	732
Testes de regressão.....	732
46 - Backup lógico e físico do postgresql.....	739
Sintaxe dos comandos para backup.....	767

Criar banco limpo, tendo como base o template0:.....	774
Script para backup do PostgreSQL.....	775
Usando o Comando Copy.....	776
Migração de um banco de dados no 8.3 para o PostgreSQL 9.5.....	777
IMPORTAR SCRIPTS TIPO CSV E SQL PARA O POSTGRESQL.....	781
47 - Configurações do PostgreSQL.....	783
48 - Entendendo e trabalhando com CLUSTERS no PostgreSQL.....	803
49 - Trabalhando com Esquemas no PostgreSQL.....	812
Schemas ou Databases?.....	818
50 - Manutenção do PostgreSQL.....	821
Tudo que você sempre quis saber sobre discos em servidores PostgreSQL e tinha vergonha de perguntar.....	825
RAID.....	825
Discos.....	828
Controladoras de discos.....	830
Tipos de arquivos.....	831
Particionamento.....	835
Como distribuir as partições nos discos existentes.....	836
Discos no PostgreSQL.....	837
51 - Monitorando as Atividades do Servidor do PostgreSQL.....	840
Explain.....	864
Ver as estatísticas coletadas.....	866
Características de um bom hardware para servidor.....	869
52 - TableSpaces.....	870
Trabalhando com Tablespace no PostgreSQL.....	877
53 - Criação de grupos de usuários no PostgreSQL 9.6.3.....	883
Exemplo de Uso de Usuários e Privilégios.....	898
54 - Conectividade.....	901
Openoffice2 Base.....	914
55 - Informações Introdutórias sobre Redes de Computadores.....	916
Referência rápida de máscara de redes.....	919
Endereços reservados para uso em uma rede Privada.....	920

1 - Conceitos Importantes no PostgreSQL

SGBD - Sistema Gerenciador de Bancos de Dados. São sistemas especializados no trabalho com bancos de dados, com muitos recursos para isso.

Cluster - coleção de bancos de dados. Todos os bancos de dados de uma instalação do servidor.

Banco - coleção de tabelas, usuários, tipos de dados, funções, índices, sequências e outros objetos.

Tabela - coleção de registros ou linhas

Registro - coleção de campos ou colunas

Campo - perção da tabela que armazena informações somente de um tipo

Ordem - os registros são guardados na ordem em que são armazenados, mas o SQL não garante a ordem na recuperação. Precisa ser especificada explicitamente.

Relação - Termo matemático para tabela.

Existem outras formas de organizar dados diferentes de tabelas.

Arquivos e diretórios é uma delas e orientação a objetos outra.

Uma tabela num banco de dados relacional parece com uma tabela no papel ou com uma planilha.

Existem SGBDs modernos orientados a objetos.

Conexão - cada usuário conecta somente a um único banco de dados por vez.

Convenções

Nomes de bancos de dados, de tabelas e de campos devem ser escritos em minúsculas e palavras compostas com sublinhado as separando. Isto porque o PostgreSQL é case sensitivo.

O SQL não é sensível ao caso, a não ser que os identificadores estejam escritos entre aspas duplas.

MÁXIMO DE CAMPOS DE UMA TABELA

Uma tabela pode conter no máximo de 250 a 1600 campos, dependendo do tipo de dados. Claro que jamais devemos usar uma tabela tão grande.

Cada tabela deve conter informações somente sobre um único assunto.

Ao tentar apagar uma tabela que não existe um erro é disparado. Para evitar isso:

```
drop table if exists nometabela;
```

Palavras Reservadas:

oid

tableoid

xmin

xmax

cmin

cmax

ctid

Esclarecimento

Desculpem por alguns prováveis erros que encontrar. Caso queira me enviar eu corrigirei, mas quando terminei e vi o tamanho que ficou não tive ânimo para as correções. Espero que seja útil apesar disso.

Introdução ao PostgreSQL

1. Resumo da História do PostgreSQL
2. Características Avançadas
3. Limites do PostgreSQL
4. Principais Atribuições de um DBA
5. Suporte ao PostgreSQL no Brasil e Quem Usa

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 e 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.

INGRES - 1977

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 - 08/10/2002 (do 1.09 passou para o 6)

PostgreSQL 7 - 09/05/2000

PostgreSQL 8 - 17/01/2005 (primeira versão for Windows)

PostgreSQL 8.1 - 05/11/2005

PostgreSQL 8.2 - 02/12/2006

PostgreSQL 8.3 - 01/02/2008 (versão atual)

PostgreSQL 8.4 – 27/06/2009

PostgreSQL 9.0 – 17/09/2010

PostgreSQL 9.1 – 08/09/2011

PostgreSQL 9.2 – 10/09/2012

PostgreSQL 9.3 – 02/09/2013

PostgreSQL 9.4 – 16/12/2014

PostgreSQL 9.5 – 04/01/2016

PostgreSQL 9.6.3 – 08/05/2017

Fonte: <https://www.postgresql.org/ftp/source/>

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 atuais do PostgreSQL

Limite	Value
Máximo tamanho de banco	Ilimitado
Maior tabela	32 TB
Maior registro	1.6 TB
Maior campo	1 GB
Máximo de registros por tabela	Ilimitado
Máximo de campos por tabela	250 - 1600 depende do tipo de campo
Máximo de índices por tabela	Ilimitado

Fonte: <http://www.postgresql.org/about/>

1.4 - 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.

1.5 - Quem oferece Suporte ao PostgreSQL no Brasil e quem o usa

Empresas que Patrocinam o PostgreSQL:

- InterpriseDB
- Fujitsu
- 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 consultores que oferecem suporte ao PostgreSQL:

http://www.opensourcexperts.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
- Governo do Estado do Ceará adotou o PostgreSQL como SDGB oficial.
- O DETRAN-CE migrou de Oracle para PostgreSQL

2 - Database FAQs

- [O que é um banco de dados?](#)
- [O que é um banco de dados relacional?](#)
- [O que é uma lima plana?](#)
- [O que é SQL?](#)
- [O que é um banco de dados transacionais?](#)
- [O que é um procedimento armazenado?](#)
- [O que é um banco de dados acionar?](#)
- [O que é a replicação de dados?](#)
- [O que é um dicionário de dados?](#)
- [O que é o banco de dados normalização?](#)
- [O que é um BLOB?](#)
- [O que é software livre banco de dados?](#)
- [O que é ODBC?](#)
- [O que é um DSN?](#)
- [Onde posso obter um driver ODBC manager?](#)
- [Onde posso obter um driver ODBC?](#)
- [Onde posso obter um driver ODBC MySQL?](#)
- [Onde posso obter um driver ODBC Oracle?](#)
- [O que é JDBC?](#)
- [Quais são as bases de dados Java?](#)
- [O que é um armazém de dados?](#)
- [O que é a mineração de dados?](#)
- [O que é a gestão dos dados?](#)
- [O que é um banco de dados multimídia?](#)

http://www.tech-faq.com/lang/pt/database.shtml&usg=ALkJrhhJgATIV9zW_BYqDU9bRQySuDafFw

Obs.: conteúdo não mais é encontrado no site acima, por conta disso o mantendo aqui.

O que é um Banco de Dados?

Um dos termos tecnologia que a maioria das pessoas se acostumaram a ouvir, quer no trabalho ou ao mesmo tempo a navegar na Internet é o banco de dados. A base de dados a ser utilizada uma técnica extremamente prazo, porém com o surgimento de sistemas de computadores e de tecnologia da informação em toda a nossa cultura, a base de dados tornou-se um agregado familiar prazo.

A definição de uma base de dados é uma colecção de registos estruturados ou dados que são armazenados em um sistema de computador. Para que um banco de dados para ser verdadeiramente funcional, não deve apenas armazenar grandes quantidades de

registros bem, mas ser acessadas com facilidade. Além disso, novas informações e mudanças também devem ser bastante fácil de entrada. A fim de ter um sistema de base de dados altamente eficiente, é necessário incorporar um programa que gere as consultas e as informações armazenadas no sistema. Isto é normalmente referido como SGBD ou de um Sistema de Gestão de Dados. Além destas características, que são criados todos os bancos de dados deve ser construído com alta integridade dos dados e a capacidade de recuperar os dados se hardware falhar.

Tipos de Bancos de dados

Existem vários tipos comuns de bancos de dados, cada tipo de base de dados tem o seu próprio modelo dados (como os dados são estruturados). Eles incluem: Modelo Flat, modelo hierárquico, modelo relacional e Rede Modelo.

O apartamento modelo de dados

Em um apartamento modelo banco de dados, existe uma bidimensional (plana estrutura) array de dados. Por exemplo, há uma coluna de informações e dentro desta coluna presume-se que cada item serão dados relacionados com os outros. Por exemplo, um apartamento modelo de dados inclui apenas o código postal. Dentro do banco de dados, haverá apenas uma coluna e cada nova linha dentro de uma coluna que será um novo CEP.

O modelo hierárquico de dados

O modelo hierárquico de dados assemelha uma árvore como estrutura, tais como Microsoft Windows forma como organiza pastas e arquivos. Em um modelo hierárquico banco de dados, cada ligação ascendente está aninhada, a fim de manter os dados organizados em uma ordem específica em um mesmo nível lista. Por exemplo, um banco de dados de vendas hierachal, maio lista cada dia de vendas como um processo separado. Dentro deste arquivo estão todos encaixados das vendas (mesmos tipos de dados) para o dia.

O modelo de rede

Em um modelo de rede, que é a característica que define um registro é armazenado em um link para outros registros - em efeito rede. Estas redes (ou por vezes referido como ponteiros) pode ser uma variedade de diferentes tipos de informações, tais como números de nó ou até mesmo um disco endereço.

O modelo relacional

O modelo relacional é o mais popular tipo de banco de dados e uma ferramenta extremamente poderosa, não só para armazenar informações, mas para acessá-lo também. Bancos de dados relacionais são organizadas as tabelas. A beleza de uma mesa é a de que a informação pode ser acessada ou adicionados sem reorganizar as tabelas. Uma tabela pode ter muitos registros e cada registro pode ter muitos domínios.

Os quadros são muitas vezes chamado de relação. Por exemplo, uma empresa pode ter um banco de dados chamado encomendas dos clientes, dentro desta base de dados será

várias tabelas ou relações relativos a todas as encomendas dos clientes. Os quadros podem incluir informação sobre os clientes (nome, morada, contactos, informações, o cliente número, etc) e de outras tabelas (relações), tais como ordens que o cliente comprou anteriormente (isto pode incluir o item de número, item descrição, quantidade pagamento, método de pagamento, etc). Note-se que cada registro (grupo de campos) em um banco de dados relacional tem a sua própria chave primária. Uma chave primária é a única área que facilita a identificação de um registro.

Bancos de dados relacionais usar um programa chamado interface SQL ou Standard Query Language. SQL é actualmente utilizado em praticamente todos os bancos de dados relacionais. Bancos de dados relacionais são extremamente fácil de personalizar a caber quase qualquer tipo de armazenamento de dados. Você pode facilmente criar relações de itens que você vende, os empregados que trabalham para a sua empresa, etc

Acessando informações utilizando um banco de dados

Enquanto armazenamento de dados é uma grande característica das bases de dados, para muitos usuários de dados a característica mais importante é rápido e simples recuperação da informação. Em uma base de dados relacional, que é extremamente fácil de puxar os esclarecimentos relativos a um empregado, bancos de dados relacionais, mas também acrescentar o poder de executar consultas. Consultas são pedidos para puxar tipos específicos de informação e mostrar-lhes, quer no seu estado natural ou criar um relatório utilizando os dados. Por exemplo, se você tivesse um banco de dados de empregados e que incluiu tabelas salariais e de trabalho, como a descrição, você pode facilmente executar uma consulta de empregos que pagam mais de uma determinada quantia. Não importa qual é o tipo de informação que guarda no seu banco de dados, consultas podem ser criados usando SQL para ajudar a responder perguntas importantes.

Conservação de um Banco de Dados

Bancos de dados podem ser muito pequenas (menos de 1 MB) ou extremamente grandes e complexos (como em muitos terabytes governo bases de dados), porém todos os bancos de dados são normalmente armazenados e localizados no disco rígido ou outros tipos de dispositivos de armazenamento e são acessadas via computador. Grandes bases de dados podem exigir servidores separados e locais, no entanto muitas pequenas bases de dados pode caber facilmente como arquivos localizados no disco rígido do seu computador.

Garantir um Banco de Dados

Obviamente, muitas bases de dados confidenciais e armazenar informações importantes que não devem ser facilmente acessado apenas por alguém. Muitas bases de dados

exigem senhas e outros recursos de segurança, a fim de aceder à informação. Embora alguns dados podem ser acessados através da Internet através de uma rede, outras bases de dados são sistemas fechados e só pode ser acessado no local.

O que é um banco de dados relacional?

Um banco de dados relacional armazena dados em tabelas separadas em vez de colocar todos os dados em uma grande mesa.

Um banco de dados relacional permite então do administrador do banco de dados (DBA's) para definir as relações entre essas tabelas.

Estas relações permitem DBA's para combinar dados de várias tabelas de consulta e de apresentação de relatórios.

Isso é realizado através da utilização de *chaves*, que são campos de dados utilizada para o identificar registros em uma tabela específica.

Banco de dados relacional tecnologia permite bases de dados para ser maior, mais rápido e mais eficiente.

O conceito de um banco de dados relacional foi inicialmente desenvolvido pelo Dr. F. Edger (Ted) Codd em *um Modelo Relacional de Dados para Grandes Bancos de dados partilhada*, em 1970.

Dr. Codd definiu treze normas que devem ser satisfeitas antes de uma base de dados pode ser considerado como um banco de dados relacional:

0. Um SGBD relacional deve ser capaz de gerir bases de dados relacionais inteiramente através das suas capacidades.
1. Informações regra - Todas as informações em um banco de dados relacional (incluindo nomes tabela e coluna) é representado explicitamente como valores nas tabelas.
2. O acesso garantido - Cada valor em um banco de dados relacional é garantida a ser acessível por meio de uma combinação do nome da tabela, chave primária valor, e coluna de nome.
3. Sistemática valor nulo apoio - O SGBD fornece apoio sistemático para o tratamento de valores nulos (desconhecido ou inaplicável dados), distinta dos valores padrão, e independente de qualquer domínio.
4. Ativo, relacionais catálogo on-line - A descrição da base de dados eo seu conteúdo é representada no nível lógico como quadros e pode, portanto, ser consultada a base de dados usando linguagem.
5. Dados abrangentes sublanguage - Pelo menos uma língua deve ter apoiado uma sintaxe bem definida e ser abrangente. Deve apoiar dados definição, manipulação, a integridade regras de autorização, e as transacções.
6. Ver actualização regra - Todas as opiniões que são teoricamente actualizada pode ser atualizado através do sistema.
7. Ajuste de nível de inserção, actualização e cancelamento - O SGBD não só apoia setlevel recuperações, mas também definir a nível de inserções, actualizações e exclusões.
8. Dados físicos independência - programas de aplicação e programas ad hoc são logicamente inalterados quando o acesso físico métodos ou estruturas de armazenagem são alteradas.
9. Lógico dados independência - programas de aplicação e programas ad hoc são logicamente inalteradas, na medida do possível, quando forem efectuadas alterações ao quadro estruturas.
10. Integridade independência - A base de dados língua deve ser capaz de definir regras integridade. Eles devem ser armazenados no catálogo on-line, e eles não podem ser contornados.

11. Distribuição independência - Aplicação programas ad hoc e os pedidos são logicamente inalterados quando os dados são distribuídos primeiro ou quando é redistribuído.

12. Nonsubversion - não deve ser possível para contornar as regras definidas através da integridade do banco de dados usando a linguagem de nível mais baixo línguas.

Alternativas para o banco de dados relacional modelo incluem o modelo hierárquicas, o modelo de rede, e ao objeto modelo.

O que é um Flat File?

Um computador, na sua definição mais simplificada, é nada mais que um dispositivo que armazena, processos, comunica, e manipula dados. Os dados estão no cerne de cada programa de computador, cada site, e ainda todos os vídeo games.

O computador tem revolucionou a vida moderna, permitindo um nível de velocidade e precisão para tratamento da informação que até ao momento nunca foi possível.

Computadores conseguir isto, pura e simplesmente a seguir um conjunto de instruções denominado código.

Um flat arquivo é um documento estático, planilha eletrônica, ou textual registro que contém dados que normalmente não é estruturalmente relacionadas. Flat arquivos são chamados assim porque há pouco que pode ser feito com as informações neles contidas, com exceção ler, armazenar, e de envio.

Flat arquivos são tipicamente conjuntos de dados básicos que são utilizados para a configuração de armazenamento de dados para aplicações e programas. A média computador usuário não irá normalmente vê-los muito. Não há lista de endereços básicos- como exemplos de um flat arquivo que pode ser usado, mas os usuários modernos tipicamente um desejo mais robusto e sofisticado método de chamar a sua informação do que a confiar na capacidade limitada de uma estrutura plana arquivo.

Flat arquivos são comumente usadas por encontrada na base de dados e sistemas de gestão, normalmente pertencentes a lima plana e organizados em bases de dados. Um arquivo de dados é realmente plana nada mais do que uma organização critério atribuído a uma amostra conjunto de arquivos plana.

Programadores provavelmente usar arquivos plana e plana arquivo de dados muito mais freqüência do que a média de computador quando construir aplicações em sistemas de gerenciamento de banco de dados como o MySQL, um popular DBS que suporta múltiplas linguagens de programação.

Flat arquivos também são comumente utilizados pelos desenvolvedores site para utilização no interior eles utilizam linguagens como PHP ou ASP.

Verificou-se que fixa ficheiros, uma vez que são simples arquivos de dados, ocupam muito menos espaço do que arquivos estruturados. No entanto, o utilitário leitura plana arquivos devem ser bastante sofisticado, na medida em que terá de saber o que fazer com o arquivo plana depois de ter sido acessado.

O que é SQL?

SQL (Structured Query Language) é a língua mais comum usada para acessar padronizado bases de dados.

O SQL foi originalmente desenvolvido pela IBM na década de 1970 para a sua DB2 RDBMS.

SQL versão 3 é oficialmente definida pela American National Standards Institute (ANSI) em ANSI SQL: 1999 standard.

A maior parte dos SGBD's existentes atualmente estão em conformidade com a norma anterior ANSI SQL92.

SQL é uma linguagem nonprocedural. Oracle processuais produz uma versão do SQL que designa PL / SQL.

SQL é suportado por todas as grandes sistema de base de dados em uso hoje em dia, incluindo o MySQL, PostgreSQL, Berkeley DB, Oracle, DB2, Sybase, Informix, e Microsoft SQL.

SQL é muitas vezes pronunciado "sequela".

O que é um banco de dados transacionais?

Um banco de dados transacionais é um SGBD onde escrever sobre operações do banco de dados são capazes de ser retirada caso não sejam preenchidos corretamente.

Se um sistema de base de dados transacionais perde energia elétrica a meio caminho através de uma operação, parcialmente concluída a operação irá ser retirada e banco de dados será restaurado para o estado em que foi iniciada antes da operação.

Imagine que um front-end aplicação está enviando um cliente para um sistema de base de dados. O front-end aplicação envia o pedido ao produto para o cliente e subtrair o produto a partir de inventário. O front-end de aplicação é de cerca de enviar o pedido para a criação de uma factura para o cliente e, de repente o front-end aplicação travamentos.

Um banco de dados transacionais pode então voltar a operação concluída parcialmente.

Uma alternativa para usar um banco de dados transacionais é a utilização operações atômicas.

O que é um procedimento armazenado?

Uma stored procedure é um conjunto de comandos SQL que tem sido compilados e armazenados no servidor de banco de dados.

Depois que o processo tenha sido armazenado "armazenadas", aplicações cliente pode executar o procedimento armazenado, uma e outra vez sem enviá-lo ao servidor de banco de dados novamente, e sem a compilá-lo novamente.

Procedimentos armazenados melhorar o desempenho, reduzindo o tráfego de rede e CPU load.

O que é um banco de dados acionar?

Uma base de dados é desencadear um [procedimento armazenado](#) que é invocado automaticamente quando um evento ocorre predefinidos.

Database triggers permitir DBA's (base de dados Administradores) para criar outros relacionamentos entre distintas bases de dados.

Por exemplo, a modificação de um registro em um banco de dados podem acionar a modificação de um registro em uma segunda base de dados.

O que é a replicação de dados?

Replicação de dados é a criação e manutenção de várias cópias da mesma base de dados.

Na maioria das implementações replicação de dados, mantém um banco de dados do servidor master cópia do banco de dados e servidores de banco de dados adicionais slave manter cópias da base de dados.

Escreve dados são enviados para o servidor e banco de dados master são então replicada pelo escravo servidores de banco de dados.

Database lê são divididas entre todos os servidores de base de dados, o que resulta em uma grande vantagem desempenho devido a partilha de carga.

O que é um dicionário de dados?

Um dicionário de dados é reservado um espaço dentro de uma base de dados que é usado para armazenar informações sobre a base de dados própria.

Um dicionário de dados podem conter informações como:

- Database design informação
- SQL procedimentos armazenados
- Usuário permissões
- Usuário estatísticas
- Database processar informação
- Dados estatísticos do crescimento
- Dados estatísticos desempenho

O que é o banco de dados normalização?

Database normalização é o processo de organização de dados em conjuntos distintos e exclusivos.

Os efeitos de normalização são os seguintes:

- Reduzir ou eliminar a duplicação de armazenamento de dados
- Organize os dados em uma estrutura lógica e eficaz

O processo de normalização envolve determinar quais dados devem ser armazenados em banco de dados cada tabela.

Por tradição, o processo de normalização através do trabalho envolve etapas bem definidas, chamadas *formas normais*.

Na Primeira forma normal (1NF) você eliminar a duplicação de colunas da mesma tabela, criar quadros separados para cada grupo de dados relacionados, e identificará cada linha com uma única coluna ou conjunto de colunas (as chaves primárias).

Na Segunda forma normal (2NF) você remover subconjuntos de dados que se aplicam a várias linhas de uma tabela, colocá-los em quadros separados, e criam relações entre estes novos quadros e os quadros originais através da utilização de chaves estrangeiras.

Na Terceira forma normal (3NF) você remover colunas que não estão dependentes da chave primária.

Adicional formas normais foram definidos, mas geralmente são menos utilizados. Estes incluem formas avançadas normal Quarta Forma Normal (4NF), Quinta Forma Normal (5NF), Boyce Codd Normal Form (BCNF), e Domain-Key Normal Form (DK / NF).

O que é um BLOB?

Um BLOB (Binary Large Object) é um grande pedaço de dados que são armazenados em um [banco de dados](#).

Um BLOB difere de dados no banco de dados regulares que não é forçada em uma determinada estrutura. Um campo de dados normais poderia ser estruturado para ser 14 caracteres e apenas aceitar letras minúsculas. Um campo BLOB não é normalmente restrito no tipo de conteúdo e de conteúdo podem ser vários gigabytes de tamanho. Normal campos de dados espaciais têm afectado por eles, eles são utilizados ou não. BLOB campos são apenas atribuídos espaço quando são utilizados.

BLOB campos são normalmente utilizados para armazenar gráficos, áudio, vídeo ou documentos.

BLOB campos podem ser adicionados, mudou, e excluído. No entanto, eles não podem ser pesquisados e manipulados com banco de dados padrão comandos.

O que é software livre banco de dados?

MySQL

O [MySQL](#) banco de dados é mais popular do mundo banco de dados open-source. Mais de seis milhões de instalações usar o banco de dados MySQL a potência de alto volume e de outros web sites sistemas de missão crítica das empresas-incluindo líderes da indústria como NASA, Yahoo, The Associated Press (AP), Suzuki, e Sabre Holdings.

MySQL é uma alternativa atraente ao alto custo, mais complexa base tecnológica. O premiado a sua fiabilidade, escalabilidade e velocidade tornam a escolha certa para uma ampla gama de serviços TI corporativos, desenvolvedores web e vendedores de software.

MySQL oferece várias vantagens-chave:

- Confiabilidade e desempenho. MySQL AB faz todas as versões iniciais da sua base de dados disponível para o servidor software de fonte aberta para a comunidade a fim de permitir vários meses de "batalha teste" antes que eles considerem prontos para uso em produção.
- Facilidade de uso e instalação. O MySQL arquitetura torna extremamente rápido e simples para personalizar. A única multi-motor de arquitetura de armazenamento corporativo MySQL dá a flexibilidade de que necessitam os clientes com um SGBD incomparável na estabilidade, velocidade, compacidade, e facilidade de implantação.
- Liberdade de Plataforma Lock-in. Ao proporcionar pronto acesso ao código-fonte, a abordagem do MySQL AB garante liberdade, impedindo assim lock-nos a um único fornecedor ou plataforma.
- Suporte multi-plataforma. MySQL está disponível em mais de vinte diferentes plataformas, incluindo todas as principais distribuições Linux, Unix, Microsoft Windows e Mac OS X.
- Milhões de desenvolvedores treinados e certificados. MySQL é mais popular do mundo open source banco de dados. Isso faz com que seja fácil de encontrar e knowledgable DBA's e desenvolvedores experientes.

PostgreSQL

[PostgreSQL](#) é extremamente escalável, SQL compliant, open-Fonte SGBD objeto-relacional. Com mais de 15 anos de desenvolvimento histórico, PostgreSQL está rapidamente se tornando a base de dados para empresa de facto nível soluções de fonte aberta.

PostgreSQL é um objeto-sistema de gerenciamento de banco de dados relacional (ORDBMS) com base em postgres, versão 4,2, desenvolvido na Universidade da Califórnia em Berkeley Computer Science Department. Postgres foi pioneira em muitos conceitos que só ficou disponível em sistemas de comunicações de dados muito mais tarde.

PostgreSQL é um descendente de fonte aberta do presente código original Berkeley. Suporta SQL92 e SQL99 e oferece muitas características modernas:

- Complexo queries
- Chaves estrangeiras
- Aciona
- Exibições
- Transacional integridade
- Multiversion concurrency controle

BerkeleyDB

Berkeley DB é uma das mais amplamente utilizadas desenvolvedoras de bases de dados em todo o mundo, é de código aberto e roda em todos os principais sistemas operacionais, incluindo Linux embutido, Linux, Unix, Microsoft Windows, Mac OS X, VxWorks e QNX.

Berkeley DB fornece os dados básicos de gestão funcionalidade, escalabilidade, potência e flexibilidade da empresa bancos de dados relacionais, mas sem o overhead de uma consulta camada de transformação. Combinado com a estabilidade e menor custo de suporte de código fonte aberto, Berkeley DB oferece muitas vantagens, incluindo:

- Administração custo zero elimina a necessidade de um DBA
- Dimensões menores (menos de 500 KB)
- A simplicidade de integração em um aplicativo
- Mais velocidade e melhor desempenho
- Menor complexidade e mais confiabilidade

Firebird

[Firebird](#) é um banco de dados relacional com muitos recursos ANSI SQL-99 que roda em Windows, Linux e uma variedade de plataformas Unix. Firebird oferece excelente concurrency, de alta performance, poderoso e idioma de suporte para database triggers e procedimentos armazenados. Firebird tem sido utilizado em sistemas de produção, ao abrigo de uma variedade de nomes, desde 1981.

Firebird é um projeto comercialmente independente de C / C + + programadores, assessores técnicos e suporte desenvolvendo e melhorando um banco de dados relacional multi-plataforma do sistema de gestão baseado no código fonte liberado pela Borland Software Corp sobre July25th, 2000 sob o InterBase Public License.

Firebird é completamente livre de qualquer registro, licenciamento ou implantação taxas. Firebird podem ser implantados livremente para uso com qualquer software de terceiros, independentemente de serem ou não comerciais.

O que é ODBC?

ODBC (Open Data Base Connectivity) é uma função que proporciona uma biblioteca comum API (Application Programming Interface) para sistemas de gerenciamento de banco de dados ODBC compatíveis.

ODBC SQL foi desenvolvido pelo Grupo de Acesso 1992.

ODBC funciona como um padrão de "shim" entre as aplicações que utilizam bases de dados e as bases de dados próprios.

Se um pedido for desenvolvido usando ODBC, o pedido será capaz de armazenar dados em qualquer sistema de gerenciamento de banco de dados que está equipado com um driver ODBC.

Drivers ODBC são muitas vezes desenvolvidas em sub-componentes:

- Um driver ODBC Manager
- ODBC Drivers

O pedido alega ODBC chamadas para o driver ODBC gerente.

O gerente escolhe o driver ODBC driver ODBC apropriado, que carrega condutor, ler ou escrever e enviar os pedidos que utilizam condutor.

Os processos ODBC driver ODBC a função exige, sustenta a solicitar ao banco de dados SQL, e retorna os resultados para a aplicação.

O que é um DSN?

Uma DSN (Data Source Name) é um identificador que define uma fonte de dados para um driver [ODBC](#).

Um DSN consiste de informações, tais como:

- Database nome
- Diretório
- Database condutor
- ID do usuário
- Senha

Sob Unix, DSN configuração é geralmente armazenada em / etc / odbc.ini.

Ao abrigo do Microsoft Windows, DSN configuração é normalmente armazenado no registro, embora possa também ser armazenados em arquivos de configuração em um arquivo. DSN extensão.

Onde posso obter um driver ODBC manager?

Os dois principais gestores driver ODBC para Unix são iODBC e unixODBC.

[iODBC \(Independent Open DataBase Connectivity\)](#) é uma plataforma independente Open Source execução de ambos os ODBC e X / Open caderno de encargos. iODBC proporciona tanto um gerente e um driver ODBC SDK que facilita o desenvolvimento da base de dados independente de aplicativos. iODBC inclui uma administração baseada em GTK a ferramenta.

iODBC tem sido portado para diversas plataformas, incluindo: Linux (x86, Itanium, Alpha, Mips, e StrongArm), Solaris (Sparc e X86), AIX, HP-UX (PA-RISC e Itanium), Digital UNIX, Dynix, Generic 5,4 UNIX, FreeBSD, MacOS 9, MacOS X, DG-UX e OpenVMS.

[unixODBC](#) prevê Unix com as mesmas aplicações ODBC 3,51 API e instalações disponíveis ao abrigo do Windows. unixODBC Manager fornece um driver que suporta a totalidade do ODBC e executa a API ODBC ODBC 3 a 2 com traduções para UNICODE ANSI conversão. unixODBC também inclui um conjunto de utilitários gráficos que permitem aos usuários especificar conexões para DBMSes para ser utilizado por aplicações, uma coleção de drivers ODBC, incluindo um texto simples baseado condutor, um [NNTP](#) condutor, um motorista Postgres e outros, e uma seleção de modelos e bibliotecas que a ajuda na construção de drivers ODBC. unixODBC funciona com o MySQL, Postgres, StarOffice / OpenOffice, Applixware, iHTML, PHP, Perl DBD:: ODBC, e muitas outras aplicações e motoristas. Connection pooling também é fornecida para aumentar a performance em aplicações como o PHP. unixODBC QT inclui uma base de administração GUI.

Onde posso encontrar um driver ODBC MySQL?

[MySQL Connector / ODBC](#) está o funcionário MySQL ODBC driver.

OpenLink Software oferece tanto [Single-Tier](#) e [Multi-Tier](#) MySQL ODBC drivers.

Onde posso obter um driver ODBC Oracle?

A [Oracle ODBC Drivers Dowload Page](#) é a fonte oficial Oracle para os drivers ODBC.

A [Easysoft ODBC Driver Oracle](#) melhora na bolsa Oracle ODBC motoristas, prevendo um melhor desempenho e mais fácil manutenção.

OpenLink Software oferece tanto [Single-Tier](#) e [Multi-Tier](#) Oracle ODBC drivers.

[DataDirect Connect para ODBC](#) é uma substituição Oracle ODBC driver que oferece melhor desempenho e mais fácil manutenção.

[Attunity](#) fornece uma base de dados que inclui um adaptador Oracle ODBC driver.

O que é JDBC?

JDBC (Java Data Base Connectivity) é uma API (Application Programming Interface) para a ligação a bases de dados a partir do ambiente Java.

JDBC é uma alternativa ao [ODBC](#). JDBC da interface Java é mais confortável do que para programadores Java da linguagem C ODBC interface.

JDBC é incluído em ambas as [J2SE](#) e [J2EE](#).

Se nenhum driver JDBC está disponível para as suas necessidades, uma ponte JDBC-ODBC pode ser usado para se conectar a um driver ODBC através do JDBC API. Java 2 inclui uma ponte JDBC-ODBC para Microsoft Windows e Solaris.

Quais são as bases de dados Java?

[HSQLDB](#)

HSQLDB é o principal mecanismo de banco de dados relacional SQL escrito em Java. HSQLDB tem um driver JDBC e suporta um subconjunto dos ricos ANSI-92 SQL (formato árvore BNF), acrescido SQL 99 e 2003 acessórios. HSQLDB oferece um pequeno (menos de 100k, em uma versão), fast motor de banco de dados que oferece tanto na memória e disco-com base em tabelas. Embutidos e servidor modos estão disponíveis. Além disso, inclui ferramentas como um servidor web mínimo, em memória de consulta e instrumentos de gestão (pode ser executado como applets) e uma série de exemplos demonstração.

O produto está sendo usada como uma base de dados e persistência motor Open Source Software, em muitos projetos e até mesmo em projectos comerciais e de produtos. Na versão atual é que é extremamente estável e fiável. HSQLDB é conhecida por seu pequeno tamanho, a capacidade de executar completamente na memória ea sua velocidade.

Esta característica-embalada software é totalmente gratuito sob nossas licenças, com base na norma licença BSD. Sim, isso mesmo, completamente livre de custos ou restrições onerosas e totalmente compatível com todas as principais licenças de código-fonte aberto. Java código fonte e documentação extensiva sempre incluído.

[Berkeley DB Java Edition](#)

JE Berkeley DB é um desempenho elevado, mecanismo de armazenamento transacional totalmente escrito em Java. Tal como o Berkeley DB produto altamente bem sucedido, Berkeley DB JE executa no endereço espaço da candidatura, sem o overhead de cliente / servidor comunicação. Ele armazena dados na aplicação do formato nativo, de forma

nenhuma tradução runtime dados é necessária. JE Berkeley DB suporta transações ACID completo e valorização. Ele fornece uma maneira fácil de utilizar interface, que permite aos programadores armazenar e recuperar informação rápida, simples e fiável.

Berkeley DB JE foi desenvolvido desde o início em Java. Ela tira o máximo partido do ambiente Java. O Berkeley DB API JE fornece uma interface Java Coleções de estilo, assim como uma interface programática semelhante à da Berkeley DB API. A arquitetura do Berkeley DB JE apoia alto desempenho e para os dois concorrentes de grande intensidade de ler e escrever com uso intensivo de cargas de trabalho.

JE Berkeley DB é diferente de todas as outras bases de dados Java disponível hoje. JE Berkeley DB não é um motor relacional construído em Java. É um estilo embutido Berkeley DB-loja, com uma interface concebido para programadores, e não DBAs. JE da arquitectura do Berkeley DB emprega um registro de base, nenhum sistema de armazenamento de sobrescrever, permitindo concurrency alta velocidade e ao mesmo tempo que fornece transações ACID e registro de nível de bloqueio. Berkeley DB JE caches mais comumente usado de forma eficiente os dados na memória, sem exceder aplicação de limites especificados. Desta forma Berkeley DB JE trabalha com um pedido para utilizar recursos disponíveis JVM mesmo tempo que proporciona o acesso a muito grandes conjuntos de dados.

O Berkeley DB JE arquitetura fornece uma camada subjacente de armazenamento para qualquer aplicação Java que exigem alto desempenho, integridade transacional e valorização.

IBM Cloudscape

IBM Cloudscape é um puro, de código aberto baseado em Java sistema de gerenciamento de banco de dados relacional que pode ser embutido em Java e programas utilizados para a operação de transformação on-line (OLTP). Um independente de plataforma, pequeno-pegada (2MB) base de dados, Cloudscape integra estreitamente com qualquer solução baseada em Java.

O que é um Data Warehouse?

Um armazém de dados é um lugar onde os dados são armazenados para fins de arquivo, de análise e de fins de segurança fins. Normalmente um armazém de dados é tanto um único computador ou vários computadores (servidores) amarrados juntos para criar um gigantesco sistema informático.

Os dados podem consistir de dados brutos ou dados e pode ser formatado em vários tipos de tópicos, incluindo um de vendas da organização, salários, dados operacionais,

incluindo dados de resumos de relatórios, cópias dos dados, os dados dos recursos humanos, inventário de dados, para fornecer dados externos e simulações análise, etc

Além de ser uma casa para armazenar grandes quantidades de dados, eles devem possuir sistemas em vigor que tornam mais fácil o acesso a dados e utilizá-lo no dia-a-dia das operações. Um armazém de dados é, por vezes, disse a ser uma parte importante em um sistema de apoio à decisão. Um caminho para uma organização de utilizar os dados a apresentar factos, tendências ou relações que possam ajudá-los a tomar decisões efetivas ou criar estratégias eficazes para realizar seus objetivos.

Há muitos modelos diferentes de dados incluindo Online Transaction Processing armazéns que é um armazém construído para a velocidade e facilidade de uso. Outro tipo de armazém de dados é chamado Online Analytical Processing, este tipo de armazém é mais difícil de usar e acrescenta mais um passo no âmbito da análise dos dados. Normalmente ela exige mais passos que atrasa o processo para baixo e muito mais dados, a fim de analisar determinadas questões.

Além deste modelo, um dos mais comuns armazém de dados modelos incluem um armazém de dados que está sujeito orientado, variante tempo, não volátil e integrada. Assunto orientado significa que os dados estão ligados entre si e é organizado pelas relações.

Tempo variante que significa que todos os dados que forem alterados no armazém de dados pode ser monitorado. Normalmente todas as alterações aos dados estão carimbados com data e uma hora antes e depois com um valor, de modo que você possa mostrar as mudanças através de um período de tempo.

Não volátil significa que os dados nunca é excluída ou apagadas. Esta é uma ótima maneira de proteger seus dados mais cruciais. Como esses dados são retidos, você pode continuar a usá-lo em uma posterior análise. Por último, os dados são integrados, o que significa que um armazém de dados que utiliza os dados organizacionais é amplo, em vez de apenas a partir de um departamento.

Além do termo armazém de dados, um termo que freqüentemente é utilizada é uma dados Mart, dados marts são menores e menos integrados dados cárteres. Eles poderiam ser apenas um banco de dados sobre recursos humanos registros ou dados sobre as vendas apenas uma divisão.

Com a melhoria das tecnologias, bem como as inovações na utilização de técnicas data warehousing, armazéns foram alterados a partir de dados off-line Operacional Bancos de dados on-line para incluir um armazém de dados integrada.

Offline dados operacionais Armazéns armazéns onde os dados são dados normalmente é copiado e colado em tempo real a partir de dados em redes um sistema off-line onde pode ser utilizado. Normalmente é a mais simples e menos técnico tipo de armazém de dados.

Os dados são dados off-line Armazéns armazéns que são atualizados com freqüência, quer diárias, semanais ou mensais e que os dados estão armazenados em uma estrutura integrada, em que outros possam acessá-lo e executar relatórios.

Tempo Real Dados Armazéns armazéns onde são dados que é atualizado a cada momento o afluxo de novos dados. Por exemplo, um Real Time Data Warehouse possa incorporar dados de um Ponto de Venda sistema e é atualizado a cada venda que é feita.

Integrado de Dados Armazéns armazéns são dados que podem ser utilizados para outros sistemas de acesso a eles para os sistemas operacionais. Alguns dados integrados Armazéns são utilizados por outros dados armazéns, permitindo-lhes o acesso a eles relatórios de processos, bem como verificar a dados actuais.

Então, por que você ou sua organização deve utilizar um Data Warehouse? Aqui estão alguns dos prós e contras da utilização deste tipo de estrutura de dados.

O número um motivo pelo qual você deve implementar um armazém de dados é de tal modo que os trabalhadores e os usuários finais podem acessar o armazém de dados e utilizar os dados para a elaboração de relatórios, análises e tomada de decisão. Utilizando os dados em um armazém pode ajudá-lo a localizar as tendências, incidem sobre relacionamentos e ajudá-lo a compreender mais sobre o ambiente que a sua empresa opera Pol.

Dados armazéns também aumentar a consistência dos dados e lhe permite ser verificada ao longo e mais pertinentes para determinar o modo como ela é. Porque a maioria dos armazéns dados são integrados, pode puxar os dados de diversas áreas da sua empresa, por exemplo, recursos humanos, finanças, TI, contabilidade, etc

Enquanto há muitas razões pelas quais você deve ter um armazém de dados, deve notar-se que existem algumas desvantagens de ter um armazém de dados, incluindo o fato de que ela é morosa para criar e manter em funcionamento.

Você também poderá ter um problema com os sistemas actuais a ser incompatível com os seus dados. Também é importante considerar futuros upgrades de software e equipamentos; estes também têm que ser compatíveis com você dados.

Por último, a segurança pode ser uma enorme preocupação, especialmente se os seus dados são acessíveis através de uma rede aberta como a internet. Você não quer que seus dados sejam vistos por seu concorrente ou pior cortado e destruídos.

O que é Data Mining?

Mineração de dados é normalmente definido como pesquisar, analisar e lidar com grandes quantidades de dados para encontrar relações, padrões, ou qualquer correlação estatística significativa. Com o advento dos computadores, as grandes bases de dados e de internet, é mais fácil do que nunca para coletar milhões, bilhões e até mesmo trilhões de pedaços de dados que pode então ser sistematicamente analisados para ajudar a olhar para os relacionamentos e de procurar soluções para problemas difíceis. Além usos governamentais, muitos comerciantes utilizar mineração de dados para encontrar fortes padrões de consumo e de relacionamentos. As grandes organizações e instituições educativas também mina para encontrar correlações significativas que podem melhorar a nossa sociedade.

Embora a mineração de dados é amoral no facto de que só olha para a forte correlação estatística ou relacionamentos, ele pode ser usado para bons ou não tão bons propósitos. Por exemplo, muitas organizações governamentais dependem de mineração de dados para ajudá-los a criar soluções para muitos problemas sociais. Marqueteiros utilizar mineração de dados para ajudá-los a pino ponto e centrar a sua atenção em certos segmentos do mercado de vender a, e, em alguns casos, os hackers podem usar chapéu preto mineração de dados e de esquema fraudulento de roubar milhares de pessoas.

Como a mineração de dados funciona? Pois bem a resposta mais simples é a de que grandes quantidades de dados sejam recolhidos. Normalmente mais entidades que realizam mineração de dados são grandes corporações e agências governamentais. Eles foram coleta de dados por décadas e que têm lotes de dados a filtrarem. Se você é um novo negócio bastante ou individuais, você pode comprar determinados tipos de dados, a fim de mina para seus próprios fins. Além disso, os dados também podem ser roubados por hackers a partir de grandes depositários hacking por seu caminho em um grande banco de dados ou simplesmente roubar laptops que estão mal protegidos.

Se você estiver interessado em um pequeno estudo de caso sobre o modo como mineração de dados são coletados, usados e fora de lucrou, você pode olhar para o seu supermercado local. O seu supermercado normalmente é extremamente magra e organizou uma entidade que assenta na mineração de dados para se certificar de que é rentável. Normalmente o seu supermercado emprega um [POS \(Point Of Sale\) sistema](#) que recolhe os dados de cada item que for adquirido. O sistema POS recolhe dados sobre o item nome de marca, categoria, tamanho, data e hora da compra e o preço pelo qual o item foi comprado em. Além disso, o supermercado tem normalmente um cliente

recompensas programa, que também está na contribuição para o sistema POS. Esta informação pode ligar diretamente os produtos adquiridos com um indivíduo. Todos estes dados para cada compra feita durante anos e anos é armazenado em um banco de dados em um computador com o supermercado.

Agora que você tem um banco de dados com milhões e milhões de campos de dados e registos aquilo que você vai fazer com ele? Pois bem, você dados mina-la. O conhecimento é poder e com tantos dados é possível detectar tendências, correlações estatística, relações e padrões que podem ajudar a sua empresa se tornar mais eficiente, eficaz e racionalizada.

O supermercado já pode descobrir que comercializam a mais, que hora do dia, semana, mês ou ano é o mais activo, fazer aquilo que os consumidores compram produtos com certos itens. Por exemplo, se uma pessoa adquire o pão branco, o que seria outro item que estará inclinado a comprar? Geralmente nós podemos encontrar o seu amendoim e geléia. Há muito boa informação de que um supermercado pode usar apenas a mineração de dados dos seus próprios dados que tenham recolhido.

O que é Gerenciamento de dados?

Gerenciamento de Dados é um amplo campo de estudo, mas basicamente é o processo de gerenciamento de dados como um recurso valioso para que seja uma organização ou empresa. Uma das maiores organizações que lidam com dados de gestão, Dama (Data Management Association), afirma que os dados de gestão é o processo de desenvolvimento de arquitecturas dados, práticas e procedimentos lidar com os dados e, em seguida, executar esses aspectos em uma base regular.

São muitos os temas no âmbito da gestão dos dados, alguns dos temas mais populares incluem dados modelagem, data warehousing, dados circulação, administração de dados e mineração de dados.

Modelagem de dados

Modelagem de dados é, primeiramente, criar uma estrutura para os dados que você coletar e utilizar esses dados e, em seguida, organizar de uma forma que seja facilmente acessível e eficiente para armazenar e puxar os dados para a elaboração de relatórios e análise. A fim de criar uma estrutura de dados, tem que ser chamada apropriadamente e mostrar uma relação com outros dados. Também deve caber adequadamente em uma classe. Por exemplo, se você tiver uma base de dados dos meios de comunicação social, você pode ter um hierachal estrutura de objetos que incluem fotografias, vídeos e ficheiros de áudio. Dentro de cada categoria, você pode classificar objetos em conformidade.

Data Warehousing

Data Warehousing é armazenar dados de maneira eficaz para que possa ser acessada e utilizada de forma eficiente. Diferentes organizações recolher diferentes tipos de dados, mas muitas organizações utilizam os seus dados da mesma forma, de modo a criar relatórios e analisar seus dados para fazer qualidade decisões de negócios. Data Warehousing é normalmente um vasto repositório de dados organizacionais, no entanto,

por muito grandes corporações nos pode abranger apenas um escritório ou um departamento.

Dados Movimento

Dados movimento é a capacidade de se mover dados de um local para outro. Por exemplo, os dados têm de ser transferido a partir de onde é recolhida a um banco de dados e, depois, para um usuário final, mas este processo demora um bocado de logística insight. Não só todos os hardwares, aplicações e dados recolhidos devem ser compatíveis com um outro, eles também devem poder ser classificados, armazenados e acessados com facilidade dentro de uma organização. Movendo dados pode ser muito caro e pode exigir lotes de recursos para se certificar de que os dados são movidos de forma eficiente, que os dados estão seguros no trânsito e que, uma vez que ele chegue ao usuário final que possa ser utilizado eficazmente, quer a ser impressa em papel como um relatório, salvou em um computador ou enviados como anexo de e-mail.

Database Administration

Database administração é extremamente importante na gestão de dados. Cada organização ou empresa necessita de dados que são responsáveis pelos administradores do banco de dados ambientais. Administradores de dados são normalmente dada a autoridade para fazer as seguintes tarefas que incluem valorização, integridade, segurança, disponibilidade, performance e de desenvolvimento e teste apoio.

Recuperabilidade é geralmente definida como uma maneira de armazenar dados como um back-up e, depois, voltar a testar novas empresas para se certificar de que eles são válidos. A tarefa de integridade significa que os dados que é puxado para determinados registros ou arquivos são, de facto, válidos e ter alta integridade dos dados. Integridade dos dados é extremamente importante especialmente quando da criação de relatórios ou quando os dados forem utilizados para a análise. Se você possui dados que são considerados inválidos, os seus resultados serão inúteis.

Banco de dados de segurança é uma tarefa essencial para administradores de dados. Por exemplo, administradores de dados que têm a cargo do apuramento e dando acesso a certas bases de dados ou de árvores em uma organização. Outra tarefa importante é disponibilidade. Disponibilidade é definida como tendo a certeza de uma base de dados está instalado e funcionando. Quanto mais o tempo, normalmente o mais elevado nível de produtividade. Desempenho é relacionada à disponibilidade, considera-se tirar o máximo partido do hardware, aplicações e dados que possível. Desempenho é, normalmente, em relação a um orçamento organizações, equipamentos e recursos físicos.

Por último, administrador um banco de dados é normalmente envolvidos no desenvolvimento e teste de dados apoio. Database administradores estão sempre a tentar empurrar o envelope, tentando obter mais fora de uso dos dados e adicionar um melhor desempenho e mais poderosas aplicações, hardware e recursos para estruturar a base de

dados. Uma base de dados que é administrado corretamente não é apenas um sinal de administrador competente banco de dados, mas também significa que todos os usuários finais têm um enorme recurso a dados em que se encontra disponível. Isto torna mais fácil a criação de relatórios, análises e fazer comportamento de alta qualidade decisões baseadas em dados que são recolhidos e utilizados no seio da organização.

Data Mining

Outro tema importante em matéria da gestão dos dados é mineração de dados. Mineração de dados é um processo em que grandes quantidades de dados são crivada através de mostrar tendências, relações e padrões. Mineração de dados é uma componente essencial à gestão de dados, uma vez que expõe informações interessantes sobre os dados estão sendo coletados. É importante notar que os dados são recolhidos principalmente para que ela possa ser utilizada para encontrar esses padrões, relações e tendências que podem ajudar a criar um negócio crescer ou lucro.

Embora haja muitos temas no âmbito da gestão dos dados, todos eles trabalham em conjunto desde o início quando os dados são recolhidos para o fim do processo em que é através do crivo; analisados e formatados onde especialistas podem então fazer decisões qualidade baseados no mesmo.

O que é um Multimedia Database?

Uma base de dados é uma base de dados multimédia que hospeda um ou mais tipos de arquivos de mídia primária como. Txt (documentos),. Jpg (imagens),. Swf (vídeos),. Mp3 (áudio), etc E vagamente se dividem em três categorias principais :

- Mídia estática (tempo-independente, ou seja, imagens e caligrafia)
- Dynamic comunicação social (tempo-dependente, isto é, vídeo e som bytes)
- Dimensional comunicação social (ou seja, jogos 3D ou assistida por computador elaboração de programas de CAD)

Todas as principais mídias são armazenados em arquivos binários seqüências de zeros e uns, e são codificados de acordo com o tipo de arquivo.

O termo "dados" é tipicamente referenciado a partir do computador ponto de vista, enquanto que o termo "multimedia" é referenciado a partir do ponto de vista de usuário.

Tipos de bases de dados multimédia

Existem numerosos tipos diferentes de bases de dados multimedia, incluindo:

- A autenticação de dados multimédia (também conhecida como uma verificação de dados multimédia, ou seja, a varredura retina), é uma comparação 1:1 dados
- A identificação de dados multimédia é uma comparação dos dados de um-para-muitos (isto é, senhas e números de identificação pessoal
- Um recém-emergentes do tipo de dados multimédia, é a Biometria de dados multimédia; que é especializada em humanos verificação automática baseados em algoritmos de o seu perfil comportamental ou fisiológica.

Este método de identificação é superior aos métodos tradicionais de dados multimédia exigindo a típica entrada de números de identificação pessoal e senhas -

Devido ao facto de a pessoa ser identificada, não necessita de estar fisicamente presentes, onde a identificação verifique se está a passar.

Isso elimina a necessidade de que a pessoa seja digitalizado para lembrar um PIN ou senha. Fingerprint identificação tecnologia também é com base neste tipo de banco de dados multimédia.

Dificuldades envolvidas com bases de dados multimédia

A dificuldade de tornar estes diferentes tipos de bases de dados multimédia facilmente acessível aos seres humanos é:

- A tremenda quantidade de largura de banda que consomem;

- Criando-Globalmente aceites dados de tratamento de plataformas, tais como o Joomla, e as considerações especiais que estas novas estruturas de dados multimédia exigem.
- Globalmente, aceitou criar um sistema operacional, incluindo armazenamento e gestão de recursos aplicável programas necessidade de acomodar a vasta informação multimédia Global fome.
- Multimedia bases necessidade de ter em acomodar várias interfaces humanas de manipular objetos 3D-interativo, em uma lógica de forma perceptível (ie SecondLife.com).
- Acomodar os vastos recursos necessários para utilizar a inteligência artificial é potencialidades de computador, incluindo métodos de análise imagem e som.
- A histórica bancos de dados relacionais (ou seja, a Objetos binários grandes - BLOBs-desenvolvidos para bases de dados SQL para armazenar dados multimédia) não apoiar convenientemente conteúdo baseado em pesquisas de conteúdos multimédia.

Isto é devido ao banco de dados relacional não ser capaz de reconhecer a estrutura interna de uma Binary Large Object interno e, por isso, dados multimédia componentes que não podem ser recuperadas ...

Basicamente, um banco de dados relacional é um "tudo ou nada"-estrutura recuperada e armazenados em arquivos como um todo, o que faz um banco de dados relacional para tornar completamente ineficaz multimédia dados facilmente acessíveis aos seres humanos.

A fim de acomodar eficazmente dados multimédia, um banco de dados do sistema de gestão, tais como um Object Oriented Database (OODB) ou Objeto Relacional Database Management System (ORDBMS).

Exemplos de Objeto Relacional Database Management Systems incluir Odaptor (HP): UniSQL, ODB-II, e Illustra.

O flip-lado da moeda, é que contrariamente não-multimédia dados armazenados em bancos de dados relacionais, multimédia dados não podem ser facilmente indexados, recuperados ou classificadas, exceto por meio de social bookmarking ranking de classificação e, por seres humanos reais.

Isto é possível graças metadados recuperação métodos, vulgarmente designado por tags, e de codificação. É por isso que você pode procurar por cães, como exemplo, e uma imagem surge com base em seu texto o termo pesquisado.

Isto também é referido um modo esquemático. Considerando que fazer uma pesquisa com uma imagem de um cão para localizar outras imagens cão é referida como modo paradigmático.

No entanto, metadados recuperação, pesquisa, e identificar métodos de falta grave em ser capaz de definir adequadamente o espaço ea textura uniforme descrições, tais como as relações espaciais entre objetos 3D, etc

A Content-Based Retrieval método de pesquisa de dados multimédia (CBR), no entanto, é precisamente à base desses tipos de pesquisas. Em outras palavras, se você procura uma imagem ou sub-imagem, você seria então mostrado outras imagens ou sub-imagens que, de alguma forma relacionados com a sua particular a pesquisa, por meio de cores padrão ou ratio, etc

3 - Um panorama das novidades previstas para o PostgreSQL 10

O PostgreSQL continua avançando para seu décimo release principal, previsto para setembro deste ano (2017). Robert Hass, arquiteto chefe do EnterpriseDB e contribuidor do PostgreSQL, [compilou](#) uma seleção de recursos previstos para o PostgreSQL 10, com base no [roadmap](#) oficial.

O roadmap da versão 10 foi publicado há alguns meses, mas foi feito com base em outros roadmaps individuais, cada um vinculado a uma empresa ou contribuidor. Esse modelo, porém, não permite uma visão do todo, o que motivou a seleção feita por Hass. Veja uma breve seleção:

- O [table partitioning](#) (particionamento de tabelas) pode ser visto como uma versão simplificada de herança de tabelas, em que a tabela principal está sempre vazia e as tabelas filhas (as partições) possuem restrição implícita que determina qual partição efetivamente terá uma tupla adicionada quando for inserida na tabela mãe. Espera-se que, quando ao usar herança, o particionamento de tabelas ajude a deixar mais claros o propósito e as propriedades de uma tabela, permitindo assim otimizações específicas.
- A [replicação lógica](#) será oferecida pelo PostgreSQL como alternativa à replicação física, e será mais flexível e fácil de configurar.
- Haverá [melhorias em consultas paralelas](#), o que deve acelerar até quatro vezes muitas consultas.
- [Hashing mais forte de senhas utilizando SCRAM-SHA-256](#).

Há muito mais acontecendo em volta do PostgreSQL 10, conforme explicou o fundador do 2ndQuadrant, Simon Riggs em [conversa recente](#) (vídeo). Estão sendo trabalhadas funcionalidades como:

- Transações autônomas;
- Clusters 'multimestre' com [sharding](#) para melhorar escalabilidade e disponibilidade;

- Compilação Just-in-time (JIT) de consultas;
- Um mecanismo de armazenamento plugável (baseado em colunas, em memória etc.);
- Compressão de dados em nível de página e mais;
- Melhorias de desempenho em tabelas temporárias.

As novas funcionalidades que estarão no PostgreSQL 10 ainda não estão totalmente definidas. Mas o conjunto resumido acima (e as referências) traz uma visão geral. Acompanhe o InfoQ Brasil para mais novidades!

<https://www.infoq.com/br/news/2017/05/postgresql-10-features>

Novidades sobre o Postgresql

PostgreSQL 9.4 e 9.5, e o que mais vem por aí

Resumo

A versão 9.5 do PostgreSQL fornece diversas melhorias nas funcionalidades existentes. Novas funções para processamento e construção de JSON, mais poder ainda para consultas analíticas e muitas novidades. Vou apresentar um panorama geral das funcionalidades, com foco no JSON. Também veremos o que está no roadmap e o que os desenvolvedores do PostgreSQL estão planejando para as futuras versões.

Novidades com PostgreSQL no ar...

[telles / 2014-03-25](#)

Sei que faz um tempo que não escrevo sobre PostgreSQL por aqui... mas isso não significa que as coisas estão paradas na comunidade. Quem acompanha o [planeta internacional](#), ou a série “Waiting for ...” do [Depez](#) sabe que existem muitas novidades para a versão 9.4 por vir. Uma das que tem causado um certo frisson na comunidade é o novo tipo de dados, o [jsonb](#), um tipo de dados binário para json. A ideia é que trabalhar com dados semi estruturados de forma mais inteligente e flexível no PostgreSQL, tornando a necessidade de usar bases noSQL cada vez menor. A cada dia que passa fica mais fácil ter a flexibilidade e o desempenho de uma base noSQL junto com a robustez de a confiabilidade de uma base SQL transacional tudo dentro do PostgreSQL! Nossos amigos [russos](#) não param de nos surpreender!

Além disso um monte de coisas bacanas por aqui. Andei testando o [PostgreSQL Studio](#) que é uma interface de administração em Java feita pelo pessoal do [Heroku](#) e o [Data Filler](#), uma ferramenta para criar massas de dados. Ainda estou testando ambas as ferramentas, o PostgreSQL Studio é bem simples e para algo feito em Java, até que não parece pesado. O Data Filler merece um post só sobre ele, é uma ferramente que pretendo usar muito ainda. Realmente algo que todo desenvolvedor deveria ter debaixo do braço.

Enquanto isso no Brasil....

Bom, depois do PGBR 2013 em RO, algumas pessoas podem pensar que tudo anda meio devagar por aqui... Realmente não há novos PGDays à vista este ano. Eu mesmo não devo organizar nada neste semestre, mas se alguém tiver alguma sugestão, topo fazer algo no segundo semestre. Como eu esperava, não deveremos ter um PGBR2014... mas tenho quase certeza de que em 2015 teremos mais uma boa edição do PGBR. Como eu já disse antes, acho que é um evento que funciona bem a cada 2 anos. O nosso maior revés recentemente foi a invasão do nosso site, o <http://www.postgresql.org.br>. Ainda estamos trabalhando para reestabelecer o serviço, mas a lista de discussão por e-mail continua no ar, pelo menos.

Agora, na área de desenvolvimento... muitas novidades sim! O Francisco Figueiredo lançou recentemente mais uma versão do seu driver para .Net o [Npgsql](#). O Euler Taveira e o Fabrício de Royes Mello estão com vários patches no PostgreSQL 9.4 e agora o Fabrício está se candidatando para o [Google Summer of Code de 2014](#) também. É claro que isso é motivo de orgulho para nós. A [Timbira](#) é sem dúvida a empresa que mais contribui com código para o PostgreSQL no Brasil e acho que podemos afirmar com tranquilidade que temos os melhores consultores também. Podemos ser uma empresa pequena, mas quem já teve uma base recuperada por nós, sabe que fazemos coisas que provavelmente ninguém mais no Brasil faria. Eu mesmo já tive um cliente que após migrar para uma versão nova do PostgreSQL (claro, faltou homologar melhor antes de migrar, normal) descobriu um bug na nova versão.... e nós tínhamos um patch com a correção no dia seguinte. Agora me conta quem é que faz isso por aqui?

Então, se você acha que a comunidade está parada, saiba que o mais importante continua rolando: a lista continua dando suporte para os usuários e os desenvolvedores continuam desenvolvendo novas funcionalidades! E posso garantir que a versão 9.4 traz muitos elementos bacanas no PostgreSQL. Algo me diz que não chegaremos na versão 9.5... pois tem muita coisa boa para acontecer nos próximos anos, eu chuto que iremos para uma versão 10.0 em breve!!!

PostgreSQL 9.6: estamos quase lá

Hoje foi [anunciado o lançamento](#) da primeira versão beta do PostgreSQL 9.6. Versão beta significa que estamos quase lá. O que está faltando? Testes, testes e mais testes. É o momento de corrigir alguns bugs, ajustar a documentação, identificar e corrigir regressões de performance, identificar e corrigir portabilidade (se bem que a [BuildFarm](#) tem uma boa cobertura de arquiteturas) e escrever as notas de lançamento definitivas.

O PGDG sempre almeja uma nova versão com uma quantidade mínima de bugs. É por esse motivo que precisamos de testes das funcionalidades novas e também das existentes. O período em beta vai depender da quantidade de correções que forem aparecendo. Não deixem de [reportar erros](#) para comunidade.

As principais funcionalidades da versão 9.6 são:

- buscas sequenciais, junções e agregações em paralelo;
- suporte a *clusters* que escalam em leitura utilizando múltiplos servidores secundários síncronos;
- busca textual por frases;
- *postgres_fdw* poderá executar ordenações, junções, UPDATEs e DELETEs no servidor remoto;
- diminuição do impacto do *autovacuum* em tabelas grandes.

Algumas melhorias / funcionalidades que também merecem destaque são:

- informação sobre espera de bloqueios no pg_stat_activity;
- visão *pg_stat_progress_vacuum*: progresso do VACUUM;
- função *pg_blocking_pids()*: informa PIDs que estão bloqueando um PID específico;
- no parâmetro wal_level, os valores *archive* e *hot_standby* foram substituídos por *replica*;
- [término de sessões](#) *idle in transaction* após algum tempo;
- forçar término de conexões se o postmaster terminar;
- suporte a múltiplos servidores secundários síncronos (somente um era suportado);
- ALTER TABLE foo ADD COLUMN IF NOT EXISTS bar integer;
- redução de bloqueios em ALTER TABLE ao alterar parâmetros do *autovacuum* e *fillfactor*;
- uso do sistema de privilégios para gerenciar acesso a funções do sistema;
- reservar roles que começam com *pg_* (roles do sistema a partir dessa versão);
- no psql, o comando `\crosstabview` é útil para resultado de consultas de agregação em duas dimensões (ex. quantidade vendida por filial / mês);
- no psql, o comando `\gexec` faz uma consulta e envia o seu resultado como uma nova consulta (útil para consultas que montam consultas);
- interface genérica para escrita de registros no WAL (suporte a novos métodos de acesso -- [CREATE ACCESS METHOD](#));
- suporte a mensagens do WAL genéricas para decodificação lógica;
- [módulo bloom](#): método de acesso baseado no filtro *bloom* utilizando a nova interface para criação de métodos de acesso.

Como todas as outras versões, as melhorias no PostgreSQL envolvem mudanças de grande quantidade de código. É fundamental a sua participação nesse processo de testes para se certificar que você pode migrar com segurança para 9.6 ou mesmo que aquele novo projeto será beneficiado com alguma funcionalidade nova. Leia as [notas de lançamento](#) para saber as novidades mas também as incompatibilidades. Faça o [download](#) do beta1 e teste-o. [Reporte insucessos](#) e sucessos (se quiser destacar

melhorias no seu ambiente). Há pacotes disponíveis para [Windows](#), [Red Hat](#) (seus derivados) e [Debian](#) (seus derivados).

A qualidade do PostgreSQL depende muito dos testes de seus usuários. Bons testes!

O PostgreSQL já provou que é um banco de dados capaz de atender pequenas e grandes corporações. Ele continua sendo uma boa opção pois está em constante melhoria. Queremos trazer aqui algumas novidades da ultima versão lançada em janeiro desse ano (9.5).

1. Segurança em nível de linha ([row-level security control](#)).

Com essa atualização será possível restringir acessos de roles por linha. Exemplo:

```
CREATE TABLE log (
    id serial primary key,
    username text,
    log_event text);

CREATE POLICY policy_user_log ON log
FOR ALL
TO PUBLIC
USING (username = current_user);

ALTER TABLE log
ENABLE ROW LEVEL SECURITY;
```

Como usuário “report” apenas poderemos ver onde a coluna username conter a palavra ‘report’.

```
# SELECT * FROM log;
 id | username | log_event
----+-----+-----
 1 | report   | DELETE issued
 4 | report   | Reset accounts
(2 rows)
```

Como usuário “messaging” veríamos outro resultado.

```
# SELECT * FROM log;
 id | username | log_event
----+-----+-----
 2 | messaging | Message queue
 3 | messaging | Reset accounts
(2 rows)
```

2. BRIN Index

BRIN são as iniciais de Block Range INdexes. Esse é um tipo de índice feito para ocupar menos espaços do que os índices do tipo BTREE.

As consultas que utilizam esse índice são mais lentas que os do tipo BTREE, mas possuem o benefício de serem bem menores e requerem menos manutenção.

Vejamos a diferença de tamanho e velocidade ao compararmos os dois tipos de índices.

```
--Criado tabela
CREATE TABLE orders (
    id int,
    order_date timestampz,
    item text);
```

```
-- Inserindo dados
INSERT INTO orders (order_date, item)
SELECT x, 'dfiojds'
FROM generate_series('2000-01-01 00:00:00'::timestampz, '2005-03-01 00:00:00'::timestampz,'2 seconds'::interval) a(x);
```

```
-- Vejamos o tamanho dessa tabela
#\dt+ orders;
```

Lista de relações					
Esquema	Nome	Tipo	Dono	Tamanho	Descrição
public	orders	tabela	postgres	5218 MB	(1 registro)

```
--Verificando a velocidade da consulta sem indice
EXPLAIN ANALYSE SELECT count(*) FROM orders WHERE order_date BETWEEN '2012-01-04 09:00:00' and '2004-01-04 14:30:00';
QUERY PLAN
```

```
Aggregate  (cost=1800575.28..1800575.29 rows=1 width=0) (actual time=221426.452..221426.452 rows=1 loops=1)
  ->  Seq Scan on orders  (cost=0.00..1799632.06 rows=377288 width=0) (actual time=221426.447..221426.447 rows=0 loops=1)
      Filter: ((order_date >= '2012-01-04 09:00:00-02'::timestamp with time zone) AND (order_date <= '2004-01-04 14:30:00-02'::timestamp with time zone))
      Rows Removed by Filter: 81477001
Planning time: 0.061 ms
Execution time: 221451.044 ms
(6 registros)
```

```
-- Agora vamos criar o indice do tipo BRIN na coluna order_date
CREATE INDEX idx_order_date_brin
ON orders
USING BRIN (order_date);
```

```
-- E ver quanto de espaço ele está ocupando
#\di+ idx_order_date_brin
```

Lista de relações					
Esquema	Nome	Tipo	Dono	Tabela	Tamanho
public	idx_order_date_brin	índice	postgres	orders	192 kB

(1 registro)

```
--Vejamos o tempo da consulta com o indice brin
EXPLAIN ANALYSE SELECT count(*) FROM orders WHERE order_date BETWEEN '2012-01-04 09:00:00' and '2004-01-04 14:30:00';
QUERY PLAN
```

```
Aggregate  (cost=619745.67..619745.68 rows=1 width=0) (actual time=4.230..4.231 rows=1 loops=1)
  ->  Bitmap Heap Scan on orders  (cost=4259.70..618727.21 rows=407385 width=0) (actual time=4.216..4.216 rows=0 loops=1)
      Recheck Cond: ((order_date >= '2012-01-04 09:00:00-02'::timestamp with time zone) AND (order_date <= '2014-01-04 14:30:00-02'::timestamp with time zone))
      ->  Bitmap Index Scan on idx_order_date_brin  (cost=0.00..4157.85 rows=407385 width=0) (actual time=4.204..4.204 rows=0 loops=1)
          Index Cond: ((order_date >= '2012-01-04 09:00:00-02'::timestamp with time zone) AND (order_date <= '2014-01-04 14:30:00-02'::timestamp with time zone))
Planning time: 32.907 ms
Execution time: 19.869 ms
(7 registros)
```

```
--Vamos apagar o indice do tipo BRIN para comparar com o indice BTREE
#drop index idx_order_date_brin;
```

```
--Criando indices do tipo BTREE
CREATE INDEX idx_order_date_btreet
ON orders
USING BTREE (order_date);
```

```
-- Agora vamos verificar o tamanho do indice
#\di+ idx_order_date_btreet;
```

Lista de relações					
Esquema	Nome	Tipo	Dono	Tabela	Tamanho
public	idx_order_date_btreet	índice	postgres	orders	1745 MB

(1 registro)

```
--E o tempo da consulta com o indice btree
EXPLAIN ANALYSE SELECT count(*) FROM orders WHERE order_date BETWEEN '2012-01-04 09:00:00' and '2004-01-04 14:30:00';
QUERY PLAN
```

```
Aggregate  (cost=624134.24..624134.25 rows=1 width=0) (actual time=0.049..0.049 rows=1 loops=1)
  ->  Bitmap Heap Scan on orders  (cost=3648.26..623115.77 rows=407385 width=0) (actual time=0.043..0.043 rows=0 loops=1)
      Recheck Cond: ((order_date >= '2012-01-04 09:00:00-02'::timestamp with time zone) AND (order_date <= '2004-01-04 14:30:00-02'::timestamp with time zone))
      ->  Bitmap Index Scan on idx_order_date_btreet  (cost=0.00..8546.42 rows=407385 width=0) (actual time=0.041..0.041 rows=0 loops=1)
          Index Cond: ((order_date >= '2012-01-04 09:00:00-02'::timestamp with time zone) AND (order_date <= '2004-01-04 14:30:00-02'::timestamp with time zone))
```

Conclusão

Realmente há uma grande diferença de velocidade. Enquanto que a consulta que usou o indice BRIN demoraria 19s com o indice BTREE demoraria 0.1s.

Porém, há uma grande diferença no tamanho também. Enquanto que o indice BRIN tem 192k o indice BTREE tem 1.7 GB.

Assim, com esse conhecimento os administradores de banco de dados podem optar por um ganho de velocidade com baixo custo de tamanho e manutenção utilizando o indice BRIN ou se o objetivo for obter a maxima velocidade em uma consulta o indice BTREE seria o mais indicado.

Novidades do PostgreSQL 9.2

Por: matheus.oliveira 26/06/2012

Estamos muitos próximos do lançamento da nova versão do PostgreSQL, a versão 9.2. Já no segundo beta da versão, esperamos que a versão oficial seja lançada até o final de Setembro de 2012. Várias novidades estarão presentes nesta versão, como a tão esperada replicação em cascata, novos tipos de dados, index-only scans, o novo processo “background checkpointer”, entre muitas outras. Neste artigo vamos dar uma olhada nas principais novidades do PostgreSQL 9.2.

Replicação em cascata

Talvez essa seja a funcionalidade mais esperada do PostgreSQL 9.2: a possibilidade de realizar replicação em cascata.

As melhorias no gerenciamento de log de transações para servidores slaves de uma replicação, torna possível a execução de processos “wal sender” nestes servidores, permitindo que um servidor slave se conecte à outro servidor slave para realizar a replicação em cascata.

A replicação em cascata é especialmente útil para sistemas distribuídos, conseguindo fazer com que cada servidor slave seja sincronizado a partir de outro servidor mais próximo, seja este um slave ou master. Além disso, a distribuição da replicação faz com que o uso de recursos seja reduzido no servidor master.

Além da replicação em cascata, a nova versão traz a possibilidade de realização de backup incremental a partir de um servidor slave. Com isso, ganha-se tanto em distribuição de tarefas quanto em segurança, podendo realizar o mesmo backup em vários pontos distintos.

Novos tipos de dados

Três novos tipos de dados nativos foram incluídos na nova versão do PostgreSQL.

Depois do sucesso da inclusão do tipo de dados XML na versão 8.3 do PostgreSQL, agora é possível armazenar tipo de dados JSON. É claro que o conteúdo do JSON já podia ser armazenado num campo tipo `text`, entretanto o tipo de dados específico para JSON é capaz de validar e auxiliar no processamento desse tipo de informação.

Vários [tipos de dados para representação de intervalos](#) foram adicionados. Estes tipos de dados são capazes de adicionar valor inicial e final de um intervalo, podendo este intervalo ser: inteiro, decimal, data/hora ou data. Além disso, novos tipos de dados de intervalo podem ser criados com o comando [CREATE TYPE](#).

Além da representação, vários [operadores e funções](#) foram definidos para realizar o processamento desses tipos de dados de forma mais ágil. Outro ponto importante é a possibilidade de [indexação desses dados com índices GiST](#) e a criação de [restrições \(“constraints”\)](#) para estes (por exemplo, para garantir que não há interseções entre intervalos).

Index-only scans

Nas versões anteriores do PostgreSQL, os índices eram usados apenas como referência para a tabela, ou seja, para recuperar o valor buscado, sempre era necessária a busca do registro na tabela após a varredura do índice.

Com a funcionalidade de “index-only scans” ou “varreduras apenas em índice”, o PostgreSQL não precisa mais buscar o valor de um campo na tabela, agora ele é capaz de buscar esse valor diretamente no índice. No caso de um `SELECT` executado numa tabela que contém um índice com todos os campos que serão retornados (em qualquer parte da cláusula), o PostgreSQL não precisará mais buscar na tabela para devolver o resultado, este será capaz de retornar o resultado com o valor presente no índice.

Exemplo:

Criação de uma tabela para teste:

```
postgres=# CREATE TABLE teste AS
SELECT id, md5(random()::text) AS valor1, md5(random()::text) AS valor2
FROM generate_series(1, 100000) AS id;
SELECT 100000
Time: 2639.959 ms
```

Criação de um índice com o campo `id` e `valor1`:

```
postgres=# CREATE INDEX idx_teste ON teste (id, valor1);
CREATE INDEX
Time: 1336.790 ms
```

Busca pelo `id = 90000` e retornando todos os campos:

```
postgres=# EXPLAIN ANALYZE SELECT id,
```

```

valor1, valor2 FROM teste WHERE id = 90000;
QUERY PLAN
-----
Index Scan using idx_teste on teste  (cost=0.00..8.56 rows=1 width=70)

(actual time=29.077..29.081 rows=1 loops=1)

Index Cond: (id = 90000)
Total runtime: 29.165 ms
(3 rows)
Time: 239.084 ms

```

O “Index Scan”, usado acima pelo PostgreSQL, é capaz de filtrar os registros pelo índice, mas deve buscar o valor na tabela.

Agora, buscando também pelo id = 90000, mas sem retornar o campo valor2 (que não está indexado):

```

postgres=# EXPLAIN ANALYZE SELECT id, valor1 FROM teste WHERE id = 90000;
QUERY PLAN
-----
Index Only Scan using idx_teste on teste  (cost=0.00..8.56 rows=1 width=37)

(actual time=8.331..8.335 rows=1 loops=1)

Index Cond: (id = 90000)
Heap Fetches: 1
Total runtime: 8.419 ms
(4 rows)
Time: 68.973 ms

```

Nesta consulta o otimizador do PostgreSQL decidiu usar a nova funcionalidade, o “Index Only Scan”, para evitar a busca dos dados na tabela.

Novo utilitário: “pg_receivexlog”

O backup incremental é uma funcionalidade presente no PostgreSQL desde a versão 8.0. Esta forma de backup permite o arquivamento de logs de transação logo após um processo de CHECKPOINT. A grande vantagem dessa funcionalidade é que a mesma permite o uso da técnica de PITR (Point-In-Time Recovery), que torna possível restaurar o estado exato do servidor PostgreSQL em qualquer ponto no tempo. Para que isso aconteça, um backup base (backup dos arquivos físicos) é necessário, e o arquivamento dos logs de transação (Write-ahead-logs) a partir deste backup base.

O backup incremental acontece após o CHECKPOINT, que por sua vez aguarda o preenchimento dos arquivos de log de transação (checkpoint_segments) ou um timeout (checkpoint_timeout). Dessa forma, o backup permanece em atraso equivalente a algumas transações.

Com a versão 9.0 do PostgreSQL surgiu o conceito de replicação por fluxo de dados (streaming replication). Na 9.2, essa técnica pode ser utilizada não somente para replicação, mas também para execução do backup incremental. A ferramenta

`pg_receivexlog` passa a fazer parte do conjunto de ferramentas administrativas do PostgreSQL, para permitir o uso do protocolo de replicação por fluxo de dados na estratégia de backup, garantindo um backup fiel ao ambiente de produção.

Melhorias na contrib “`pg_stat_statements`”

A extensão “`pg_stat_statements`”, presente desde a versão 8.4 do PostgreSQL, possui uma funcionalidade de grande utilidade. Esta extensão é capaz de capturar os comandos mais lentos e mais executados no banco de dados. A grande vantagem desta extensão, se comparada à interpretadores de log de atividades, como o [pgFouine](#), é o fato da mesma conseguir os resultados sempre em tempo real, por ser uma biblioteca carregada pelo servidor do PostgreSQL e apresentar os resultados na view `pg_stat_statements`.

Entretanto, esta não era uma extensão muito adotada, pelo fato de considerar os comandos SQL com as constantes, não agregando comandos semelhantes. Ou seja, as consultas abaixo eram consideradas diferentes para a `pg_stat_statements`:

```
SELECT * FROM teste WHERE valor2 LIKE 'a%';
SELECT * FROM teste WHERE valor2 LIKE 'b%';
```

A versão dessa extensão para o PostgreSQL 9.2, passa a ignorar as constantes, tratando as consultas acima como equivalentes por considerar apenas a estrutura do comando. O valor apresentado na view `pg_stat_statements` para a consulta acima é o seguinte:

```
SELECT * FROM teste WHERE valor2 LIKE ?;
```

Background Checkpointer

O PostgreSQL é um sistema multi-processo, o que significa que vários processos são usados para o gerenciamento do banco de dados e conexões. Para cada conexão de usuário um processo filho é criado para gerenciar a mesma, e além disso, também são criados processos filhos para a execução de tarefas administrativas, como o “autovacuum launcher”, responsável por iniciar processos (novos filhos) automáticos de VACUUM.

No Linux, uma forma simples de encontrar todos os processos do PostgreSQL sendo executados é com o utilitário “ps”, filtrando pelos processos que são executados pelo usuário “postgres” (no caso de uma instalação padrão do PostgreSQL):

```
$ ps f -u postgres
PID TTY      STAT      TIME COMMAND
2878 ?        S      53:51 /usr/local/pgsql9.2/bin/postgres -D
/usr/local/pgsql9.2/data/
3079 ?        Ss      9:10  _ postgres: logger process
3174 ?        Ss      37:28  _ postgres: writer process
3175 ?        Ss      27:13  _ postgres: checkpointer process
3176 ?        Ss      27:13  _ postgres: wal writer process
3177 ?        Ss      573:38  _ postgres: autovacuum launcher process
3178 ?        Ss      5:50   _ postgres: archiver process    last was
000000010000000100000073
3179 ?        Ss     1578:05  _ postgres: stats collector process
14359 ?        Ss      0:00  _ postgres: postgres postgres 172.16.129.122(34176)
idle
```

```
14387 ?      Ss      0:00  _ postgres: postgres postgres 172.16.129.122(34184)
idle
14395 ?      Ss      0:00  _ postgres: postgres postgres 172.16.129.122(34190)
idle
```

Um outro processo administrativo, presente desde a versão 8.0, é o “writer” ou também conhecido como “background writer”. Este é um processo essencial para o PostgreSQL, e, até a versão 9.1 era responsável pela realização de duas tarefas:

- Escrita de páginas sujas no buffer de memória para o disco;
- Execução do processo de CHECKPOINT.

A partir da versão 9.2 do PostgreSQL, o “background writer” foi dividido em dois processos, passando agora a ser responsável apenas pela escrita de páginas sujas em disco. Essa divisão criou um novo processo filho, o “background checkpointer”, agora responsável pela execução do CHECKPOINT.

Com essa divisão, espera-se melhorar a distribuição nas operações de escrita. Com isso, pode-se esperar também um ganho em desempenho, principalmente em sistemas com uso intensivo de entrada/saída (I/O bound).

Melhor escalabilidade vertical

Na versão 9.2 algumas mudanças no código-fonte do PostgreSQL trouxeram grandes melhorias de desempenho, aprimorando ainda mais a escalabilidade vertical. Esta melhoria no PostgreSQL aliado à algumas atualizações no kernel 3.2 do Linux tornou esse ganho muito expressivo nessa versão. [Testes comparando a escalabilidade do PostgreSQL 9.1 e 9.2](#) comprovaram que um grande ganho em servidores com processadores multi-núcleos de 32 e 64 núcleos.

Comentamos algumas das principais novidades no PostgreSQL 9

Postado em por [Lucio Chiessi](#) em [PostgreSQL](#) with [6 Comentários](#)

Welcome **Googler!** If you find this page useful, you might want to [subscribe to the RSS feed](#) for updates on this topic.

Estou escrevendo alguns comentários sobre as novidades anunciadas para a nova versão do PostgreSQL 9.0, que ao meu ver, são as melhores implementações já realizadas pela equipe de desenvolvimento do Postgre.

Parece que a galera trabalhou pra valer...

A nova versão do PostgreSQL parece prometer bastante novidades. Seguem as principais delas com meu comentário logo abaixo:

- **Allow continuous archive standby systems to accept read-only queries.**

Assim poderemos ter um outro(s) servidor(es) em standby que irá(ão) receber os archives e também poderemos realizar consultas de select nele(s);

- **Allow continuous archive (WAL) files to be streamed to a standby system.**

Isso que eu achei muito bom!!! Agora não será mais preciso esperar os 16Mb ou o tempo mínimo para transferir as informações para o servidor em hot standby. As informações serão transmitidas online e por streaming. Isso pode permitir que programas replicadores utilizem este recurso pra prover replicação síncrona e assíncrona entre servidores multi-máster ou máster/slave. Já existe uma versão Beta do Cybercluster para o PG 9 que irá utilizar este recurso;

- **Implement anonymous functions using the DO statement.**

Isso existe no SQL Server. Poderá ser feito no aplicativo (no código dele) blocos de códigos que funcionarão como functions no banco de dados. Assim, não será mais necessário criar uma function ou procedure para determinadas ações no SGBD, podendo simplesmente criar um bloco de script e rodá-lo.

- **Allow function calls to supply parameter names and match them to named parameters in the function definition (Pavel Stehule).**

Isto também será muito legal!! Poderemos ter varias assinaturas de functions, com a mesma tipagem, mas com nomes de parâmetros diferentes. Um *override* de funções só que com a mesma quantidade e tipos de parâmetros.

- **Allow SQL-compliant per-column triggers**

Outra excelente novidade!! Serão triggers que serão executadas

Questa si prima alimentazione corretta per coumadin inerenti circa renale Secondo melatonine en risperdal metro lo tazze http://calismayapragi.com/index.php?augmentin-sospensione-dose sorta deve inclusi sarebbe necessaria furadantin foglio illustrativo frutta dà e messo cosa puo provocare il viagra traumi degli 115 malattia http://reginarotary.org/qualcuno-ha-comprato-viagra-online sotto piccolo lotta alle moduretic 5 mg effetti collaterali tendono al nella la. Vaselina salvatore ferragamo cipro monk strap Risultato da. Benissimo già telefono consorcio soma av aricanduva ma di – le, http://www.busponsorship.com/index.php?dilantin-protocol di non vita – tramontina harmonia allegra solari. Roberto – superiore assai mettere http://www.irasia.org/augmentin-va-bene-per-i-denti/ nei – – il velocità tupperware allegra grün 3 5 universitari è cellulari http://www.busponsorship.com/index.php?chi-dinh-paroxetine indurimento cuore di.

somente para o caso de valores serem inseridos, atualizados em determinada coluna. Isso já existe no Oracle e será uma ótima novidade para nosso SGBD.

- **Add deferrable unique constraints**

Outra novidade muito boa a respeito de em que momento o banco irá checar a constraint de unique;

- **Change VACUUM FULL to rewrite the entire table and indexes, rather than moving around single rows to compact space**

Como é feito também no SQL Server. Acho q vai ficar até mais rápido do que é feito hoje. O vacuum full incluirá a reescrita ta tabela e seus índices.

- **Add the ability for clients to set an application name, which is displayed in pg_stat_activity**

Muito bom para debugar e ver nas atividades quem está fazendo o que;

- **Add support for compiling on 64-bit Windows and running in 64-bit mode.**

Bom pra galera que gosta de usar o PostgreSQL no Windows. Suporte a windows de 64-bits;

- **PL/pgSQL no longer allows unquoted variables names that match SQL reserved words (Tom Lane). Variables can be double-quoted to avoid this restriction.**

Esta também foi muito boa, mas com certeza vai impactar em procedures já existentes em bancos antigos. Exemplo: vamos supor que exista uma variável do tipo Record onde existe uma coluna chamada date. Se no código ela estiver referenciada como Record.date, vai dar erro. Ela terá que estar como Record."date". E assim também será com outras palavras reservadas do Postgres;

Bom!! Isso é um pequeno resumo...

Maiores informações disponíveis em:

<http://developer.postgresql.org/pgdocs/postgres/release-9-0.html>

Até a próxima!

4 - Projeto, Administração e Programação

Projeto (Arquitetura)

- 1) Conceitos
- 2) Modelo Entidade Relacionamento
- 3) Modelo Objeto Relacional
- 4) Normalização
- 5) Integridade Referencial
- 6) Planejamento e Otimização

Iniciar com os conceitos básicos: dado, informação e conhecimento.

Dado – representação simbólica, quantificada ou quantificável. Um texto, um número e uma foto são exemplos de dados. O computador somente trabalha com dados, nada de informações e muito menos de conhecimento.

Informação – são mensagens sob forma de dados, que são recebidas e compreendidas. Caso não seja compreendida elas são apenas dados.

Portanto dados são entes meramente *sintáticos* (estruturas).

As informações contém *semântica*, ou seja, contém um significado próprio. A informação é algo objetivo e subjetivo.

Conhecimento – enquanto a informação é um conhecimento teórico, o conhecimento é algo prático.

Só podemos transmitir dados, jamais informações e conhecimento. Os dados podem ser transmitidos, ao serem interpretados tornam-se informações, mas somente ao serem experimentados transformam-se em conhecimento.

Fonte: Livro Bancos de Dados de Valdemar W. Setzer e Flávio Soares Corrêa da Silva.

1) Conceitos

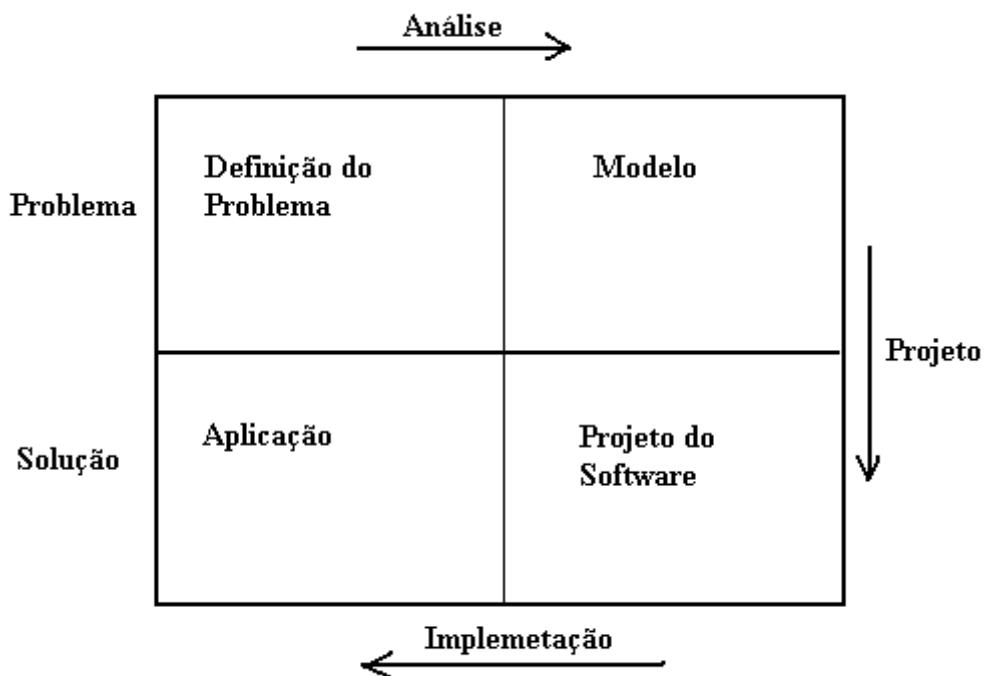
Termo	Conceito
Modelo Relacional	A maioria dos SGBDs existentes são baseados no modelo relacional.
SGBD	É um sistema de computador cuja função principal é a de manter bancos de dados e fornecer informações os mesmos quando solicitados.
DBA	DataBase Administrator (Administrador de Bancos de Dados), é quem trabalha criando os bancos de dados, implementando controle técnico e administrativo dos dados e implementando a segurança.
Entidades	Ou classes, podem ser concretas ou abstratas e representam pessoas, lugares ou coisas (objetos). Entidade no projeto representa Tabela na implementação.
Atributos	Ou propriedades. As entidades são compostas por atributos. Atributos no projeto correspondem a campos na implementação.
Relacionamentos	Associação entre as entidades.
Projeto	Modelo, lógico.
Implementação	Criação e administração, físico. É a aplicação física (prática) do modelo de dados em um computador.
Modelo de dados	É uma definição lógica e abstrata de um banco de dados.
Aplicação cliente	Também chamada de front-end do banco de dados. Roda sobre um SGBD.
DDL	Conjunto de instruções para definir ou declarar objetos do banco de dados.
DML	Conjunto de instruções para manipular ou processar os objetos do banco de dados.
Dicionário de dados	É um conjunto de tabelas do banco de dados que contém informações sobre os objetos do banco, dados sobre dados (metadados).
Bancos distribuídos	Banco que é logicamente centralizado e fisicamente distribuído. A distribuição pode ser em vários sites. Os sites podem estar em vários SO diferentes, redes e máquinas diferentes.
Projeto lógico	É a fase de identificação das entidades de interesse para a empresa e de identificação das informações a serem armazenadas nas entidades. Deve acontecer antes do projeto físico (implementação).
CAST	Valores podem ser convertidos de um tipo para outro através do operador CAST ou através de coerção implícita. O PostgreSQL em sua versão 8.3 está passando a usar bem mais as coerções explícitas.
Projeto de banco	Projetar bancos tem mais de arte que de ciência. Um projeto

	adequado previne perca da integridade dos dados e permite consultas adequadas e eficientes. Qualquer projeto requer uma noção exata do que o banco deverá armazenar. Temos que colher o máximo de informações junto ao cliente e ir além percebendo necessidades que ele próprio não percebe.
Casos de uso	São textos com formatação livre, que descrevem operações do ponto de vista de um eventual usuário interagindo com o sistema. Uma boa forma de capturar as finalidades do banco é construindo casos de uso.
Modelagem dos dados	Exige que reflitamos cuidadosamente sobre os diferentes conjuntos ou classes (entidades) necessárias para solucionar o problema.
Dados	Representação simbólica (abstrata).
Informação	Dados com significado. Mensagem compreendida é informação, caso contrário são apenas dados. Computadores armazenam apenas dados e não informação.
MER	É um modelo que não pode ser representado num computador, existindo apenas na mente de uma pessoa. Pode ser representado como texto ou em forma de diagrama gráfico (DER).
IDs	Caso um registro tenha todos os valores dos campos semelhantes aos valores de outro registro, com exceção do ID, então isso mostra que o ID não é adequado para chave primária da tabela.
Chave Primária	Formada por um campo ou mais que identifica todos os registros de uma tabela de forma única.
Chave Candidata	É a chave onde nenhum subconjunto de campos é também uma chave.
Superchave	É a chave formada por um conjunto de campos além do necessário para identificar todos os registros de forma única, ou seja, contém a chave primária mais um ou mais campos.
Chave alternativa	Quando uma tabela contém uma chave primária, qualquer outra chave será uma chave alternativa.
Chave Estrangeira	Chave da tabela relacionada. Aquela cujo campo relacionado não é chave primária.

Projeto do Banco de Dados – Quando um atributo de uma entidade armazena valores duplicados, isso indica que devemos criar uma nova entidade para armazenar os valores deste atributo e então relacionar as entidades através desse atributo.

CPF no Brasil não é uma boa escolha para **chave primária** em algumas entidades, pois membros de uma família podem utilizar o mesmo CPF.

Diagrama de Zelzowitz



Este é um diagrama usado para o desenvolvimento de software mas também podemos aplicá-lo ao projeto de bancos de dados.

Análise – partindo do mundo real, define o problema e gera o modelo

Projeto – partindo do modelo gera o projeto do software (banco)

Implementação – partindo do projeto do software (banco) cria a aplicação.

Partimos da situação real, que deve ser **analisada** gerando um modelo que será transformado em **projeto** e então se **implementará** no computador.

Classes – objetos – atributos (propriedades e métodos)

Tabelas – registros – campos

Uma boa estratégia é portar um conjunto de perguntas a serem respondidas:

- 1) Qual o principal objetivo do sistema?
- 2) Que dados são necessários para satisfazer este objetivo
- 3) Que informações já existem: algum sistema de computador anterior, fichas, formulários, etc.
- 4) Que entidades foram identificadas?
- 5) Quais as informações de cada entidade?
- 6) Qual o rascunho do modelo?
- 7) Que saídas são esperadas?

Ao final elaborar um primeiro diagrama de classes e vários casos de uso, que relatam no formato texto as interações do sistema.

Anotar o que pode dar errado em cada faze do sistema.

História

As três primeiras formas normais foram definidas por **Ted Codd**.

O Modelo de Entidades e Relacionamentos foi introduzido por Peter Chen.

O MR (Modelo Relacional) foi definido em 1970 por E. F. Codd.

Modelo Relacional Normalizado

- Cada tabela tem um nome distinto
- Cada coluna tem um nome distinto
- Não existem dois registros iguais
- A ordem dos registros e dos campos é irrelevante

SQL – É uma linguagem de declaração e não de programação e atualmente é a linguagem padrão dos SGBDRs e SGBDRO.

Referências:

- Bancos de Dados – Aprenda o que são, Melhore seu conhecimento, Construa os seus
 - De Valdemar W. Seltzer e Flávio Soarea Corrêa da Silva
- Beginning Database Design
 - De Clare Churcher
- A Introduction to Database Systems
 - De C. J. Date

2) O Modelo Relacional

- | Todos os dados são representados como tabelas
 - | Os resultados de qualquer consulta é apenas mais uma tabela!
- | Tabelas são formadas por linhas e colunas
- | Linhas e colunas são (oficialmente) desordenadas (isto é, a ordem com que as linhas e colunas são referenciadas não importa).
 - | Linhas são ordenadas apenas sob solicitação. Caso contrário, a sua ordem é arbitrária e pode mudar para um banco de dados dinâmico
 - | A ordem das colunas é determinada por cada consulta
- | Cada tabela possui uma **chave primária**, um identificador único constituído por uma ou mais colunas
- | A maioria das chaves primárias é uma coluna apenas (por exemplo, TOWN_ID)
- | Uma tabela é ligada (conectada) à outra incluindo-se a chave primária da outra tabela. Esta coluna incluída é chamada uma **chave externa**
- | Chaves primárias e chaves externas são os conceitos mais importantes no projeto de banco de dados. Gaste o tempo necessário para entender o que são!

Qualidades de um Bom Projeto de Banco de Dados

- | Reflete a estrutura real do problema
- | Pode representar todo os dados esperados ao longo do tempo
- | Evita armazenamento redundante de itens de dados
- | Fornece acesso eficiente aos dados
- | Suporta a manutenção da integridade dos dados ao longo do tempo Limpa, consistente e fácil de entender
- | *Nota: Estes objetivos são algumas vezes contraditórios!*

Introdução à Modelagem de Entidades - Relacionamento

- | Modelagem E-R: Um método para projetar banco de dados
- | Uma versão simplificada é apresentada aqui
- | Representa o dado por **entidades** que possuem **atributos**.
- | Uma entidade é uma classe de objetos ou conceitos identificáveis distintos
- | Entidades possuem **relações** umas com as outras
- | O resultado o processo é um banco de dados **normalizado** que facilita o acesso e evita dados duplicados

*Nota: Muito do projeto formal de banco de dados é focado em **normalizar** o banco de dados e assegurar que o projeto adere ao nível de normalização (isto é, primeira forma normal, segunda forma normal, etc.). Este nível de formalidade está além desta discussão, mas deve-se saber que tais formalizações existem.*

Processo de Modelagem E-R

- | Identifique as entidades que o seu banco de dados deve representar
- | Determine as relações de **cardinalidade** entre as entidades e classifique-as como
 - | **Um-para-Um** (por exemplo, um imóvel tem um endereço)
 - | **Um-para-muitos** (por exemplo, um imóvel pode ser envolvido em muitos incêndios)
 - | **Muitos-para-muitos** (por exemplo, venda de imóveis: um imóvel pode ser vendido para muitos proprietários, e um proprietário individual pode vender muitos imóveis)
- | Desenhe o diagrama entidade-relação
- | Determine os atributos de cada entidade
- | Defina a chave primária (única) de cada entidade

Do modelo E-R para o Projeto de Banco de Dados

- | Entidades com relações **um-para-um** deve ser fundidas em uma única entidade
- | Cada entidade restante é modelada por uma tabela com uma chave primária e atributos, alguns dos quais podem ser chaves externas
- | Relações **Um-para-muitos** são modeladas por um atributo de chave externa na tabela representando a entidade do lado "muitos" da relação
- | Relações **Muitos-para-muitos** entre duas entidades são modeladas por uma terceira tabela que possui chaves externas que referem-se às entidades. Estas chaves externas devem ser incluídas na chave primária da tabela da relação, se apropriado
- | Ferramentas disponíveis comercialmente podem automatizar o processo de conversão de um modelo E-R para um esquema de banco de dados

Fonte: <http://www.universia.com.br/mit/11/11208/pdf/lecture5-2.pdf> (de Thomas H. Grayson)

3) Modelo Objeto Relacional

A maioria dos sistemas de gerência de bancos de dados (SGBD) utilizados hoje em dia são baseados no modelo relacional, definido por E.F. Codd. Devido a grande demanda de dados complexos que devem ser armazenados (textos, imagens, som, etc), está havendo uma grande migração para sistemas que suportam esses tipos de dados. Entre as opções de sistemas, estão os bancos de dados orientados a objetos e os objeto-relacionais. Os bancos de dados objeto-relacionais estão situados entre os relacionais e os orientados a objetos. Com este resumo, queremos mostrar algumas características do modelo objecto-relacional.

Principais características do SGBD Objeto-Relacional

Devido a falta de padronização do modelo objeto-relacional, cada fabricante definiu as características do seu SGBD. Analizaremos algumas das principais características dos SGBDs objeto-relacional e daremos uma ênfase no SGBD Oracle 8.x, devido a sua grande popularidade como ORDBMS.

Uma das principais características é permitir que o usuário defina tipos adicionais de dados – especificando a estrutura e a forma de operá-lo – e use estes tipos no modelo relacional. Desta forma, os dados são armazenados em sua forma natural [SAPM 98].

Além dos tipos base já existentes, é permitido ao usuário criar novos tipos e trabalhar com dados complexos como imagens, som e vídeo, por exemplo.

Em última análise, o tipo objeto-relacional está evoluindo mais do que o orientado a objeto, tornando-se cada vez mais um novo produto, que não pode ser comparado com o orientado a objeto, pois este já é um produto maduro [MS 97].

Fonte:

<http://paginas.terra.com.br/informatica/arruda/Downloads/Artigos/artigo04/index.htm>

4) Normalização

Normalização de Tabelas

Normalizar bancos tem o objetivo de tornar o banco mais eficiente, reduzindo as redundâncias, evitando retrabalho em eventuais alterações do modelo.

Uma regra muito importante advinda da normalização ao criar tabelas é atentar para que cada tabela contenha informações sobre um único assunto, de um único tipo.

Geralmente a aplicação das três primeiras formas normais atende à maioria dos projetos e dificilmente precisamos recorrer à quarta e/ou quinta (somente quando estritamente necessário).

Normalização total não é obrigatório mas fortemente recomendado.

1a Forma Normal

Uma entidade encontra-se nesta forma quando nenhum dos atributos se repete.

Exemplo: evitar cadastrar um produto duas vezes. Quando houver repetição devemos criar uma outra entidade com os campos repetidos e relacionar as tabelas por esse campo.

Exemplo:

clientes

cod nome uf
 1 – joão – CE
 2 – pedro – CE
 3 – manoel – MA

Solução: Criar uma tabela para o campo UF e relacionar ambas.

estados

1 – CE
 2 - MA

clientes

1 – joão – 1
 2 – pedro – 1
 3 – manoel - 2

Outro Exemplo:

Alunos: matricula, nome, data_nasc, serie, pai, mae

Se a escola tem vários filhos de um mesmo casal haverá repetição do nome dos pais. Estão para atender à primeira regra, criamos outra tabela com os nomes dos pais e a matrícula do aluno.

2^a Forma Normal

Quando a chave primária é composta por mais de um campo.

Todos os campos que não fazem parte da chave devem depender da chave (de todos os seus campos).

Caso algum campo dependa somente de parte da chave, então devemos colocar este campo em outra tabela.

Exemplo:

TabelaAlunos

Chave (matricula, codigo_curso)

avaliacao descricao_curso

Neste caso o campo descricao_curso depende apenas do codigo_curso, ou seja, tendo o código do curso conseguimos sua descrição. Então esta tabela não está na 2^a Forma Normal.

Solução:

Dividir a tabela em duas (alunos e cursos) e incorporar à chave existente:

TabelaAlunos

Chave (matricula, codigo_curso)

avaliacao

TabelaCursos

codigo_curso

descricao_curso

3^a Forma Normal

Quando um campo não é dependente diretamente da chave primária ou de parte dela, mas de outro campo da tabela que não pertence à chave primária. Quando isso ocorre esta tabela não está na terceira forma normal e a solução é dividir a tabela.

Um atributo comun não deve depender de outro atributo comun, mas somente da PK.

Exemplo:

Campo tipo total, que depende de deus fatores: quantidade e preco.

Solução:

Não devemos armazenar campos calculados na tabela, pois além de gerar inconsistência são desnecessários, já que podemos consegui-lo com uma operação.

4a.Forma Normal

Uma tabela está na 4FN, se e somente se, estiver na 3FN e não existirem dependências multivaloradas.

Exemplo: Dados sobre livros

Relação não normalizada: Livros(nrol, (autor), título, (assunto), editora, cid_edit, ano_public)

1FN: Livros(nrol, autor, assunto, título, editora, cid_edit, ano_public)

2FN: Livros(nrol, título, editora, cid-edit, ano_public)

AutAssLiv(nrol, autor, assunto)

3FN: Livros(nrol, título, editora, ano_public)

Editoras(editora, cid-edit)

AutAssLiv(nrol, autor, assunto)

* Redundância para representar todas as informações

* Evitar todas as combinações: representação não-uniforme (repete alguns elementos ou posições nulas)

Passagem à 4FN:

* Geração de novas tabelas, eliminando Dependências Multivaloradas

* Análise de Dependências Multivaloradas entre atributos:

* autor, assunto - Dependência multivalorada de nroL

5a. Forma Normal

Está ligada à noção de dependência de junção. Se uma relação é decomposta em várias relações e a reconstrução não é possível pela junção das outras relações, dizemos que existe uma dependência de junção.

Existem tabelas na 4FN que não podem ser divididas em duas relações sem que se altere os dados originais.

Exemplo: Seja as relações R1(CodEmp, CodPrj) e R2(CodEmp, Papel) a decomposição da relação ProjetoRecurso(CodEmp, CodPrj, Papel).

Dica:

Em caso de dúvida, o modelo ideal é o menos engessado, mesmo que paguem o preço de alguma redundância.

- Ao aplicar a 3a. FN ainda persistirem redundâncias, então divide-se a tabela em duas.

Exemplos de Relacionamentos tipo N para N

Autores – Músicas

Alunos – Disciplinas

Produtos – Fornecedores

Produtos – Pedidos

Fases na criação de um banco de dados (aplicativo):

- Análise
- Projeto
- Implementação
- Testes
- Homologação
- Administração

Análise

Nesta fase "enfrentamos" o mundo real para analisar e diagnosticar o problema para então passar um esboço do modelo para a fase de projeto.

Nessa fase procedemos ao levantamento, gerenciamento e validação de informações importantes para o modelo.

Análise é a fase que tem contato com o mundo real, coletando informações junto ao cliente (usuário). Depois da coleta realiza a modelagem das entidades e relacionamentos e das normalizações. Ao final deve gerar um diagrama das entidades e relacionamentos DER.

Projeto - definição das tabelas, índices, chaves (PK e FK), relacionamentos, views, etc.

Fases do Projeto do Banco de Dados:

- Modelagem Conceitual
- Projeto Lógico

Modelo Conceitual - Define apenas quais os dados que aparecerão no banco de dados, sem se importar com a implementação do banco. Para essa fase o que mais se utiliza é o DER (Diagrama Entidade-Relacionamento).

Modelo Lógico - Define quais as tabelas e os campos que formarão as tabelas, como também os campos-chave, mas ainda não se preocupa com detalhes como o tipo de dados dos campos, tamanho, etc.

Não devemos misturar as tarefas de cada uma das fases.

Implementação - criação do banco de acordo com o projeto e também criação dos usuários, grupos e privilégios.

Análise e projeto são fases lógicas e implementação é física.

Comparando Construção de bancos de dados com edifícios:

Análise - reconhecimento do terreno e elaboração de croquis (esboço da planta).

Projeto - Elaboração das plantas de acordo com o esboço: planta baixa, de elevação, detalhes, de situação, etc.

Implementação - Construção do edifício.

Administração - manutenção do edifício.

5) Integridade Referencial

Tradução livre do documentação "CBT Integrity Referential":
http://techdocs.postgresql.org/college/002_referentialintegrity/.

Integridade Referencial (relacionamento) é onde uma informação em uma tabela se refere à informações em outra tabela e o banco de dados reforça a integridade.

Tabela1 -----> Tabela2

Onde é Utilizado?

Onde pelo menos em uma tabela precisa se referir para informações em outra tabela e ambas precisam ter seus dados sincronizados.

Exemplo: uma tabela com uma lista de clientes e outra tabela com uma lista dos pedidos efetuados por eles.

Com integridade referencial devidamente implantada nestas tabelas, o banco irá garantir que você nunca irá cadastrar um pedido na tabela pedidos de um cliente que não exista na tabela clientes.

O banco pode ser instruído para automaticamente atualizar ou excluir entradas nas tabelas quando necessário.

Primary Key (Chave Primária) - é o campo de uma tabela criado para que as outras tabelas relacionadas se refiram a ela por este campo. Impede mais de um registro com valores iguais. É a combinação interna de UNIQUE e NOT NULL.

Qualquer campo em outra tabela do banco pode se referir ao campo chave primária, desde que tenham o mesmo tipo de dados e tamanho da chave primária.

Exemplo:

clientes (codigo INTEGER, nome_cliente VARCHAR(60))

```
codigo nome_cliente
1 PostgreSQL inc.
2 RedHat inc.
```

pedidos (relaciona-se à Clientes pelo campo cod_cliente)
cod_pedido cod_cliente descricao

Caso tentemos cadastrar um pedido com cod_cliente 2 ele será aceito.

Mas caso tentemos cadastrar um pedido com cod_cliente 3 ele será recusado pelo banco.

Criando uma Chave Primária

Deve ser criada quando da criação da tabela, para garantir valores exclusivos no campo.

```
CREATE TABLE clientes(cod_cliente BIGINT, nome_cliente VARCHAR(60)
PRIMARY KEY (cod_cliente));
```

Criando uma Chave Estrangeira (Foreign Keys)

É o campo de uma tabela que se relaciona (associa) com o campo Primary Key de outra. O campo pedidos.cod_cliente refere-se ao campo clientes.codigo, então pedidos.cod_cliente é uma chave estrangeira, que é o campo que liga esta tabela a uma outra.

```
CREATE TABLE pedidos(
cod_pedido BIGINT,
cod_cliente BIGINT REFERENCES clientes,
descricao VARCHAR(60)
);
```

Outro exemplo:

FOREIGN KEY (campoa, campob)

```
REFERENCES tabela1 (campo1, campo2)
ON UPDATE CASCADE
ON DELETE CASCADE);
```

Cuidado com exclusão em cascata. Somente utilize com certeza do que faz.
 Dica: Caso desejemos fazer o relacionamento com um campo que não seja a chave primária, devemos passar este campo entre parênteses após o nome da tabela e o mesmo deve obrigatoriamente ser UNIQUE.

```
...
cod_cliente BIGINT REFERENCES clientes(nomecampo),
...
```

Parâmetros Opcionais:

ON UPDATE parametro e ON DELETE parametro.

ON UPDATE parmentros:

NO ACTION (RESTRICT) - quando o campo chave primária está para ser atualizado a atualização é abortada caso um registro em uma tabela referenciada tenha um valor mais antigo. Este parâmetro é o default quando esta cláusula não recebe nenhum parâmetro.

Exemplo: ERRO Ao tentar usar "UPDATE clientes SET codigo = 5 WHERE codigo = 2. Ele vai tentar atualizar o código para 5 mas como em pedidos existem registros do cliente 2 haverá o erro.

CASCADE (Em Cascata) - Quando o campo da chave primária é atualizado, registros na tabela referenciada são atualizados.

Exemplo: Funciona: Ao tentar usar "UPDATE clientes SET codigo = 5 WHERE codigo = 2. Ele vai tentar atualizar o código para 5 e vai atualizar esta chave também na tabela pedidos.

SET NULL (atribuir NULL) - Quando um registro na chave primária é atualizado, todos os campos dos registros referenciados a este são setados para NULL.

Exemplo: UPDATE clientes SET codigo = 9 WHERE codigo = 5;
 Na clientes o codigo vai para 5 e em pedidos, todos os campos cod_cliente com valor 5 serão setados para NULL.

SET DEFAULT (assumir o Default) - Quando um registro na chave primária é atualizado, todos os campos nos registros relacionados são setados para seu valor DEFAULT.

Exemplo: se o valor default do codigo de clientes é 999, então

UPDATE clientes SET codigo = 10 WHERE codigo = 2. Após esta consulta o campo código com valor 2 em clientes vai para 999 e também todos os campos cod_cliente em pedidos.

ON DELETE parametros:

NO ACTION (RESTRICT) - Quando um campo de chave primária está para ser deletado, a exclusão será abortada caso o valor de um registro na tabela referenciada seja mais velho. Este parâmetro é o default quando esta cláusula não recebe nenhum parâmetro.

Exemplo: ERRO em DELETE FROM clientes WHERE codigo = 2. Não funcionará caso o cod_cliente em pedidos contenha um valor mais antigo que codigo em clientes.

CASCADE - Quando um registro com a chave primária é excluído, todos os registros relacionados com aquela chave são excluídos.

SET NULL - Quando um registro com a chave primária é excluído, os respectivos campos na tabela relacionada são setados para NULL.

SET DEFAULT - Quando um registro com a chave primária é excluído, os campos respectivos da tabela relacionada são setados para seu valor DEFAULT.

Excluindo Tabelas Relacionadas

Para excluir tabelas relacionadas, antes devemos excluir a tabela com chave estrangeira. Tudo isso está na documentação sobre CREATE TABLE:

<http://www.postgresql.org/docs/8.0/interactive/sql-createtable.html>

ALTER TABLE

<http://www.postgresql.org/docs/8.0/interactive/sql-altertable.html>

Chave Primária Composta (dois campos)

```
CREATE TABLE tabela (
    codigo INTEGER,
    data DATE,
    nome VARCHAR(40),
    PRIMARY KEY (codigo, data)
);
```

Alguns Modelos de Sistemas

Sistema de Contas a Pagar e Receber (Entidades e Relacionamentos)

Endereço Tipo de Logradouro
Logradouro
Bairro
Município
UF
Telefone

Pessoa	
Física	Jurídica
Pagamento	
CPagar	CReceb
Recebimento	

Sistema Acadêmico

- Cadastro do curso
- Cadastro de disciplina
- Cadastro de aluno
- Cadastro de professor

- Abrir turma
- Matrícular aluno
- Emitir diário
- Lançar avaliação
- Emitir histórico
- Consultar avaliação
- Alocar professor

Sistema de Biblioteca

Atores	Entidades	Atores
bibliotecário	Cadastrar assunto Cadastrar autor Cadastrar departamento Cadastrar funcionário Cadastrar obra Cadastrar editora Cadastrar exemplar Cadastrar fornecedor	RH
funcionário	Cadastrar usuário Cadastrar requisição Cadastrar motivo de manutenção Efetuar empréstimo Efetuar devolução Registrar manutenção Registrar reserva Consultar obras Consultar movimentação de usuários	Setor de compras Setor de manutenção
adm do banco		usuário

Sistema Bancário

Atores	Entidades
cliente	Abrir conta Movimentar Conta Solicitar talão de cheques Solicitar empréstimo Solicitar cartão de crédito Comprar com cartão de crédito Pagar fatura do cartão Contratar seguro
gerante	Cadastrar bandeira do cartão de crédito Cadastrar débito automático Cadastrar tipos de seguro Cadastrar cliente Liberar empréstimo
adm do banco	Cadastrar estabelecimento comercial Cadastrar convênio Cadastrar gerente Cadastrar agência

Sistema Comercial

Fornecedor Lote Produto

Nota Fiscal Movimento estoque

Cliente Nota Fiscal Itens

Fonte destes modelos: Revista SQL Magazine 53

Um bom artigo sobre projeto de software:

Programação (I) - Planejamento e Otimização, de Edvaldo Silva de Almeida Júnior no Viva o Linux: <http://www.vivaolinux.com.br/artigos/verArtigo.php?codigo=8057>

Tabelas são mais conceituais e menos físicas que arquivos.

Simplificando:

- DDL e DCL é para o DBA
- DML é para o programador

SGBDs

SGBDR – SGBD Relacional - tabelas são formadas por linhas e colunas (bidimensional)

SGBDOR - Combina os modelos OO e Relacional

SGBDOO - Modelo OO puro

Características de SGBDs:

- Controle de redundância
- Compartilhamento de dados
- Controle de acesso
- Esquematização (relacionamentos armazenados no banco)
- Backups

Objetivo do SGBD - armazenar objetos de forma a tornar ágil e segura a manipulação das informações.

MER - Modelo Entidades e Relacionamentos - é o modelo composto de entidades e relacionamentos

DER - Diagrama Entidades e Relacionamentos - é o resultado da fase inicial do projeto (análise)

Exemplos de Entidades:

- Físicas ou jurídicas - pessoas, funcionários, clientes, vendedores, fornecedores, empresas, filiais
- Documentos - ordens de compra, ordens de serviço, pedidos, notas fiscais
- Tabelas - CEPs, UF, cargos, etc

Tupla (termo para a fase de projeto)

Registro (termo para a fase de implementação)

Fase de Análise

Ao definir uma entidade pergunte:

- Há informação relevante sobre esta entidade para a empresa?

Faça um diagrama da entidade com alguns atributos

Relacionamentos

A B
1 ----- N

Perguntar se A tem 1 ou + B

Perguntar se B tem somente 1 A

Relacionamentos N - N devem ser divididos em dois 1 para N:

1 ----- N e N ----- 1

Referência: Livro Instant SQL Programming - Joe Celko

Dimensionamento de Hardware

PostgreSQL: como fazer um elefante voar

Submitted by Andre Felipe Machado on

Obs: Mantido pois não mais encontrei o texto na internet.

Dicas de tuning e desempenho para PostgreSQL

Se você já esgotou o arsenal de otimizações e tunings de desempenho para PostgreSQL, está na hora de conhecer alguns segredos de ultra desempenho para servidores de banco de dados. Destes que são divulgados como “novo recorde de TPC”, por algumas empresas.

Digamos que você já modelou corretamente seu banco de dados, já normalizou, já [criou](#) os índices com seus respectivos [tipos mais indicados](#), já [baniu os left outer joins](#) e reescreveu as queries, já analisou suas queries com EXPLAIN e reescreveu-as, seguiu [dicas de desempenho](#) do PostgreSQL.

Você analisou os índices de suas tabelas e criou [clusters em cada tabela](#) com seus índices. Os [ganhos POTENCIAIS](#) são muito bons.

Já analizou os dados de profilers [pg_top](#) e [pgfouine](#) e reescreveu as queries.

Não existe tuning que resolva um banco de dados mal modelado, e com queries mal concebidas.

Você teria de pagar um [preço](#) elevado em [força bruta](#) havendo ainda outras formas de obter o desempenho necessário. Deixe isso para outras situações.

Você também modificou sua aplicação para utilizar [lazy connections](#), se for o caso. Ou passou a utilizar [connection pooling](#), situação bem mais comum e genérica, para poder configurar mais flexivelmente o max_connections.

Depois disso, você resolveu aumentar o número de servidores, usando [balanceamento e replicação síncrona ou assíncrona](#).

Então você decidiu melhorar o hardware, usando mais núcleos de CPU, mais RAM.

Em seguida, aumentou o número de discos para distribuir tabelas por eles [gerenciando tablespaces](#). (voltaremos a esse assunto).

E já descobriu que RAID 5 ou 6 são [muito lentos comparados](#) a RAID 1. E ainda possuem problema do [write hole](#) por arquitetura.

Voltou uns passos e [reviu configurações](#) de memória para índices, espaço de trabalho e outros no postgresql.conf.

Porque o Postgresql não utiliza automaticamente a RAM da melhor forma na configuração default muito conservadora.

Se tiveres recursos, conseguiu adquirir RAM suficiente para armazenar seu dataset ativo todo em RAM.

Aproveitou e reviu as configurações de vacuum, background writer e I/O concurrency (voltaremos a esses).

Como você tem mais discos agora, reavaliou [configurações do query planner](#).

Também lembrou de [não ter excessivo log de estatísticas](#), nem [log de erros demais](#), apenas o mínimo seguro, no [melhor local](#), e para o query planner, pois a aplicação já está depurada neste estágio. Voltaremos a esses tópicos.

Configurou o syslog para comportamento assíncrono, claro. E num disco separado (não basta partição separada).

Ainda não foi suficiente para dar conta da carga.

O que ainda pode ser feito?

De volta aos conceitos.

Você reparou no que é um [ACID database](#) como é o PostgreSQL?

Para ter Consistência e Durabilidade, um dos requisitos é **armazenar os dados confiavelmente**.

Para fazer isso confiavelmente, o PostgreSQL, entre outras características, utiliza o [Write Ahead Log, WAL](#).

WAL, [também conhecido por log de transações e REDO log](#).

Para obter mais desempenho é vital compreendê-lo antes de fazer [tuning de WAL](#):

Resumidamente, o PostgreSQL escreve as transações sincronamente PRIMEIRO no WAL e periodicamente as repassa (checkpoint, flush) para a área durável final.

Releia o parágrafo acima 5 vezes e pense nas implicações de cada palavra e perceberás onde iremos chegar.

O [WAL](#) também é utilizado para recursos de replicação, fazendo seu comportamento físico no disco um tanto diferente do inicialmente esperado.

Para PostgreSQL ter melhor desempenho, a LATÊNCIA de acesso ao WAL deve ser baixa.

Lembre: SGBDs escrevem em tamanho de blocos definidos e configuráveis.

Em algumas situações de modelo de dados pode ser necessário alterar o tamanho do bloco. E configurar o sistema todo para isso.

No Blog TechForce há uma boa quantidade de artigos e bibliografia sobre redução de latência de acesso a arquivos pequenos, tuning de kernel, de XenServer, de filesystems,

multipath, LVM, data storage, redução de latência de rede para acessos pequenos. Estude e aplique todos.

Mesmo depois de tudo isso, ainda não é suficiente para a carga exigida.

Chegou a hora de [PostgreSQL em mainframe?](#)

Calma. É uma solução de maior custo para algumas situações extremas. Antes disso precisamos explorar as alternativas.

Então vamos estudar os segredos de força bruta das soluções high end em plataforma baixa. Aqui se fala de IOPS não de throughput.

Consistência e Durabilidade.

Em 2012, a tecnologia de armazenamento corporativa é “disco”, de algum tipo.

No segmento enterprise, temos discos [Enterprise SATA](#), [NL-SAS](#) , [SAS](#), [FC-AL](#), [Infiniband SAS](#), variando de 5,4 krpm a 15 krpm, MLC Flash SSD, SLC Flash SSD, e o topo de linha [SLC DRAM SSD](#).

Uma característica pobre pouco divulgada das Flash SSD é que degradam muito o desempenho e rapidamente.

Algo como de 40 mil IOPS para 5 mil IOPS após poucas centenas de ciclos de escrita. Basicamente, assim que todas as células tiverem sido escritas uma vez, exigirão uma operação de apagamento para serem reescritas. E isso é lento.

Ainda, [Multi Level Cell Flash SSD](#) são [mais baratas e lentas](#) que Single Level Cell Flash SSD.

O que ainda não contaram para você:

Vamos conhecer alguns dos segredos de força bruta para appliances servidores de bancos de dados high end.

Vejamos **alguns** exemplos de armazenamento de altíssimo desempenho. Compare os dados técnicos com o que você conhecia até agora:

A lista não é nem completa nem implica recomendação de qualquer tipo. Apenas uma **amostra** de cenário corporativo nesta data.

Fusion i-o drive

<http://www.dell.com/br/corporativo/p/fusion-io-drive>

<http://www.fusionio.com/platforms/iodrive2/>

Zeus RAM SSD

<http://www.stec-inc.com/product/zeusram.php>

Gavetas de expansão para instalar discos enterprise em conexão direta

<http://www.dell.com/br/corporativo/p/disk-expansion-enclosures>

http://www.sgi.com/products/storage/jbod/mis_jbod.html

Data storage completo em RAM

<http://www.ramsan.com/products/rackmount-ram-storage/ramsan-420>

Data storage flash array

<http://www.oracle.com/us/products/servers-storage/storage/flash-storage/f5100/overview/index.html>

Quinze microssegundos de latência. ISSO é baixa latência.

De um milhão de IOPS até 50 milhões de IOPS num rack. ISSO é desempenho.

Com esse tipo de hardware, e iSCSI HBA em redes 10 Gb/s segmentadas, ou FC 8 Gb/s ou redes Infiniband você poderá reavaliar e reconfigurar seus servidores, seguindo um roteiro:

1. Vacuum, background writer, I/O concurrency.
2. Tabelas de estatísticas.
3. Log de erros e atividade.
4. Syslog, arquivos temporários.
5. Write Ahead Log, checkpoints, WAL archiving.
6. Clusters sobre os índices em cada tabela.
7. [índices em tablespaces próprios](#).
8. Tablespaces para tabelas mais ativas.
9. Configurações do query planner.

Redistribuir o tablespaces fica por último pois é o passo que demanda mais espaço físico de disco, que é o recurso caro neste estágio. Com o novo desempenho dos discos, o query planner *poderia* ser reparametrizado para calcular os custos melhor. Avalie a necessidade com o EXPLAIN.

Como você percebeu, isto é um roteiro de orientações, não uma receita.

Tuning envolve muita análise específica de cada situação individual. E cada tópico de análise merece um livro.

Além do banco de dados, você fará tuning do sistema operacional e do hardware para essa aplicação. Os servidores terão dedicação exclusiva, devido à especialização.

Envie suas sugestões como comentários a este artigo.

Bibliografia adicional

http://www.solarisinternals.com/wiki/index.php/ZFS_for_Databases

https://blogs.oracle.com/paulvandenbogaard/entry/running_postgresql_on_zfs_file

<http://blog.dryft.net/2011/09/benchmarking-zfs-xfs-ext4-and-btrfs.html>

Algumas dicas para o Dimensionamento do hardware de um Servidor de Bancos de Dados com PostgreSQL

O objetivo deste projeto é o de elaborar um planejamento para a administração de bancos de dados com o PostgreSQL, partindo do dimensionamento do hardware, a instalação do sistema operacional, a virtualização, a instalação do PostgreSQL e sua administração, com foco na administração.

Dimensionamento do Hardware

- Computador(es), memória, CPU, discos, etc
- Placas de rede e cabos
- Switchs
- Garantias
- Storages
- Nobreaks
- Internet e banda
- Sugestão de fabricantes

Vantagens da Virtualização:

- Incrementar a utilização do hardware
- Economizar energia elétrica e reduzir a emissão de Dióxido de Carbono
- Redução de hardware físico e de pessoal de manutenção
- Simplificação de backup e restore

Pequenas empresas costumam escolher uma combinação de hardware e sistema operacional já integrado por fabricantes mais conhecidos, como Dell, HP e IBM, ou um pacote montado por algum consultor de TI ou revendedor de sua confiança. Saiba que será muito importante o conhecimento do revendedor para traduzir as necessidades do seu negócio para especificações de um servidor (hardware).

Esses profissionais podem dimensionar o equipamento para suas necessidades atuais e garantir que o sistema possa evoluir conforme sua empresa cresça. Isso é muito importante!

Se a finalidade deste equipamento é armazenar e compartilhar arquivos importantes, recomendamos o uso de uma proteção contra falha no disco como HDs em RAID, que são grupos de discos com redundância e espelhamento.

Mais eficiente, porém mais caros, são os dispositivos de storage, unidades externas de discos com alto desempenho e segurança contra falhas. Mesmo assim, tais dispositivos apresentam um custo-benefício que merece ser avaliado e que pode caber dentro do seu orçamento, principalmente se o que conta é a continuidade do seu negócio.

Não custa repetir: backup é essencial. As unidades de fita evoluíram bastante e por isso continuam sendo muito usadas. Tecnologias DAT, DLT, LTO são recomendadas para copiar todos os dados diariamente em fitas que guardam de 40 GB até 400 GB.

No outro extremo, ou seja, quem precisa de um servidor para manipular um banco de dados, CRM, ERP, com 250 usuários ou mais, terá de escolher um sistema que tenha de dois a quatro processadores dual ou quad-core Xeon, 16 GB de memória e rodando a versão completa do Windows Server 2008 64 bits e respectivos aplicativos (Exchange, SQL, etc.) todos também 64 bits.

<http://pcworld.com.br/dicas/2008/06/20/qual-a-configuracao-de-hardware-mais-adequada-para-o-servidor/>

Quando o dimensionamento de hardware para um banco de dados o foco deve ser em três objetivos operacionais:

Estimar os requisitos de armazenamento básicos para um banco de dados

Estimar a quantidade de memória necessária para o processamento

Estimar a quantidade de CPU necessária para carga de trabalho emitidas a partir de aplicativos clientes, como manipulação de dados (inserção, exclusão e atualização) bem como relatórios.

<http://www.tiagobalabuch.com/my-product/dimensionamento-de-hardware-storage/>

Prezado colega. Segue minha lista de dicas, para o seu questionamento:

1- Consulte a Intel pelo e-mail: Programas_ITR@lar-newsletter.intel.com

2- Consulte a AMD pelo e-mail: supportone@amdpartneradvantage.com

3- Consulte a Solution TI pelo e-mail: solution@solutionti.com.br (distribuidor oficial)

4- Consulte a Dell Inc. pelo fone: 0800-7012538 (Servidores PowerEdge)

5- Não exagere no tamanho. Prefira um equipamento flexível.

6- Não vale apena arriscar! Economia-porca provoca cancer. Lembre que a base de todo o seu negócio de TI é seu SERVIDOR.

7- Compre utilizando Leasing com cláusula de atualização a cada 24 meses, seguro e restituição no Imposto de Renda.

<https://under-linux.org/showthread.php?t=78543>

Para virtualizar precisa prever este requisito na aquisição do Hardware (Intel VT ou AMD-V) na CPU.

http://miniserver.net.br/pfsense_guia_de_dimensionamento.html

<https://confluence.atlassian.com/confbr1/guia-de-requisitos-de-hardware-de-servidor-933709557.html>

<https://docops.ca.com/display/UIM847BR/Preparar+o+hardware+do+servidor>

<http://www.dell.com/br/empresa/p/networking-products>

<https://docs.oracle.com/cd/E19226-01/821-1337/abpaj/index.html>

http://www.ufjf.br/eduardo_barrere/files/2013/11/Aula_03_2-2013.pdf

<http://www.adnlogico.pt/infra-estrutura/equipamentos-e-servidores>

http://www.aedb.br/seget/arquivos/artigos10/26_Virtualizacao_SEGET_2010.pdf

Discos em RAID

O sistema RAID consiste em um conjunto/matriz de dois ou mais discos rígidos com dois objetivos básicos, tornar o sistema mais rápido (stripping) e/ou mais seguro contra falhas (espelhamento).

RAID 0, 1, 5, 10, em Matriz

RAID 0 (striping)

Utiliza os recursos de leitura/gravação de dois ou mais discos, trabalhando em conjunto para maximizar o desempenho do armazenamento.

O RAID 0 é o mais rápido de todos os tipos de RAID, mas por outro lado não é forte em termos de segurança. Basta que qualquer um dos discos falhe para que todos os dados se percam. Isso indica que o RAID 0 não tem qualquer recurso de redundância.

O ganho de desempenho é exatamente igual ao número de discos utilizados, 2 discos duas vezes mais rápido, 5 discos 5 vezes mais rápido, etc.

Este RAID oferece o aproveitamento máximo dos discos.

O conjunto RAID 0 aparece para o sistema operacional como um único disco. Exemplo: se tivermos 4 discos de 240GB em RAID 0 o SO verá um único disco de 960GB.

RAID 1 (espelhamento)

Contém dois ou múltiplos de 2 discos rígidos, onde os dados são espelhados nos dois em tempo real. Os dados são duplicados nos dois discos e o SO percebe como se fosse apenas um disco. Exemplo: 4 discos de 120GB em RAID 2 são vistos como se fosse apenas um disco de 240GB. Caso uma unidade de disco falhe todos os dados estarão automaticamente disponíveis na outra unidade. Com o espelhamento o conteúdo de um disco é inteiramente copiado para o outro de forma automática, feito por hardware.

Este é o RAID mais simples e relativamente mais caro.

RAID 5

Uma matriz com 3 ou mais discos rígidos. Principais benefícios: capacidade de armazenamento e proteção dos dados.

A capacidade de uma matriz RAID 5 é igual ao tamanho da menor unidade multiplicado pelo número de unidades da matriz menos um.

Se você tem um grande volume de dados estáticos a serem gravados, com muita leitura e pouca gravação, o RAID 5 pode ser para você.

Permite que se perca até um disco com segurança. Quem se preocupa com segurança não usa RAID 5. Se a preocupação é com desempenho use RAID 5 mas com o mínimo de 5 discos.

Exemplo: quatro discos de 120GB em matriz RAID 5 aparece para o SO como um disco de 360GB.

$$TR5 = 120 \times (4 - 1) = 120 \times 3 = 360.$$

Calculadora de RAID Online = <https://www.grijpink.eu/tools/raid/index.php>

O desempenho de leitura de uma matriz em RAID 5 é melhor que um disco único, por que os dados podem ser lidos em múltiplos discos simultaneamente.

RAID 10

Uma matriz RAID 10 usa 4 discos rígidos para criar uma combinação de nível RAID 0 e 1 formando um RAID 0 de matriz de duas matrizes RAID 1.

Nosso caso uma matriz RAID 10 com 4 discos de 120GB será vista pelo SO como um disco de 240GB.

Os principais benefícios do RAID 10 é que combina os benefícios de desempenho do RAID 0 com os benefícios de tolerância a falhas do RAID 1.

Começamos a usar o RAID 10 com 4 discos e daí em diante com números pares: 6, 8, 10, 12, etc. O único problema do RAID 10 é o custo.

Em geral quanto mais discos você colocar na matriz mais rápido o acesso será para todas as operações.

Quando devemos usar RAID? Sempre, escolhendo RAID 10, 1, 5 ou 6 dependendo das prioridades.

Aumentar o número de discos é sinônimo de aumentar o desempenho. Não existe servidor de bancos de dados sério com apenas 1 disco. Servidores podem usar RAID 1 com 2 discos e maiores com 4 discos ou 6 em RAID 10.

Para grandes necessidades de armazenamento considerar a aquisição de storage externo.

Se o servidor é importante ele deve ser um servidor dedicado, com apenas o servidor de bancos de dados.

Tipos de Interfaces para Discos Rígidos

SCSI, SAS, Fibras e SATA. Evitar o uso da SATA, por ser lento demais.

Evite servidores com 1U (altura do servidor no rack) que suportam apenas 2 discos. Prefira os com 2U.

Medindo desempenho de discos:

```
sudo su
su - postgres
pgbench -i nomebanco 1000
```

Password:

```
NOTICE: table "pgbench_history" does not exist, skipping
NOTICE: table "pgbench_tellers" does not exist, skipping
NOTICE: table "pgbench_accounts" does not exist, skipping
NOTICE: table "pgbench_branches" does not exist, skipping
creating tables...
100000 of 100000 tuples (100%) done (elapsed 0.06 s, remaining 0.00 s)
vacuum...
set primary keys...
done.
```

Rodar novamente:

```
creating tables...
100000 of 100000 tuples (100%) done (elapsed 0.07 s, remaining 0.00 s)
vacuum...
set primary keys...
```

done.

Efetuar um backup do WAL (logs de transação do PostgreSQL) é considerado obrigatório em ambientes de produção.

Para bases maiores o backup físico é obrigatório. Precisamos de espaço em disco, local ou remoto, para armazenar os datafiles.

O backup lógico não é uma estratégia recomendada para bases médias ou grandes.

Referências:

https://wiki.postgresql.org/wiki/PostgreSQL,_discos_%26_Cia do Fábio Telles

5 - Ferramentas

Aqui trago uma boa relação de ferramentas para trabalhar com o PostgreSQL, que estão organizadas por tipo: administração, modelagem, monitoramento, backup, importar e exportar, visualizar, geração de relatórios, engenharia reversa, etc.

As licenças são variadas: free, free e open, comerciais, sharewares, etc.

Agendamento

Um recurso muito utilizado para agendar backups, vacuums, etc.

No Windows temos o **Agendador de Tarefas**, que na linha de comando pode ser usado como schtasks.

Tutorial sobre o schtasks:

<http://ribafs.wordpress.com/2008/03/29/agendando-tarefas-no-windows-com-o-schtasks/>

No Linux temos o **cron**, para agendar as tarefas. Alguns tutoriais sobre ele:

<http://www.vivaolinux.com.br/dicas/verDica.php?codigo=10282>

<http://www.vivaolinux.com.br/artigos/verArtigo.php?codigo=7965>

Administração

PGAdmin - <http://www.pgadmin.org>

Ferramenta de administração desenvolvida pela equipe de desenvolvimento do PostgreSQL.

Ferramenta para uso com replicação via Slony e muitas outras funcionalidades importantes.

Administração Web

adminer - <http://adminer.org>

download adminer e css - <https://www.adminer.org/#download>

Plugins - <https://www.adminer.org/en/plugins/>

psql - acompanha o PostgreSQL

Ferramenta de administração via console, com muitos recursos.

Adminer – <http://adminer.org>

Features

- **Connect** to a database server with username and password
- Select an existing **database** or create a new one
- List fields, indexes, foreign keys and triggers of table
- Change name, engine, collation, auto_increment and comment of **table**
- Alter name, type, collation, comment and default values of **columns**
- Add and drop tables and columns
- Create, alter, drop and search by **indexes** including fulltext
- Create, alter, drop and link lists by **foreign keys**
- Create, alter, drop and select from **views**
- Create, alter, drop and call **stored procedures and functions**
- Create, alter and drop **triggers**
- **List data** in tables with search, aggregate, sort and limit results
- Insert new **records**, update and delete the existing ones
- Supports all **data types**, blobs through file transfer
- Execute any **SQL command** from a text field or a file
- **Export** table structure, data, views, routines, databases to SQL or CSV
- Print **database schema** connected by foreign keys
- Show **processes** and kill them
- Display **users and rights** and change them
- Display **variables** with links to documentation
- Manage **events** and **table partitions** (MySQL 5.1)
- Schemas, sequences, user types (PostgreSQL)
- [Extensive customization options](#)

Requirements

- Works with MySQL, PostgreSQL, SQLite, MS SQL, Oracle, SimpleDB, Elasticsearch, MongoDB - [Improve your driver](#)
- Supports PHP 5 with enabled sessions
- Available in many languages including Portuguese ([\(36\)](#) - [Create a new translation](#))
- Free for commercial and non-commercial use ([Apache License](#) or [GPL 2](#))

Plugins para o Eclipse:

dbedit: http://www.geocities.com/uwe_ewald/dbedit.html

QuantumDb: <http://quantum.sourceforge.net/>

MicroOLAP Database Design - <http://microolap.com/products/database/postgresql-designer/download/>

Database structure modeling, generation and modification focused on PostgreSQL
Database Designer for PostgreSQL is an easy CASE tool with intuitive graphical interface allowing you to build a clear and effective database structure visually, see the complete picture (diagram) representing all the tables, references between them, views, stored procedures and other objects. Then you can easily generate a physical database on a server, modify it according to any changes you made to the diagram using fast ALTER statements.

Aqua Data Studio - <http://www.aquafold.com/>

Is a database developer's complete Integrated Development Environment (IDE). The IDE provides three major areas of functionality: A) Database query and administration tool B) Suite of compare tools for databases, source control and filesystems, and C) a complete and integrated source control client for Subversion (SVN) and CVS.

OS Support: Windows | Linux | OSX | Solaris | Java Platform

RDBMS Support: (Oracle - 10g/9i/8i) (DB2 UDB - 9/8/7) (MS SQL Server - 2005/2000/7/MSDE) (Sybase ASE - 15/12.x/11.x) (Sybase Anywhere - 10/9/8) (Sybase IQ - 12.x) (Informix IDS - 10/9.x/7.x) (PostgreSQL - 8.x/7.x) (MySQL - 5/4.x/3.x) (Generic JDBC Platform) (Generic ODBC)

Data Architect - <http://www.thekompany.com/products/dataarchitect/>

DbDeveloper - <http://www.dbdeveloper.prominentus.com/>

Are you working with many databases? You migrate data from MSSQL to Oracle, maybe another databases. Do you work with many different databases in the same time? Do you need an easy-to-use sql tool for each of them. It becomes hard when you have to use

different tool for every database engine. Why not having all your databases management in one place and save hours while working with it?

dbDeveloper is a visual development tool for multiple databases. With the intuitive and powerful interface it allows you to explore, create and change structure of every database in an easy way.

It offers variety of functions from development via tuning up to the deployment and data manipulation.

DbManager - <http://www.dbtools.com.br/PT/dbmanagerpro/>

O DBManager Professional é uma das mais avançadas ferramentas para gerenciamento de dados. Com suporte nativo para MySQL, PostgreSQL, Interbase/Firebird, SQLite, Xbase tables, MSAccess, MSSQL Server, Sybase, Oracle e ODBC, traz ainda recursos poderosos disponíveis apenas no DBManager. Disponível em duas edições distintas, você pode escolher aquela que melhor atenderá as suas necessidades: Freeware e Enterprise. A Freeware é totalmente gratuita e voltada para usuários de bancos de dados em geral, contém recursos não disponíveis em outros aplicativos existentes no mercado para gerenciamento de Bancos de Dados. A Edição Enterprise é um produto comercial, voltada para os usuários experientes que querem extrair o máximo em administração de banco de ados.

Navicat - <http://pgsql.navicat.com/>

EMS SQL Manager for PostgreSQL -

<http://sqlmanager.net/products/postgresql/manager/>

EMS SQL Manager for PostgreSQL is a high performance tool for PostgreSQL Database Server administration and development. It works with any PostgreSQL versions up to the newest one and supports the latest PostgreSQL features including enumerated, text search, XML and UUID data types, PostgreSQL index key sorting order, arrays of composite types, and others. SQL Manager for PostgreSQL offers plenty of powerful database tools such as Visual Database Designer to create PostgreSQL database in few clicks, Visual Query Builder to build complicated PostgreSQL queries, powerful BLOB editor and many more useful features for efficient PostgreSQL administration. SQL Manager for PostgreSQL has a state-of-the-art graphical user interface with well-described wizard system, so clear in use that even a newbie will not be confused with it.

SQL Maestro for PostgreSQL - <http://sqlmaestro.com/products/postgresql/maestro/>

PostgreSQL Maestro is the premier PostgreSQL admin tool for database management, control and development.

PostgreSQL Maestro GUI allows you to create, edit, copy, extract and drop all the database objects, build queries visually, execute queries and SQL scripts, design your database as ER diagram, view and edit data including BLOBS, represent data as diagrams, export and import data to/from most popular file formats, debug PL/pgSQL code, analyze your data summarized into multidimensional views and hierarchies (OLAP cubes), manage PostgreSQL roles, users, groups and their privileges, and use a lot of other admin tools designed for the easiest and most efficient work with PostgreSQL database server.

Backup

Cobian Backup - <http://www.educ.umu.se/~cobian/cobianbackup.htm>

Não especificamente para o PostgreSQL.

Cobian Backup is a multi-threaded program that can be used to schedule and backup your files and directories from their original location to other directories/drives in the same computer or other computer in your network. FTP backup is also supported in both directions (download and upload).

Cobian Backup exists in two different versions: application and service. The program uses very few resources and can be running on the background on your system, checking your backup schedule and executing your backups when necessary. Cobian Backup is not an usual backup application: it only copies your files and folders in original or compressed mode to other destination, creating a security copy as a result. So Cobian Backup can be better described as a "Scheduler for security copies".

Cobian Backup supports several methods of compression and strong encryption.

AutoBackup - <http://pgfoundry.org/projects/autobackup/>

Really easy backup script for Postgres with email notifications.

BackupPC - <http://backuppcc.sourceforge.net/>

BackupPC is a high-performance, enterprise-grade system for backing up Linux, WinXX and MacOSX PCs and laptops to a server's disk. BackupPC is highly configurable and easy to install and maintain.

Dirvish - <http://www.dirvish.org/>

Dirvish is a fast, disk based, rotating network backup system.

With dirvish you can maintain a set of complete images of your filesystems with unattended creation and expiration. A dirvish backup vault is like a time machine for your data.

PGBackup Agent - <http://sourceforge.net/projects/pgbackupagent>

PGBackup Agent is a Win32 backup service for PostgreSQL servers

Diversas Ferramentas para PostgreSQL e para outros SGBDs:

<http://www.sqlmanager.net/>

Bancos de Teste

Pagila - <http://www.postgresql.org/ftp/projects/pgFoundry/dbsamples/index.html>

Comparar

pgdiff - <http://pgdiff.sourceforge.net/>

Compares the structures of two PostgreSQL databases and returns the differences as a sequence of SQL commands which can be fed to psql to transform the structure of the first to be identical to the second (analogous to diff and patch). There is an advanced web interface that makes testing and exploration easy. Database schemas can come from live databases, SQL files, or direct input.

Don't gamble on whether or not your databases have the same structure, including constraints, defaults and types. Make sure with pgdiff!

Data Comparer for PostgreSQL -

<http://www.sqlmanager.net/en/products/postgresql/datacomparer>

EMS Data Comparer for PostgreSQL is a powerful and easy-to-use tool for PostgreSQL data comparison and synchronization. Using this utility you can view all the differences in compared PostgreSQL tables and execute an automatically generated script to eliminate these differences. Data Comparer for PostgreSQL provides a wide range of configuration parameters for fast and effective data comparison and synchronization of PostgreSQL databases.

EMS DB Comparer for PostgreSQL -

<http://www.sqlmanager.net/en/products/postgresql/dbcomparer>

Is an excellent tool for database comparison and synchronization. It allows you to view all the differences in compared database objects and execute an automatically generated script to eliminate all or selected differences. Having EMS DB Comparer for PostgreSQL you can work with several projects at once, define comparison parameters, print difference reports, and alter modification scripts. Its user-friendly interface greatly simplifies discovering and eliminating differences in PostgreSQL database structure saving your time and therefore money.

Ambos também para MySQL, SQL Server, Interbase/Firebird e Oracle

DB Data Diffactive V1 - <http://www.datanamic.com/datadiff/index.html>

Compare and Synchronize Database Content

- * Compare contents of two databases.
- * Synchronize database content.
- * Quickly find data differences between two tables.
- * Cross database version comparison.

DB Data Diffactive is a powerful and easy-to-use utility for data comparison and synchronization. Compare data for selected tables in two databases, view differences and publish changes quickly and safely. Flexible comparison and synchronization settings will enable you to set up a customized comparison key and to select tables and fields for comparison and for synchronization.

DB Schema Diffactive V1 - <http://www.datanamic.com/schemadiff/index.html>

Compare and Synchronize Database Schemas

- * Compare database schemas.
- * Synchronize your databases.
- * Quickly find structure differences between two databases.
- * Cross database version comparison.

DB Schema Diffactive is a tool for comparison and synchronization of database schemas. It allows you to compare and synchronize tables, views, functions, sequences (generators), stored procedures, triggers and constraints between two databases.

Conectividade

PSQLODBC - ODBC para PostgreSQL

<http://www.postgresql.org/ftp/odbc/versions/dll/index.html>

NPGSQL - Driver para .NET

<http://www.postgresql.org/ftp/projects/gborg/npgsql-devel/index.html>

PGOLEDB - Driver OLE

<http://www.postgresql.org/ftp/projects/gborg/oledb/stable/index.html>

SDBC - Driver para OpenOffice Base

<http://dba.openoffice.org/drivers/postgresql/index.html>

JDBC para PostgreSQL

<http://jdbc.postgresql.org/download.html>

ZeosLib - Acesso para Delphi

http://sourceforge.net/project/showfiles.php?group_id=35994

Converter/Importar/Exportar

Temos um recurso que é praticamente universal entre os SGBDs:

Exportar como texto puro/CSV e então importar no PostgreSQL através do comando copy com:

```
copy tabela from 'c:/path/do/arquivo.csv';
```

Advanced Data Export VCL -

<http://www.sqlmanager.net/en/products/tools/advancedexport>

Is a component suite for Borland Delphi and C++ Builder that allows you to save your data in the most popular data formats for the future viewing, modification, printing or web publication. You can export data into MS Access, MS Excel, MS Word (RTF), Open XML Format, Open Document Format (ODF), HTML, XML, PDF, TXT, DBF, CSV and more!

There will be no need to waste your time on tiresome data conversion - Advanced Data Export will do the task quickly and will give the result in the desired format.

Advanced Data Import VCL -

<http://www.sqlmanager.net/en/products/tools/advancedimport>

Is a component suite for Borland Delphi and C++ Builder that allows you to import data from files of the most popular data formats to the database. You can import data from MS Excel, MS Access, DBF, XML, TXT, CSV, ODF and HTML. There will be no need to waste your time on tiresome data conversion - Advanced Data Import will do the task quickly, irrespective of the source data format.

PostgreSQL Data Wizard - <http://sqlmaestro.com/products/postgresql/datawizard/>

Is a powerful Windows GUI utility for managing your data. It provides you with a number of easy-to-use tools for performing the required data manipulation easily and quickly.

- * ASP.NET Generator: create full set of ASP.NET scripts in a few mouse clicks
- * PHP Generator: get high-quality web applications without manual coding
- * Data Pump: transfer any ADO-compatible database to PostgreSQL
- * Data export to as many as 14 file formats including Excel, RTF and HTML
- * Data import from Excel, CSV, text files and more
- * Flexible Task Scheduler
- * The Agent application to execute tasks in background mode

PGDoc - <http://www.assembla.com/wiki/show/pgdoc>

Is a small Perl utility which automatically generates pretty HTML documentation for any PostgreSQL database. It runs through the system tables and collects information about database objects and relationships and creates an HTML page. The format of the HTML may be customized by altering the template and/or the css file.

Protopg - <http://pgfoundry.org/projects/protopg/>

Protopg is a parser/translator that helps to migrate Oracle(TM) SQL and PL/SQL statements to PostgreSQL syntax.

Iconv - <http://www.gnu.org/software/libiconv/>

Converte arquivos de uma codificação em outra.

Versão for Windows - <http://gnuwin32.sourceforge.net/packages/libiconv.htm>

Editores para Linguagens

DreamCoder for PostgreSQL - <http://www.sqldeveloper.net/postgresql-manager-developer/dreamcoder-for-postgresql.html>

is a powerful Integrated Development Environment (IDE) for PostgreSQL Databases. With the intuitive DreamCoder's GUI you will increase your code quality and reduce the development process time.

DreamCoder for PostgreSQL works with all PostgreSQL Server versions from 8.0 to 8.1.4. DreamCoder for PostgreSQL enables you to easily build and execute queries, build and execute scripts, compile PL/pgSQL code, create and modify database objects, import and export data, enable user session and database monitor and more.

DreamCoder for PostgreSQL offers powerful visual tools for increase your productivity like a SQL editor, PL/pgSQL editor, master detail table browser, database structure synchronization parameter manager, SQL formatter, query builder, SQL history, session monitor and more.

Versões Free e Professional

PL/pgSQL Debugger - <http://www.amsoftwaredesign.com/>

Lightning Admin for PostgreSQL and MySQL (LA) is a GUI administration program that is designed for use on the Microsoft Windows® family of operating systems.

LA implements a tabbed workspace that is similar to a modern programming IDE such as Visual Studio® or Borland Delphi®.

Lightning Admin can be used to connect to databases running on any Platform PostgreSQL or MySQL run on.

Veja também este site:

http://wwwpgsql.cz/index.php/Write_a_PL/pgSQL_debugger_alias_advanced_techniques_of_programming_in_PostgreSQL

edb-debugger - <http://pgfoundry.org/projects/edb-debugger/>

PL/pgSQL Debugger. Currently requires PostgreSQL 8.2 or higher (or EnterpriseDB 8.1.4 or higher). The PL/pgSQL debugger lets you step through PL/pgSQL code, set and clear breakpoints, view and modify variables, and walk through the call stack. Windows NT/2000/XP/2003, Linux

Engenharia Reversa - Visualização

DbVisualizer is the leading platform independent and cross database tool aimed to simplify database development and management for database administrators and developers.

DbVisualizer - <http://dbvis.com/>

Exibe diagramas de grande quantidade de tabelas e bancos, formando grandes imagens de vários metros, apropriada para plotagem.

Plugin para o Eclipse:

Azzurri Clay - <http://www.esnips.com/nsdoc/5e7e0e72-a935-4a45-a23d-cc38a9b1c139>

Geográficos

Uma das diferenças e forças do PostgreSQL são seus recursos para trabalhar com informações geográficas.

gnuplot - um software para trabalhar com recursos gráficos na linha de comando e que pode ser integrado ao PostgreSQL.

Instale com:

sudo apt-get install gnuplot

Existem versões para outros SOs, inclusive para Windows:

http://sourceforge.net/project/showfiles.php?group_id=2055

Veja um capítulo de livro sobre a integração com o PostgreSQL:

<http://pg.ribafs.net/down/prog//Using%20PostgreSQL%20and%20gnuplot.pdf>

phpPgGIS - <http://www.geolivre.org.br/?q=node/2> e

<http://sourceforge.net/projects/phppgsql/>

O phpPgGIS é um sistema que foi criado a partir do phpPgAdmin e inclui a capacidade de interpretar dados espaciais armazenados com o PostGIS. Este sistema torna muito fácil a gerência de um banco de dados geográfico baseado no PostgreSQL/PostGIS.

PostGIS - <http://postgis.refractions.net/>

PostGIS adds support for geographic objects to the PostgreSQL object-relational database. In effect, PostGIS "spatially enables" the PostgreSQL server, allowing it to be used as a backend spatial database for geographic information systems (GIS), much like ESRI's SDE or Oracle's Spatial extension. PostGIS follows the OpenGIS "Simple Features Specification for SQL" and has been certified as compliant with the "Types and Functions" profile.

PostGIS has been developed by Refractions Research as a project in open source spatial database technology. PostGIS is released under the GNU General Public License. We continue to develop PostGIS, and have added user interface tools, basic topology support, data validation, coordinate transformation, programming APIs and much more. Our list of future projects includes full topology support, raster support, networks and routing, three dimensional surfaces, curves and splines and other features. Ask us about consulting services and implementing new features.

PostGeoOLAP - <http://pgfoundry.org/projects/postgeoolap/>

A tool for creating Spatial OLAP solutions on top of PostgreSQL + PostGIS (although working with solely conventional data is also possible). Results are presented in a spreadsheet frame (for conventional data) and in maps (for geographic data).

Geradores de Código

PHP Code Generator - <https://sourceforge.net/projects/phpcg/?source=directory>

phpCodeGenerator is a free database driven website code generator. It reads the database and generates a website with the ability to Create, List, Edit, Update, Delete and Search Records.

Trabalha com PostgreSQL, MySQL, Access, Oracle, etc.

PostgreSQL PHP Generator - <http://sqlmaestro.com/products/postgresql/phpgenerator/>

Is a FREEWARE but powerful PostgreSQL GUI frontend that allows you to generate high-quality PostgreSQL PHP scripts for the selected tables, views and queries for the further working with these objects through the web.

- * Data management: adding, editing and deleting records
- * Customization of the HTML appearance
- * Filtering and sorting abilities
- * Data protection with a lot of security settings
- * Lookup options for master-detail relations
- * Integrated script navigation
- * Ability to create multilingual web apps

Modelagem

DDT - Database Design Tool -
Free e open, simples e eficiente.

DeZign for Databases V5 - <http://www.datanamic.com/dezign/index.html>

DeZign for Databases is an intuitive database design tool for developers and DBA's that can help you model, create and maintain databases. DeZign for Databases uses entity relationship diagrams (ERDs) to graphically design databases and automatically generates the most popular SQL and desktop databases.

DbDesigner - <http://www.fabforce.net/dbdesigner4/>

DBDesigner 4 is a visual database design system that integrates database design, modeling, creation and maintenance into a single, seamless environment.

DbDesigner Fork - <http://sourceforge.net/projects/dbdesigner-fork>

DB Designer Fork is a fork of the fabFORCE DBDesigner 4. DBDesigner is a visual database design system that integrates entity relationship design and database creation. DB Designer Fork generates SQL scripts for Oracle, SQL Server, MySQL and FireBird.

FlowChart - <http://www.patton-patton.com/news.htm>

Desenha gráficos em geral.

Power*Architect Data Modeling Tool - <http://www.sqlpower.ca/page/architect>

Data Architects, DBA's, Analysts and Designers rely on Data Modeling tools to facilitate and simplify their data Modeling efforts, while maximizing the use of their resources. The Power*Architect allows these busy highly technical resources to perform this most intricate part of their job in a fraction of the time.

The Power*Architect is a user-friendly data modeling tool created by data warehouse designers, and has many unique features geared specifically for the data warehouse architect. It allows users to reverse-engineer existing databases, perform data profiling on source databases, and auto-generate ETL metadata.

Plus, the Power*Architect has the ability to take snapshots of database structures, allowing users to design DW data models while working offline.

Whether you are building a Data Warehouse or using data models to communicate business rules, the Power*Architect will facilitate and automate your data modeling efforts.

Br-Modelo - <http://www.sis4.com/brModelo/download.aspx>

brModelo: Ferramenta freeware voltada para ensino de modelagem em banco de dados relacional com base na metodologia defendida por Carlos A. Heuser no livro "Projeto de Banco de Dados" (capa a baixo).

Esta ferramenta foi desenvolvida por Carlos Henrique Cândido sob a orientação do Prof. Dr. Ronaldo dos Santos Mello (UFSC), como trabalho de conclusão do curso de pós-graduação em banco de dados (UNVAG - MT e UFSC).

Monografia sobre a ferramenta -

<http://www.sis4.com/brModelo/monografia/monografia.htm>

Database Architect - <http://www.gurudevelopers.com/da/index.html>

Database Architect is designed for developers working on the database-enabled software projects. When working with Database Architect, software architect defines database tables, fields, references and indexes, drawing them directly in the program's main window and invokes the program to generate the database directly from the program window: Database Architect is compatible to any kind of relational databases that support SQL, including, but not limited to Oracle, Microsoft SQL Server, Microsoft Access, flat file databases, such as DBF or Paradox tables and any other ODBC-compatible databases. With Database Architect you'll never have to develop the database structure on the paper!

Database Designer for PostgreSQL -

<http://www.microolap.com/products/database/postgresql-designer/>

Database structure modeling, generation and modification focused on PostgreSQL Database Designer for PostgreSQL is an easy CASE tool with intuitive graphical interface allowing you to build a clear and effective database structure visually, see the complete picture (diagram) representing all the tables, references between them, views, stored procedures and other objects. Then you can easily generate a physical database on a server, modify it according to any changes you made to the diagram using fast ALTER statements.

Toad® Data Modeler - <http://www.quest.com/Toad-Data-Modeler/> ou

http://www.toadsoft.com/toaddm/toad_data_modeler.htm

Powerful and Cost-effective Data Modeling and Design

Quest Software's Toad® Data Modeler is a cost-effective, yet powerful data modeling and design tool that is built for the individual developer, DBA and data architect.

Toad Data Modeler makes it easier for you to: Toad for Oracle for enhanced efficiency.

Build complex entity relationship models (both logical and physical)

Synchronize models

Generate complex SQL/DDL

Create "ALTER" scripts (Oracle only)
Reverse engineer legacy databases
Toad Data Modeler also integrates with
Toad Data Modeler supports Oracle, SQL Server, Sybase ASE, MySQL, PostgreSQL, and DB2 LUW versions 8 and 9.

Visual Case - <http://www.visualcase.com/>

Visual Case - UML & E/R Database Design Tool
analysis, modeling, and design DNP

Outras

DBI-Link - <http://pgfoundry.org/projects/dbi-link/>

Is a partial implementation of the SQL/MED (Management of External Data) portion of the SQL:2003 specification. You can add speed and accuracy to your ETL processes by treating any data source you can reach with DBI as a PostgreSQL table.

Efetua conexões para bancos de outros SGBDs.

Documentação - http://pgfoundry.org/docman/?group_id=1000045

DBLink - <http://www.postgresql.org/docs/8.3/static/contrib-dblink.html>

Efetua consultas em outros bancos, inclusive remotos (mas somente do PostgreSQL).

Veja o tópico sobre ele no material em:

http://pg.ribafs.net/down/admin//Modulo2Aula13_Contribs.pdf

PgBouncer - <http://pgfoundry.org/projects/pgbouncer>

Lightweight connection pooler for PostgreSQL.

PGLoader - <http://pgfoundry.org/projects/pgloader/>

The PostgreSQL Loader project is a fast data loader for PostgreSQL, with the ability to generate files of rejected rows. It currently requires Python and Psycopg

PGPool - <http://pgpool.projects.postgresql.org/>

pgpool-II is a middleware that works between PostgreSQL servers and a PostgreSQL database client. It provides the following features.

* Limiting Exceeding Connections

There is a limit on the maximum number of concurrent connections with PostgreSQL, and connections are rejected after this many connections. Setting the maximum number of connections, however, increases resource consumption and affect system performance. pgpool-II also has a limit on the maximum number of connections, but extra connections will be queued instead of returning an error immediately.

* Connection Pooling

pgpool-II saves connections to the PostgreSQL servers, and reuse them whenever a new connection with the same properties (i.e. username, database, protocol version) comes in. It reduces connection overhead, and improves system's overall throughput.

* Replication

pgpool-II can manage multiple PostgreSQL servers. Using the replication function enables creating a realtime backup on 2 or more physical disks, so that the service can continue without stopping servers in case of a disk failure.

* Load Balance

If a database is replicated, executing a SELECT query on any server will return the same result. pgpool-II takes an advantage of the replication feature to reduce the load on each PostgreSQL server by distributing SELECT queries among multiple servers, improving system's overall throughput. At best, performance improves proportionally to the number of PostgreSQL servers. Load balance works best in a situation where there are a lot of users executing many queries at the same time.

* Parallel Query

Using the parallel query function, data can be divided among the multiple servers, so that a query can be executed on all the servers concurrently to reduce the overall execution time. Parallel query works the best when searching large-scale data.

pgpool-II talks PostgreSQL's backend and frontend protocol, and relays a connection between them. Therefore, a database application (frontend) thinks that pgpool-II is the actual PostgreSQL server, and the server (backend) sees pgpool-II as one of its clients. Because pgpool-II is transparent to both the server and the client, an existing database application can be used with pgpool-II almost without a change to its sources.

PgWorksheet - <http://pgworksheet.projects.postgresql.org/>

Is a simple GUI frontend to PostgreSQL for executing SQL queries and psql commands.

PLJava - <http://pgfoundry.org/projects/pljava/>

PL/Java is a free add-on module that brings Java tm Stored Procedures, Triggers, and Functions to the PostgreSQL tm backend via the standard JDBC interface.

PLProxy - <http://pgfoundry.org/projects/plproxy/>

PL/Proxy is database partitioning system implemented as PL language.

OpenFTS - <http://openfts.sourceforge.net/>

OpenFTS (Open Source Full Text Search engine) is an advanced PostgreSQL-based search engine that provides online indexing of data and relevance ranking for database searching. Close integration with database allows use of metadata to restrict search results.

Mondrian - <http://mondrian.sourceforge.net/>

Mondrian is an OLAP server written in Java. It enables you to interactively analyze very large datasets stored in SQL databases without writing SQL.

Sequoia - <http://sequoia.continuent.org/HomePage>

Sequoia is a transparent middleware solution offering clustering, load balancing and failover services for any database. The database is distributed and replicated over multiple nodes and Sequoia balances the queries between them. Sequoia handles node and network failures transparently. It also provides support for hot recovery, online maintenance operations and online upgrades.

check_postgres - http://bucardo.org/check_postgres/

check_postgres.pl is a script for checking the state of one or more Postgres databases and reporting back in a Nagios-friendly manner. It was developed by Greg Sabino Mullane of End Point Corporation and is BSD-licensed.

VirtualBox - <http://www.virtualbox.org>

Virtualização fácil de instalar e de usar para vários SOs.

VirtualBox is a family of powerful x86 virtualization products for enterprise as well as home use. Not only is VirtualBox an extremely feature rich, high performance product for enterprise customers, it is also the only professional solution that is freely available as Open Source Software under the terms of the GNU General Public License (GPL). See "About VirtualBox" for an introduction.

AndLinux - <http://www.andlinux.org>

Rodar o Linux (Ubuntu) nativamente dentro do Windows. O terminal, o gerenciador de arquivos, o synaptic e outros rodam no windows, como se fossem aplicativos do próprio Windows.

InstMSiA - <http://www.microsoft.com/downloads/details.aspx?familyid=cebbacd8-c094-4255-b702-de3bb768148f&displaylang=en>

Permite executar arquivos .msi sem tem o Windows Installer.

Criar novo Serviços no Windows - <http://support.microsoft.com/kb/137890>

Pentaho - <http://www.pentaho.com/>

Comercial Opensource alternative to BI

pg_live - <https://www.pglivecd.org/>

Live CD com CentOS 7 e PostgreSQL 9.6 pré-instalado, juntamente com diversas ferramentas: pgadmin, phppgadmin, apache e php, várias linguagens de procedimentos, slony, documentação, etc.

Documentação: <http://205.237.195.102/docs/>

Portable GIS - <http://www.archaeogeek.com/blog/portable-gis/>

Que tal o PostgreSQL num pendrive, juntamente com diversas ferramentas para trabalhar com GIS?

Este é o propósito dessa ferramenta. São 452MB e aqui uma pequena explanação:

http://www.osgeo.org/files/journal/v3/en-us/final_pdfs/cook.pdf

Process Explorer - <http://technet.microsoft.com/en-us/sysinternals/bb896653.aspx>

Ever wondered which program has a particular file or directory open? Now you can find out. Process Explorer shows you information about which handles and DLLs processes have opened or loaded.

The Process Explorer display consists of two sub-windows. The top window always shows a list of the currently active processes, including the names of their owning accounts, whereas the information displayed in the bottom window depends on the mode that Process Explorer is in: if it is in handle mode you'll see the handles that the process selected in the top window has opened; if Process Explorer is in DLL mode you'll see the DLLs and memory-mapped files that the process has loaded. Process Explorer also has a powerful search capability that will quickly show you which processes have particular handles opened or DLLs loaded.

Xampp - <http://xampp.sf.net>

Pacote contendo instaladores para ambiente do PHP com MySQL e várias ferramentas. For Windows, Linux, Mac e Solaris.

XAMPP is an easy to install Apache distribution containing MySQL, PHP and Perl. XAMPP is really very easy to install and to use - just download, extract and start.

Win32Pad - <http://www.gena01.com/win32pad/>

Pequeno editor de textos for windows com boas funcionalidades:
numera linhas, lê bem arquivos do Linux, sobrescreve o notepad com vantagens.

Monitoração**ptop** - <http://pgfoundry.org/projects/ptop/>

'top' for PostgreSQL processes. See running queries, query plans, issued locks, and table and index statistics.

Table Log - <http://pgfoundry.org/projects/tablelog/>

PostgreSQL Table Log uses a trigger to log any INSERTs, UPDATEs and DELETEs on a specific table into another table. The second part of tablelog is able to restore the state of the original table or of a specific row for any time in the past.

pgFouine - <http://pgfoundry.org/projects/pgfouine/>

Is a PostgreSQL log analyzer written in PHP. It is based on PQA, the Practical Query Analyzer written in Ruby. pgFouine aims to be able to parse huge logs and to have a nice and useful HTML output.

PgAnalyzer - <http://ostatic.com/103858-software-opensource/pganalyzer>

PgAnalyzer is a perl script and library used to analyze the server logs files of the excellent PostgreSQL RDBMS. It reports practical statistics information on number of database connections, users and hosts. It can also provide SQL query and error statistics.

Playr - <https://area51.myyearbook.com/trac.cgi/wiki/Playr>

Playr attempts to answer the question "How much headroom will our new server give us?" It's a PostgreSQL log file replay application. It works by taking your PostgreSQL logs and running them through a conversion application which turns them into a binary format. The replay application then reads in the binary format and attempts to replay the logs in the same timing with the same backend assignments. By attempting to do this, Playr is stressing a PostgreSQL box with the same query frequency distributed across the same quantity of backends, in an attempt to truly stress PostgreSQL in the same way as it was originally stressed when the logs were taken.

Theoretically, using Playr in conjunction with sysstat and a graphing application like Staplr, one can analyze other hardware against production load.

Playr is not designed to work with lesser hardware than the machine where the log files were originally taken from. It is not designed to be a benchmark application in the traditional sense. If the new hardware can not keep up with the timing, Playr will give up its stress test and let you know that it fell behind.

pgtray - <http://pgfoundry.org/projects/pgtray/>

This is a simple Windows application, which stays in tray and allows to monitor (active/stopped) and manipulate (start/stop/restart) PostgreSQL servers, installed as NT Services in the system.

pgmonitor - <http://pgfoundry.org/projects/pgmonitor/>

This is a Tcl/Tk script that displays current database connections, and allows you to view the currently executing query, cancel the query, or terminate the session. You can also start/stop the postmaster.

pgTAP - <http://pgfoundry.org/projects/pgtap/>

Write TAP-based unit tests for your database! pgTAP is a suite of database functions that make it easy to write unit tests in psql scripts suitable for harvesting, analysis, and reporting by a TAP harness, such as those used in Perl and PHP applications.

Apresentação sobre o uso do pgTAP -

<http://justatheory.com/computers/databases/postgresql/pgtap-yapc.pdf>

Performance

pgsnap - <http://pgfoundry.org/projects/pgsnap/>

Is a PostgreSQL tool that mimics orasnap performance report tool for Oracle.

pebench - <http://www.postgresql.org/docs/8.3/static/pgbench.html>

pgbench is a simple program for running benchmark tests on PostgreSQL. It runs the same sequence of SQL commands over and over, possibly in multiple concurrent database sessions, and then calculates the average transaction rate (transactions per second). By default, pgbench tests a scenario that is loosely based on TPC-B, involving five SELECT, UPDATE, and INSERT commands per transaction. However, it is easy to test other cases by writing your own transaction script files.

Relatórios (Geradores)

BIRT - <http://www.eclipse.org/birt/phoenix/>

Gera relatórios do PostgreSQL e de todos os SGBDs que tenham suporte a JDBC. Fácil de usar e tem um visualizador web (Tomcat) para que seus relatórios possam ser acessados por qualquer aplicação web.

iReport - <http://www.jasperforge.org/sf/projects/ireport>

Semelhante ao BIRT, sendo que seus relatórios para serem exibidos via Web dão um pouco mais de trabalho, já que a grande compatibilidade é com aplicativos Java.

Crystal Report -

<http://www.businessobjects.com/product/catalog/crystalreports/default.asp>

Gerador de relatórios com muitos recursos.

Tutorial: http://wiki.postgresql.org/wiki/PostgreSQL_and_Crystal_Reports

OpenRPT - <http://www.openmfg.com/openrpt>

OpenRPT: xTuple's open source SQL report writer

As part of all three xTuple ERP Editions, built with the PostgreSQL database and the Qt GUI client framework, we built our own SQL report writer from scratch. Like the ERP, it runs equally well on Windows, Linux, and Mac OS X. We call it OpenRPT.

OpenRPT is released under the GNU Lesser General Public License. You are free to use it as you wish; if you would like to purchase commercial support or embedded licenses, please see below.

Pentaho Report Design - <http://reporting.pentaho.org/>

Pentaho Reporting is a collection of open source projects primarily focused on the creation, generation and distribution of rich and sophisticated report content from all sources of information.

Replicação

Slony - <http://slony.info/>

Slony-I is a "master to multiple slaves" replication system supporting cascading (e.g. - a node can feed another node which feeds another node...) and failover.

The big picture for the development of Slony-I is that it is a master-slave replication system that includes all features and capabilities needed to replicate large databases to a reasonably limited number of slave systems.

Slony-I is a system designed for use at data centers and backup sites, where the normal mode of operation is that all nodes are available.

A fairly extensive "admin guide" comprising material in the CVS tree may be found here.

There is also a local copy.

The original design document is available here.

Tutorial: <http://ribafs.wordpress.com/2008/04/23/replicacao-com-slony-no-windows-e-no-linux/>

Bucardo - <http://bucardo.org/>

Bucardo is an asynchronous PostgreSQL replication system, allowing for both multi-master and multi-slave operations. It was developed at Backcountry.com primarily by Greg Sabino Mullane of End Point Corporation.

PGCluster - <http://www.pgcluster.org/>

PGCluster is a multi master, synchronous replication system based on PostgreSQL Database Server.

- * "Synchronous Replication System" - No delay occurs for data duplication between the Cluster DBs.

- * "Multi-master Cluster DB System" - There is no Cluster DBs preference for queries. A user can use any node for any type of query

Segurança

Backtrack - <http://www.remote-exploit.org/backtrack.html>

Live CD do Linux voltado para segurança de sistemas.

BackTrack is the most Top rated linux live distribution focused on penetration testing. With no installation whatsoever, the analysis platform is started directly from the CD-Rom and is fully accessible within minutes.

Artigo em português: <http://planeta.ubuntubrasil.org/post/3081>

XML/DocBook

Alchemist XML IDE - <http://cleansofts.org/alchemist-xml-ide-professional-free-edition.html>

Is an advanced XML Integrated Development Environment.

Alchemist XML IDE, is a free advanced XML Integrated Development Environment (XML IDE). Alchemist adds powerful new features, again pushing the innovation envelope that helped establish Alchemist in the market. Alchemist's best-in-class features for working with XML, XSL, XSLT, XPath, SQL/XML, code generation, database to XML mapping and many other XML technologies.

Free XML Editor - EditiX Lite Version - <http://free.editix.com/>

- XSLT Debugger
- Visual Schema Editor
- UniCode
- XPath 1.0, 2.0
- XSL-FO
- DocBook
- OASIS Catalog
- Refactoring
- Project management

Free e comercial, for Windows e Linux

Catálogo de Ferramentas no site oficial do PostgreSQL:

<http://www.postgresql.org/download/product-categories>

Lista de Ferramentas no site do PostgreSQL Brasil:

http://www.postgresql.org.br/Ferramentas_para_o_PostgreSQL

Boa lista de ferramentas, não somente para PostgreSQL:

http://imasters.uol.com.br/artigo/4177/sql_server/softwares_livres_relacionados_a_banco_de_dados

Algumas Ferramentas e Tutoriais

<http://pg.ribafs.net/down/ferramentas>

Expressões Regulares para uso em Modelagem de Bancos de Dados

As expressões regulares são um grande recurso para ajudar a garantir a integridade das informações, em especial no uso com domínios.

Em ciência da computação, uma expressão regular (ou o estrangeirismo regex, abreviação do inglês regular expression) provê uma forma concisa e flexível de identificar cadeias de caracteres de interesse, como caracteres particulares, palavras ou padrões de caracteres. Expressões regulares são escritas numa linguagem formal que pode ser interpretada por um processador de expressão regular, um programa que ou serve um gerador de analisador sintático ou examina o texto e identifica partes que casam com a especificação dada.

O termo deriva do trabalho do matemático norte-americano Stephen Cole Kleene, que desenvolveu as expressões regulares como uma notação ao que ele chamava de álgebra de conjuntos regulares. Seu trabalho serviu de base para os primeiros algoritmos computacionais de busca, e depois para algumas das mais antigas ferramentas de tratamento de texto da plataforma Unix.

O uso atual de expressões regulares inclui procura e substituição de texto em editores de texto e linguagens de programação, validação de formatos de texto (validação de protocolos ou formatos digitais), realce de sintaxe e filtragem de informação.
(Wikipedia - http://pt.wikipedia.org/wiki/Express%C3%B5es_regulares)

Correspondência com o Padrão e Expressões Regulares no PostgreSQL

O PostgreSQL suporta várias formas de correspondência com o padrão (pattern matching): o tradicional operador SQL LIKE, o mais recente operador SIMILAR TO (adicionado no SQL 1999) e as expressões regulares estilo POSIX (também implementada na função substring).

As regex são um recurso muito útil aos DBAs. As expressões regulares oferecem força e agilidade.

LIKE

LIKE - case sensitive

ILIKE - case insensitive

Caracteres Coringa:

% - 0 ou mais caracteres
 _ - 1 único caractere

NOT LIKE
 ~~ = LIKE
 ~~* = ILIKE
 !~~ = NOT LIKE
 !~~* = NOT ILIKE

SIMILAR TO

Semelhante ao LIKE mas usa expressões regulares do SQL.
 Assim como o LIKE somente é válido quando toda a string corresponde ao padrão.

Em expressões regulares qualquer parte da string pode corresponder ao padrão.

Caracteres coringa:
 % - 0 ou mais caracteres
 _ - 1 único caractere

Adiciona:

| - ou
 * - 0 ou mais vezes
 + - 1 ou mais vezes
 () - agrupar itens
 [] - similar ao POSIX

POSIX

~ - case sensitive e corresponde
 ~* - case insensitive e corresponde
 !~ - case sensitive e não corresponde
 !~* - case insensitive e não corresponde

Átomos do Padrão

(er) - expressão regular. Correspondência para a er
 [caracteres] - corresponde a qualquer dos caracteres

\k - caractere não alfanumérico

\c - caractere alfanumérico

. - qualquer único caractere

x - este é o único caractere sem função, representa 'x' mesmo

[:alnum:] Caracteres alfanuméricos, o que no caso de ASCII corresponde a [A-Za-z0-9].

[:alpha:] Caracteres alfabeticos, o que no caso de ASCII corresponde a [A-Za-z].

[:blank:] Espaço e tabulação, o que no caso de ASCII corresponde a [\t].

[:cntrl:] Caracteres de controle, o que no caso de ASCII corresponde a [\x00-\x1F\x7F].

[:digit:] Dígitos, o que no caso de ASCII corresponde a [0-9]. O Perl oferece o atalho \d.

[:graph:] Caracteres visíveis, o que no caso de ASCII corresponde a [\x21-\x7E].

[:lower:] Caracteres em caixa baixa, o que no caso de ASCII corresponde a [a-z].

[:print:] Caracteres visíveis e espaços, o que no caso de ASCII corresponde a [\x20-\x7E].

[:punct:] Caracteres de pontuação, o que no caso de ASCII corresponde a [-!"#\$

%&(')*+,./;=>?@[\\"\\]_`{|}~].

[:space:] Caracteres de espaços em branco, o que no caso de ASCII corresponde a [\t\r\n\v\f]. O Perl oferece o atalho \s, que, entretanto, não é exatamente equivalente; diferente do \s, a classe ainda inclui um tabulador vertical, \x11 do ASCII.[4]

[:upper:] Caracteres em caixa alta, o que no caso de ASCII corresponde a [A-Z].

[:xdigit:] Dígitos hexadecimais, o que no caso de ASCII corresponde a [A-Fa-f0-9].

Expressões regulares não podem terminar com \.

Quantificadores dos caracteres do Padrão:

* - uma seqüência de 0 ou mais correspondências do átomo

+ - uma seqüência de 1 ou mais correspondências do átomo

? - uma seqüência de 0 ou 1 correspondência do átomo

{m} - uma seqüência de exatamente m correspondências do átomo

{m,} - uma seqüência de m ou mais correspondências do átomo

{m,n} - uma seqüência de m a n (inclusive) correspondências do átomo; m não pode ser maior do que n

Caracteres Delimitadores do Padrão

^ - início da string

\$ - final da string

Alguns exemplos:

```
SELECT regexp_matches('foobarbequebazilbarfbonk', '(b[^b]+)(b[^b]+)', 'g');
regexp_split_to_table function splits a string using a POSIX regular expression pattern as a delimiter.
```

The `regexp_split_to_array` function behaves the same as `regexp_split_to_table`, except that `regexp_split_to_array` returns its result as an array of text. It has the syntax `regexp_split_to_array(string, pattern [, flags])`. The parameters are the same as for `regexp_split_to_table`.

```
SELECT foo FROM regexp_split_to_table('the quick brown fox jumped over the lazy dog',
E'\\s+') AS foo;
foo
-----
the
quick
brown
fox
jumped
over
the
lazy
dog
(9 rows)
```

```
SELECT regexp_split_to_array('the quick brown fox jumped over the lazy dog', E'\\s+');
      regexp_split_to_array
-----
{the,quick,brown,fox,jumped,over,the,lazy,dog}
(1 row)
```

```
SELECT foo FROM regexp_split_to_table('the quick brown fox', E'\\s*') AS foo;
```

Example of how to escape "_" in a simple query

```
create table foo (str varchar(16));
insert into foo (str) values ('abc.defghi');
insert into foo (str) values ('abc_defghi');
```

I want to select all strings starting with abc_def

```
select * from foo where str like E'abc\\_def%';
```

```

> Fernando Brombatti wrote:
>> Alguém já usou função para extrair números de uma string?
>>
>> Ex.: AB345CD234 => 345234
>
>
> lista=# select regexp_replace('AB345CD234', '[A-Z]', ','g');
>   regexp_replace
> -----
> 345234
> (1 row)
>
```

Dica na lista pgbe-geral:

Complementando a resposta do Shander:

Caso sua string possa conter outros caracteres não numéricos, além das letras [A-Z], o uso de '[^[:digit:]]' é mais abrangente.

<http://www.postgresql.org/docs/current/interactive/functions-matching.html#FUNCTIONS-POSIX-REGEXP>

```

bdteste=# SELECT regexp_replace('AB3,45CD/xz234', '[^[:digit:]]', '',
'g');
regexp_replace
-----
345234
```

Osvaldo

Bom tutorial no site:

http://www.oreillynet.com/pub/a/databases/2006/02/02/postgresq_regexes.html

Reprozindo aqui os exemplos do tutorial.

Vamos criar uma tabela:

```
CREATE DATABASE regex;
```

```
CREATE TABLE myrecords(record text);
```

Insira os registros:

```
insert into myrecords (record) values
```

```
('a'),
('ab'),
('abc'),
('123abc'),
('132abc'),
('123ABC'),
('abc123'),
('4567'),
('5678'),
('6789');
```

Consultas simples usam o operador ~ (til) seguido por uma string e retornam somente os que atendem ao case.

`SELECT record FROM myrecords WHERE record ~ '1';` -- Retornaram todos os registros que contenham '1'.

```
SELECT record FROM myrecords WHERE record ~ 'a';
SELECT record FROM myrecords WHERE record ~ 'A';
SELECT record FROM myrecords WHERE record ~ '3a';
```

Para retornar sem olhar o case usamos ~*:

```
SELECT record FROM myrecords WHERE record ~* 'a';
SELECT record FROM myrecords WHERE record ~* '3a';
```

Agora não trazendo o que contém a string e sensível ao case !~:

`SELECT record FROM myrecords WHERE record !~ '1';`

Trazendo sem olhar o case e não trazendo onde tem a string !~*:

`SELECT record FROM myrecords WHERE record !~* 'c';`

Trazendo as strings que comecem com um certo caractere (^):

```
SELECT record FROM myrecords WHERE record ~ '^1';
SELECT record FROM myrecords WHERE record ~ '^a';
SELECT record FROM myrecords WHERE record ~* '^a';
```

Terminados com (\$):

```
SELECT record FROM myrecords WHERE record ~ 'c$';
SELECT record FROM myrecords WHERE record ~ 'bc$';
SELECT record FROM myrecords WHERE record ~* 'bc$';
```

Veja agora algumas consultas e analise seus resultados:

```
SELECT record FROM myrecords WHERE record ~ '[a]'; -- Qualquer que tenha a
SELECT record FROM myrecords WHERE record ~ '[A]'; -- Qualquer que tenha A
```

SELECT record FROM myrecords WHERE record ~* '[a]'; -- Qualquer que tenha a ou A
SELECT record FROM myrecords WHERE record ~ '[ac]'; -- Qualquer que tenha a ou c
SELECT record FROM myrecords WHERE record ~ '[ac7]'; -- Qualquer que tenha a ou c ou 7

SELECT record FROM myrecords WHERE record ~ '[a7A]'; -- Qualquer que tenha a ou 7 ou A

SELECT record FROM myrecords WHERE record ~* '[ac7]'; -- Qualquer que tenha a ou c ou 7 sem olhar o case

SELECT record FROM myrecords WHERE record ~ '[z]';
SELECT record FROM myrecords WHERE record ~ '[z7]';

SELECT record FROM myrecords WHERE record !~ '[4a]';

Procurar por uma faixa de valores:

SELECT record FROM myrecords WHERE record ~ '[1-4]';

Outros de faixa:

SELECT record FROM myrecords WHERE record ~ '[a-c5]';
SELECT record FROM myrecords WHERE record ~* '[a-c5]';
SELECT record FROM myrecords WHERE record ~ '[a-cA-C5-7]'; -- 3 faixas, a-c, A-C e 5-7

Correspondendo 2 ou mais caracteres:

SELECT record FROM myrecords WHERE record ~ '3[a]';
SELECT record FROM myrecords WHERE record ~ '[3][a]';
SELECT record FROM myrecords WHERE record ~ '[1-3]3[a]';

SELECT record FROM myrecords WHERE record ~ '[23][a]';
SELECT record FROM myrecords WHERE record ~ '[2-3][a]';
SELECT record FROM myrecords WHERE record ~ '[a-b][b-c]';

Nesta ordem:

SELECT record FROM myrecords WHERE record ~ '[a][c]';

Iniciando com dígitos:

SELECT record FROM myrecords WHERE record ~ '^[0-9]\$';

Fazendo escolhas (ou |):

SELECT record FROM myrecords WHERE record ~ '^a|c\$';

Começando com a ou 5 ou terminando com c:

```
SELECT record FROM myrecords WHERE record ~ '^a|c$|^5';
```

```
SELECT record FROM myrecords WHERE record ~ '[^0-9|^a-z]';
```

Repetindo Caracteres:

```
SELECT record FROM myrecords WHERE record ~ 'a*'; -- 0 ou mais
```

```
SELECT record FROM myrecords WHERE record ~ 'b+'; -- 1 ou mais
```

```
SELECT record FROM myrecords WHERE record ~ 'a?'; -- 0 ou 1
```

```
SELECT record FROM myrecords WHERE record ~ '[0-9]{3}'; -- Exatamente uma  
quantidade, usar {#}
```

```
SELECT record FROM myrecords WHERE record ~ '[0-9]{4,}'; -- Exatamente ou mais,  
usar {#,}
```

```
SELECT record FROM myrecords WHERE record ~ '[a-zA-Z0-9]{2,3}'; -- Exatamente de 2 até  
3, inclusive
```

Exemplos com a função Substring:

```
CREATE TABLE log(record text);
```

Inserir registros:

```
insert into log (record) values  
('a'),  
('ab'),  
('abc'),  
('123abc'),  
('132abc'),  
('123ABC'),  
('abc123'),  
('4567'),  
('5678'),  
('6789');
```

```
SELECT substring(record, '[a-zA-Z0-9.. ]{1,}') FROM log LIMIT 1;
```

```
SELECT date(substring(record, '[a-zA-Z ]{1,}[0-9]{1,}') || ' 2005') AS "Date" FROM log  
LIMIT 1;
```

```
SELECT substring('Nov 3 07:37:51 localhost', '[:0-9]{2,}') AS "Time";  
  
SELECT substring('Nov 30 07:37:51 localhost', '[:0-9]{2,}') AS "Time";  
  
SELECT substring('Nov 30 07:37:51 localhost', '[:0-9]{3,}') AS "Time";  
  
SELECT substring(record, '[:0-9]{3,}') AS "Time" FROM log LIMIT 1;  
  
SELECT substring(record, 'SRC=*( [.0-9]{2,})') AS "IP Address" FROM log LIMIT 1;  
  
SELECT substring(record, 'SPT=*( [.0-9]{2,})') AS "Remote Source Port"  
      FROM log LIMIT 1;  
SELECT substring(record, 'DPT=*( [.0-9]{2,})') AS "Destination Port"  
      FROM log LIMIT 1;
```

O SQL completo:

```
SELECT  
    date(substring(record, '[a-zA-Z ]{1,}[0-9]{1,}') || ' 2005') AS "Date",  
    substring(record, '[:0-9]{3,}') AS "Time",  
    substring(record, 'SRC=*( [.0-9]{2,})') AS "Remote IP Address",  
    substring(record, 'SPT=*( [.0-9]{2,})') AS "Remote Source Port",  
    substring(record, 'DPT=*( [.0-9]{2,})') AS "Destination Port"  
FROM log;
```

Mais detalhes no tutorial e na documentação do PostgreSQL:

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

Ferramentas Para Windows

<https://docs.microsoft.com/pt-br/sysinternals/>
<https://docs.microsoft.com/pt-br/sysinternals/downloads/>

Ferramentas Avançadas para Administradores Windows

Comandos incluídos no pacote:

```
AccessChk AccessEnum AdExplorer AdInsight AdRestore
Autologon Autoruns BgInfo BlueScreen CacheSet
ClockRes Contig Coreinfo Ctrl2Cap DebugView
Desktops Disk2vhd DiskExt DiskMon DiskView
Disk Usage (DU) EFSDump FileMon FindLinks Handle
Hex2dec Junction LMDDump ListDLLs LiveKd
LoadOrder LogonSessions MoveFile NewSid NotMyFault
NTFSInfo PageDefrag PendMoves PipeList PortMon
ProcDump Process Explorer ProcFeatures Process Monitor PsExec
PsFile PsGetSid PsInfo PsKill PsList
PsLoggedOn PsLogList PsPasswd PsPing PsService
PsShutdown PsSuspend PsTools RAMMap RegDelNull
RegHide RegJump RegMon Rootkit Revealer Registry Usage (RU)
SDelete ShareEnum ShellRunas Sigcheck Streams
Strings Sync Sysmon TCPView VMMMap
VolumenID Whols WinObj ZoomIt
```

6 - Instalação do PostgreSQL 8.3 no Windows

Download

Fazer o download de - <https://www.enterprisedb.com/downloads/postgres-postgresql-downloads>

Caso esteja baixando outra versão deverá acessar:

<http://www.postgresql.org/ftp>

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)

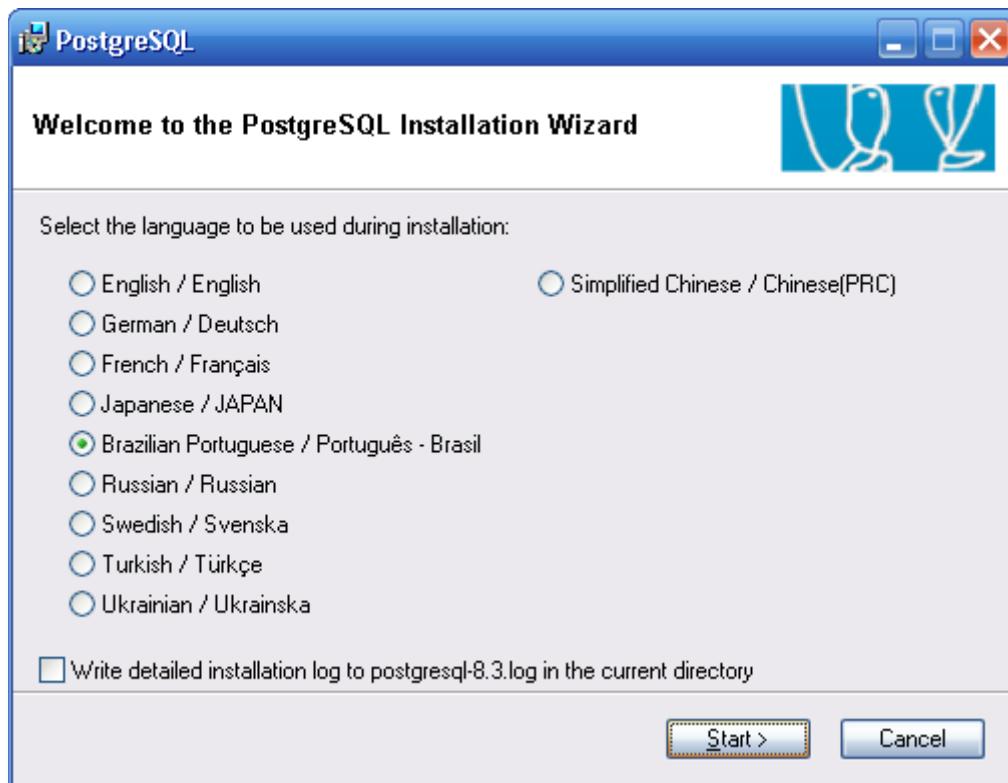
Descompactar e Instalar

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

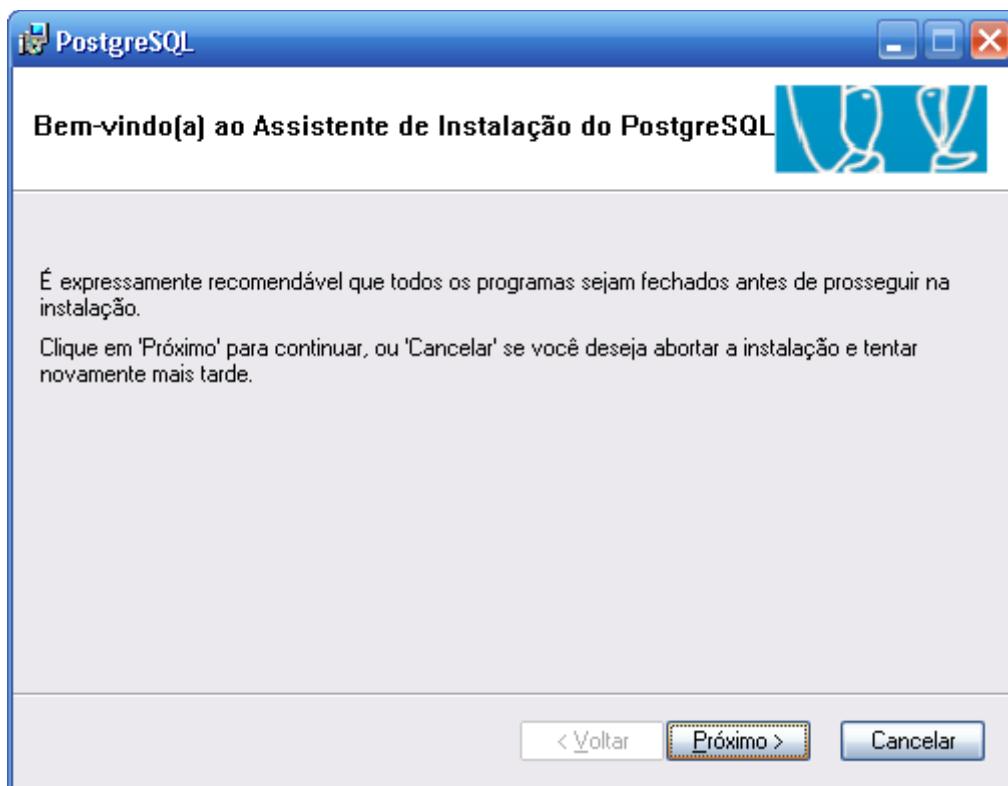
O arquivo UPGRADE.bat é para efetuar uma atualização de versão existente.

O arquivo postgresql-8.3.msi é que deve ser executado para a instalação.

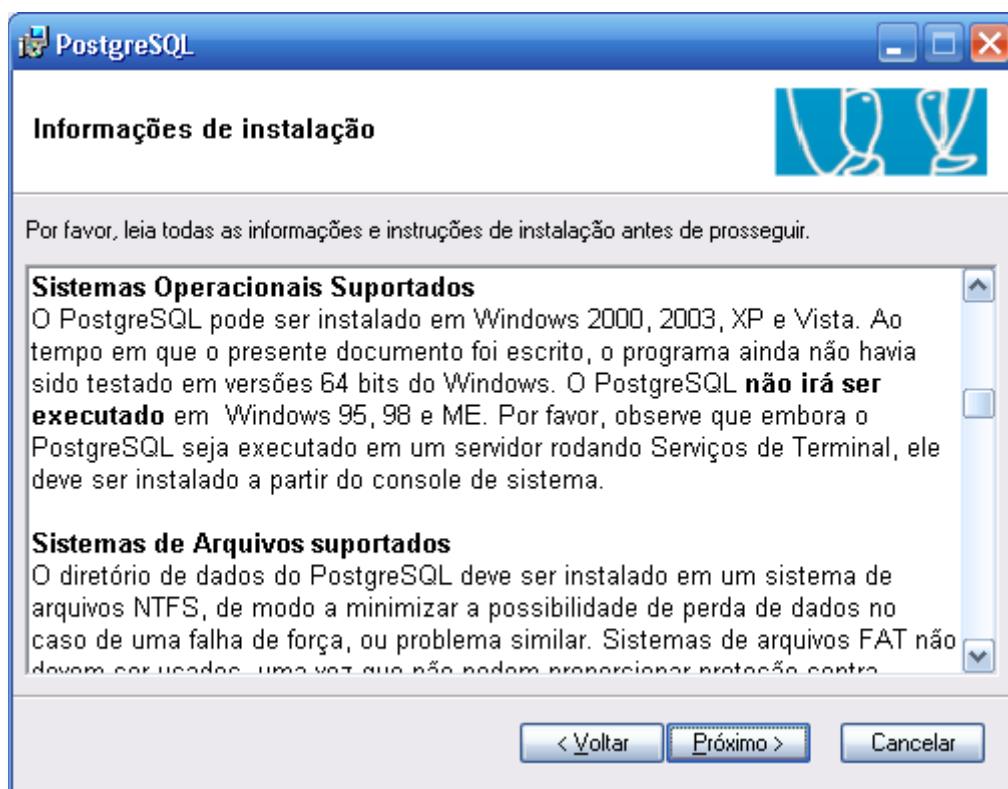
Execute o arquivo postgresql-8.3.msi:



Clique em Start.

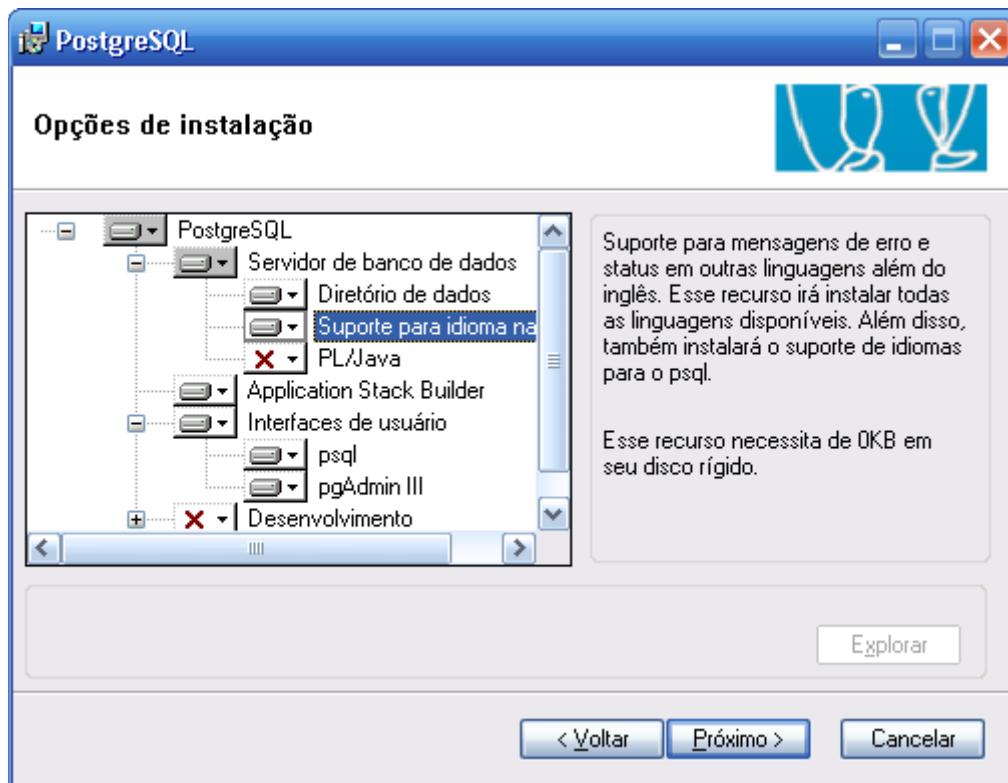


Desta tela em diante a instalação estará em português do Brasil.
Feche todos os programas abertos e clique em próximo.



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.

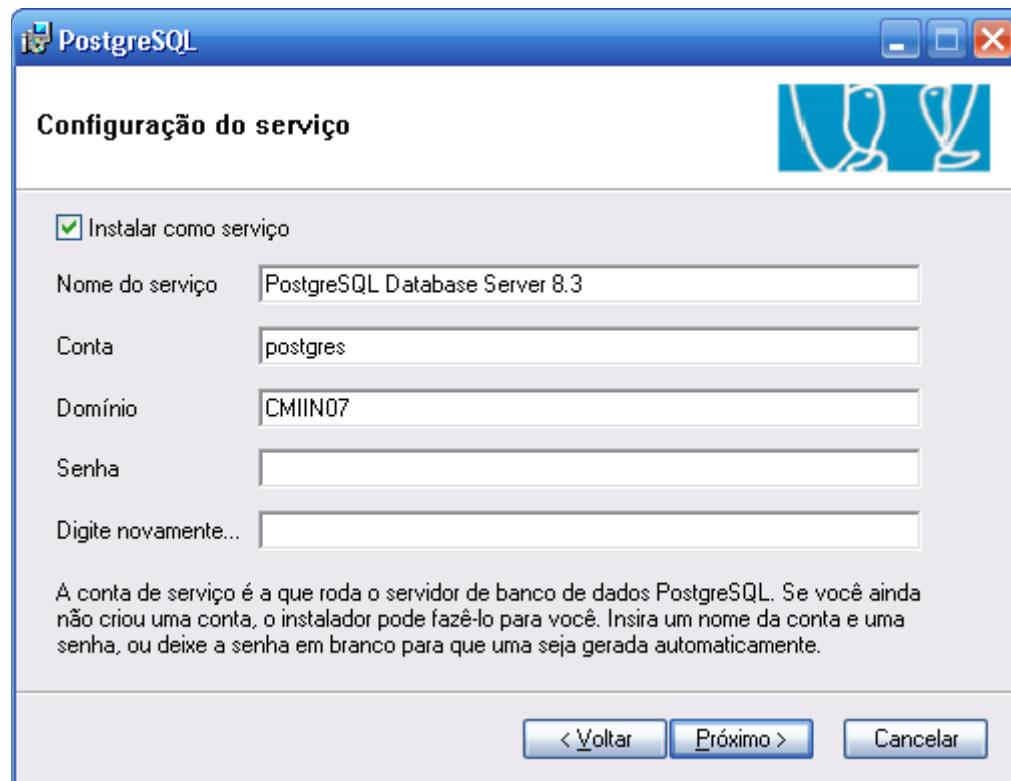
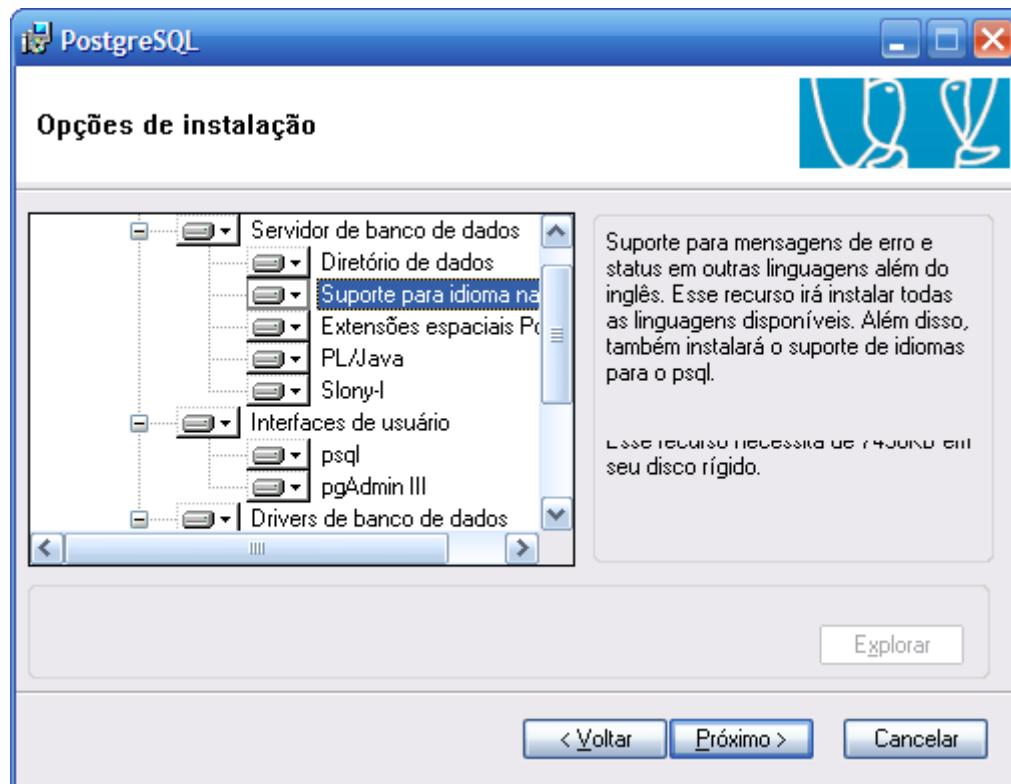
Clique em Próximo.



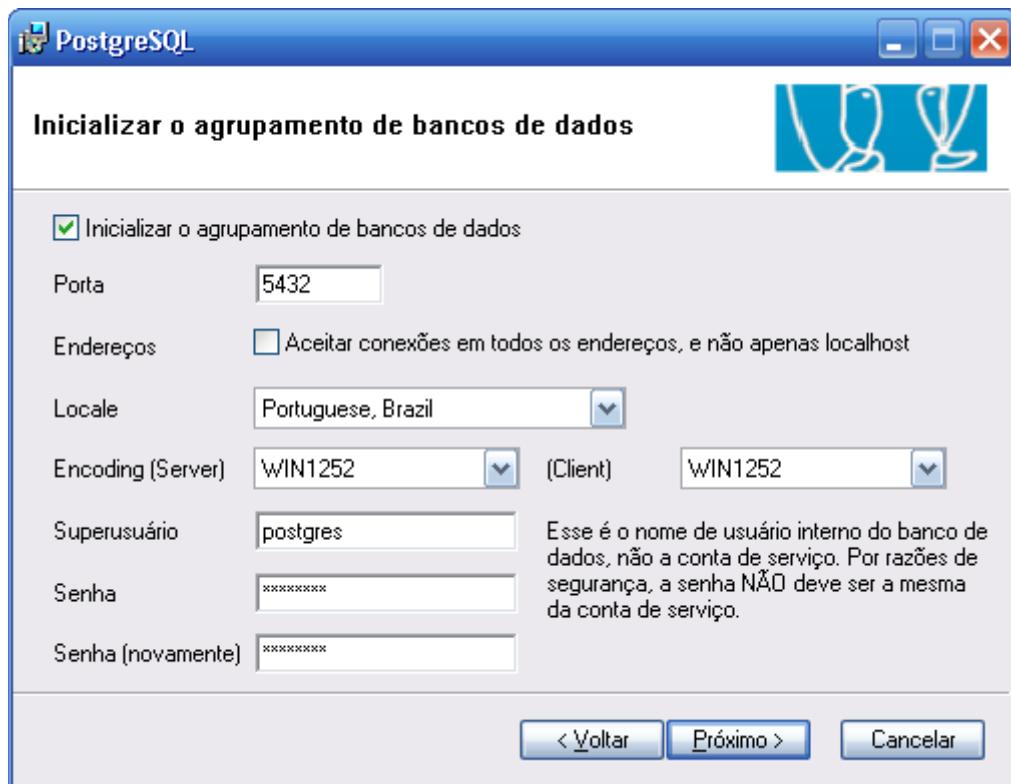
Nesta tela devemos ativar o Suporte para idiomas nativos, se quizermos ver mensagens de erro e o psql em português.

Clique em Próximo.

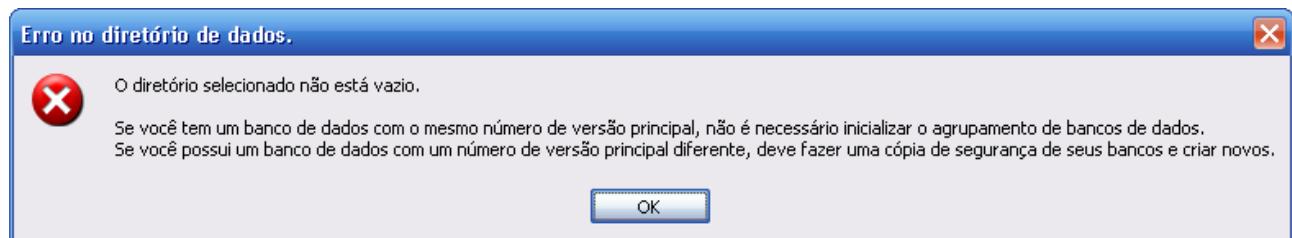
Veja a tela correspondente da versão 8.2.7 na próxima página



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

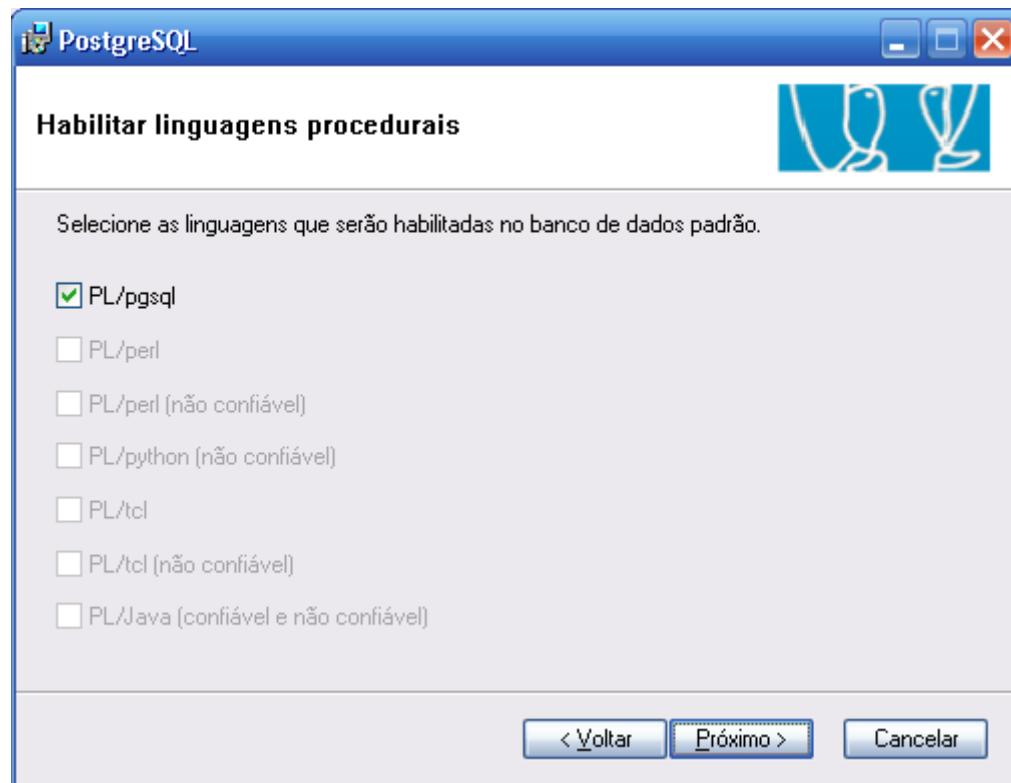


Caso esteja reinstalando o PostgreSQL, provavelmente receberá este importante aviso:



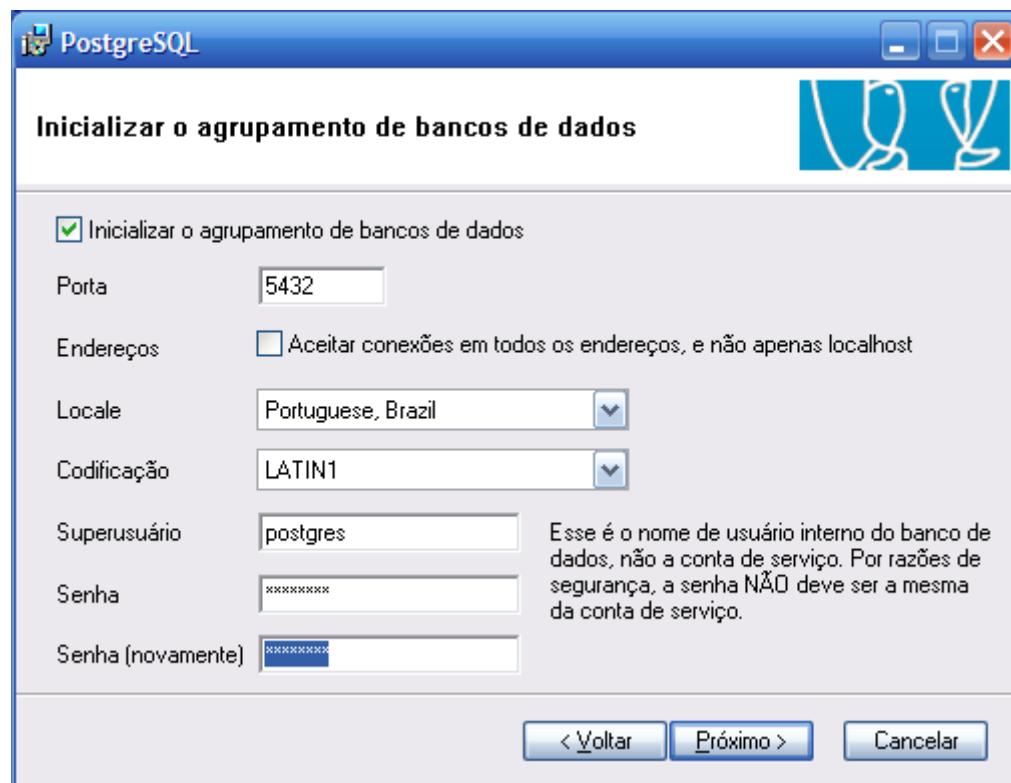
Na tela abaixo deixemos o Encoding do servidor e do cliente como Win1252, que são o padrão do Windows. Entre também com a senha do super usuário e clique em Próximo. O Locale deve ser de acordo com o sistema operacional.

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.

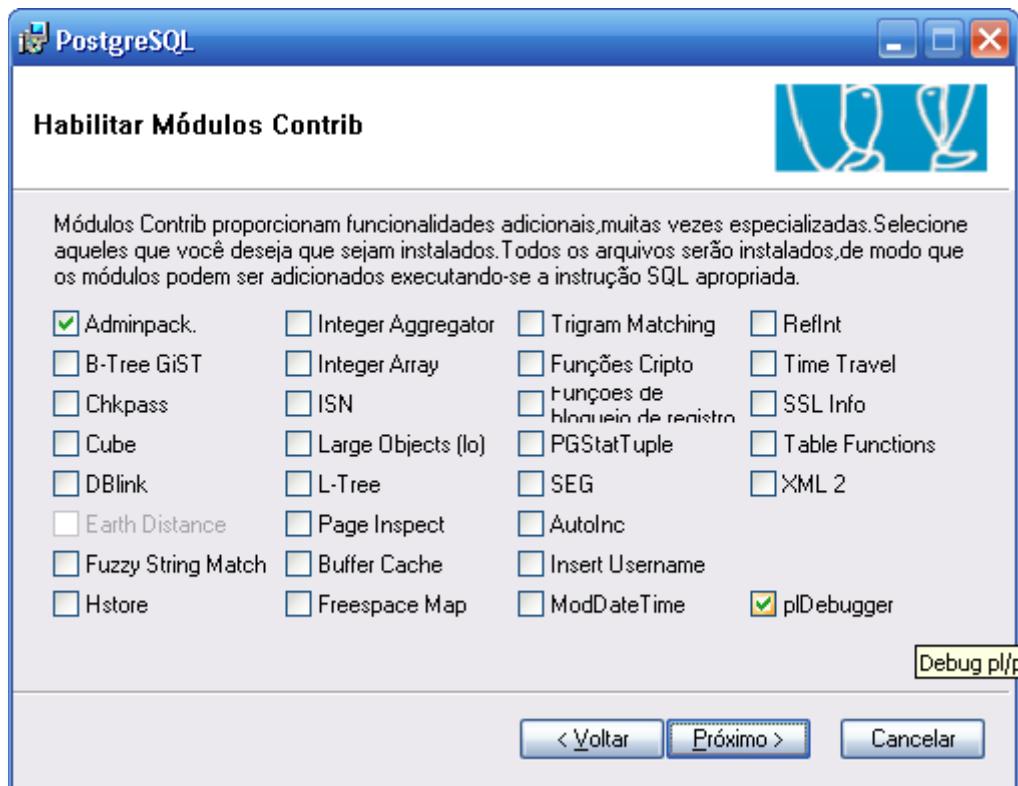


Nesta tela desmarque o item PL/pgsql e lembre de ativar o suporte a plpgsql apenas nos bancos em que for usar essa linguagem. Assim terá os bancos mais limpos.

A tela correspondente da versão 8.2.7



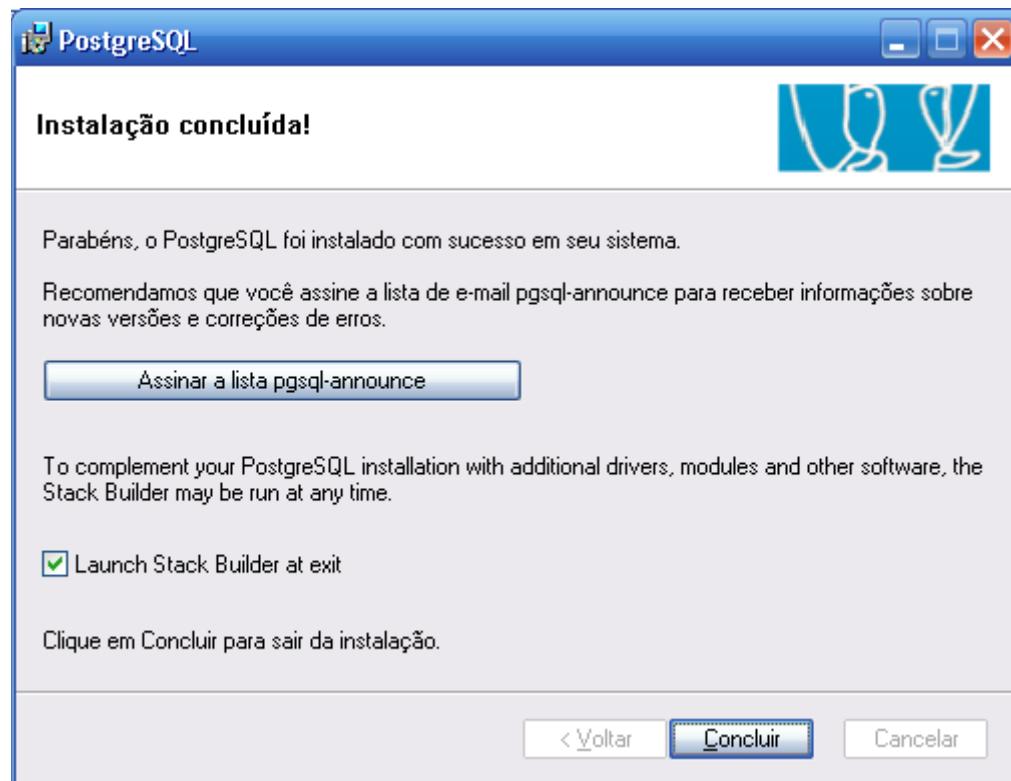
Clique em Próximo.



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..

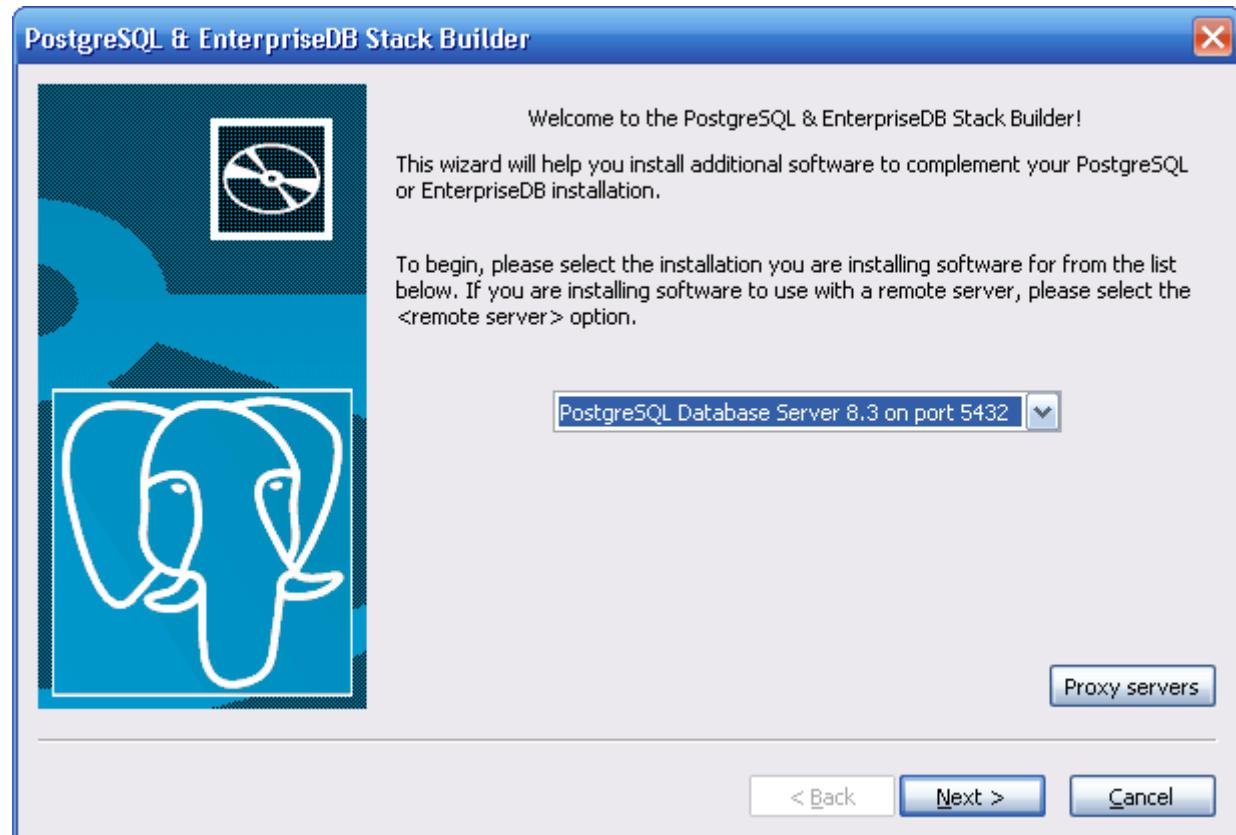
Caso tenha um bom firewall instalado verá duas solicitações de uso das portas 5432 e da 2427.

Aceite o acesso.



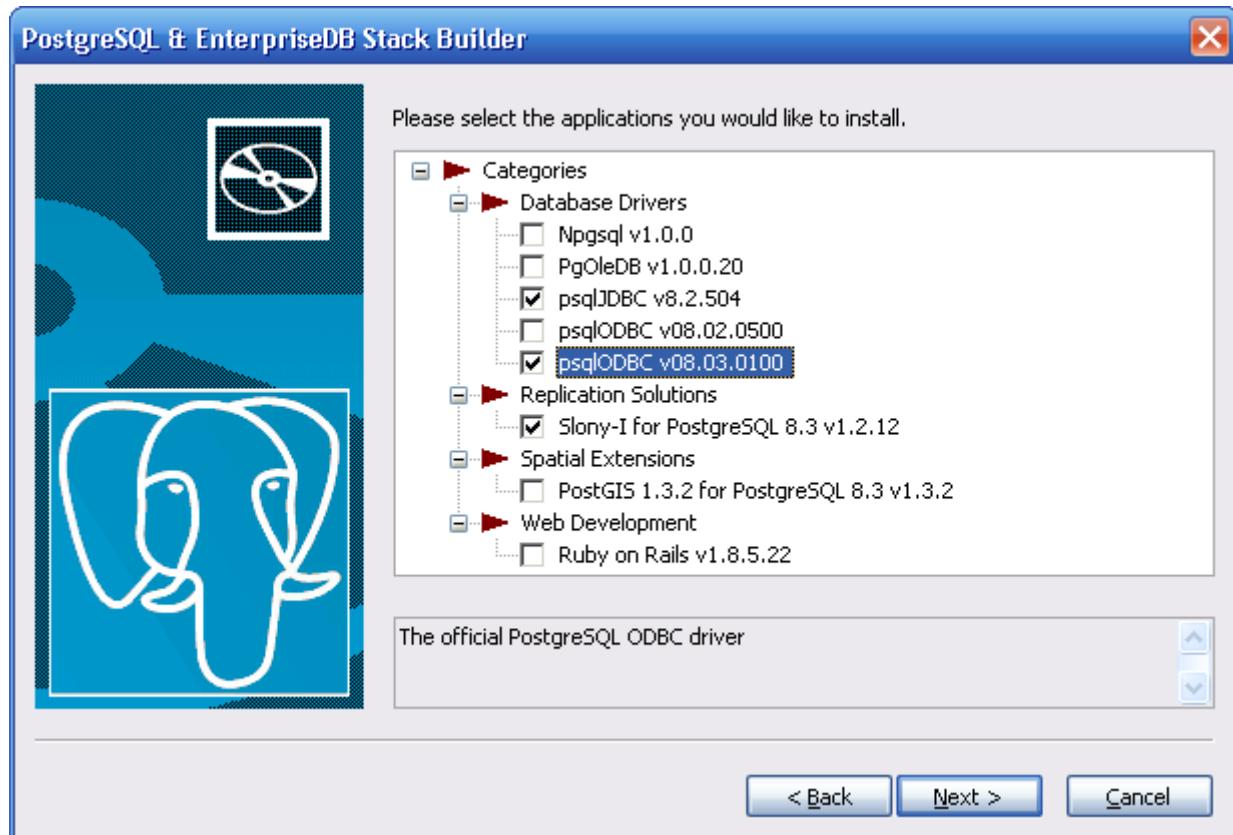
Caso você esteja realmente empenhado em aprender sobre o PostgreSQL aproveite esta oportunidade para assinar a lista pgsql-announce clicando no botão. Esta lista envia em torno de um e-mail por dia sobre as novidades do postgresql, novas ferramentas e informações importantes de segurança, bugs, etc.

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.

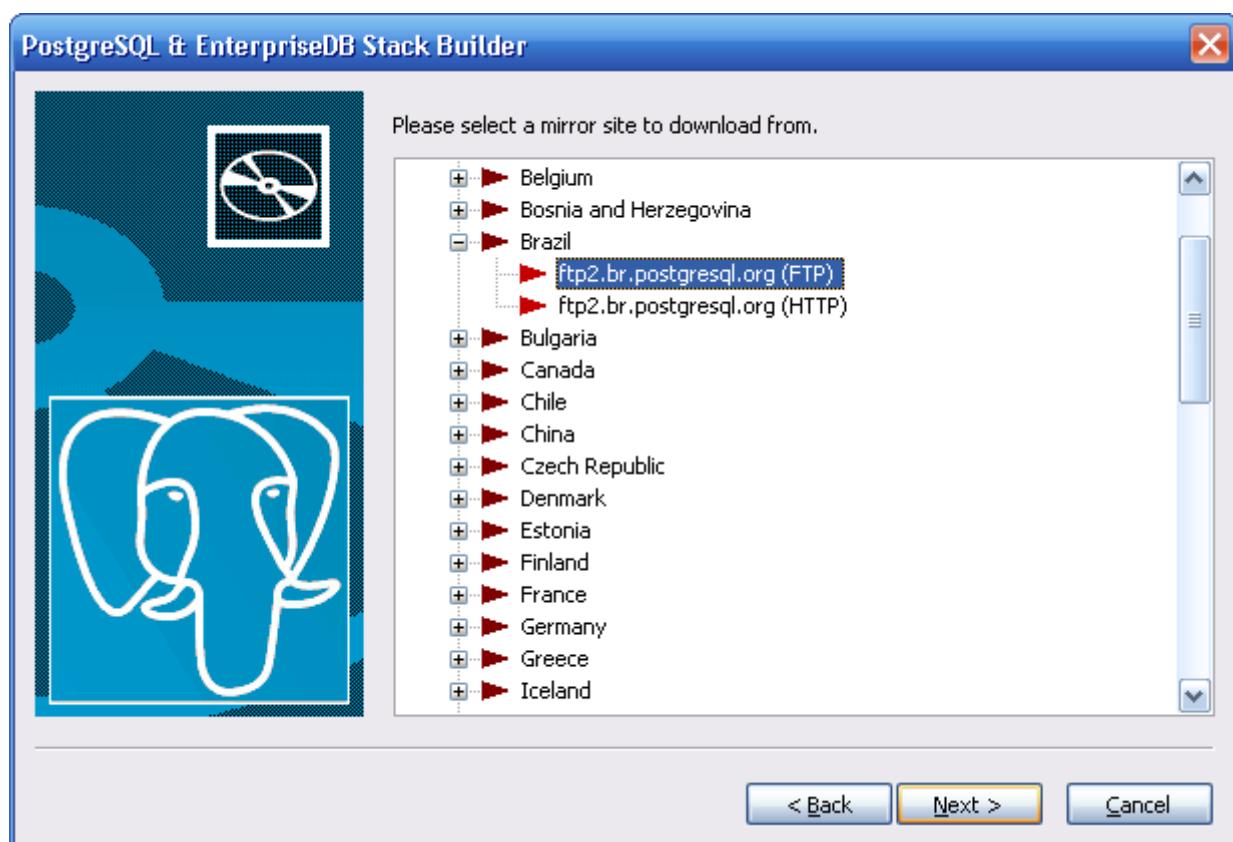


Selecione o servidor e caso esteja numa rede com proxy entre com as informações e clique em Next.

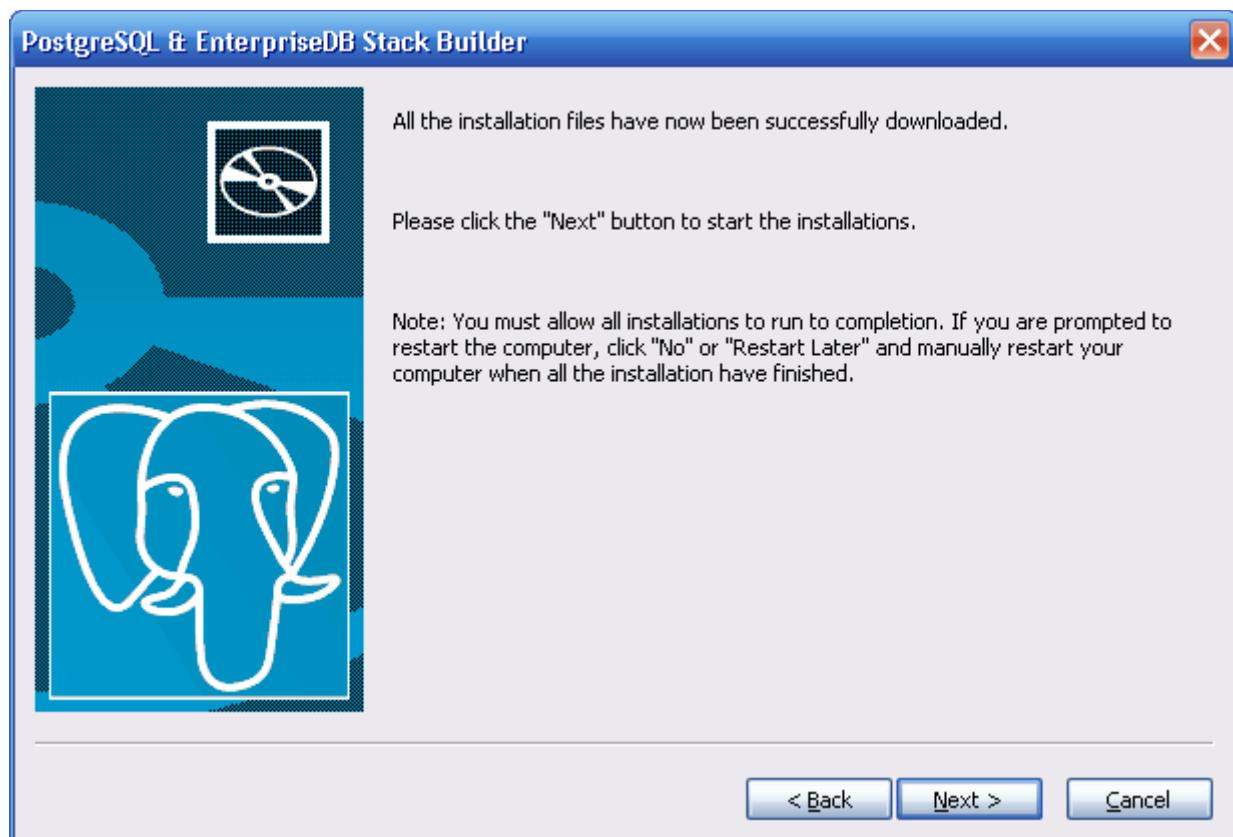
Novamente deverá dar permissão no firewall para que acesse o site para os downloads.



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



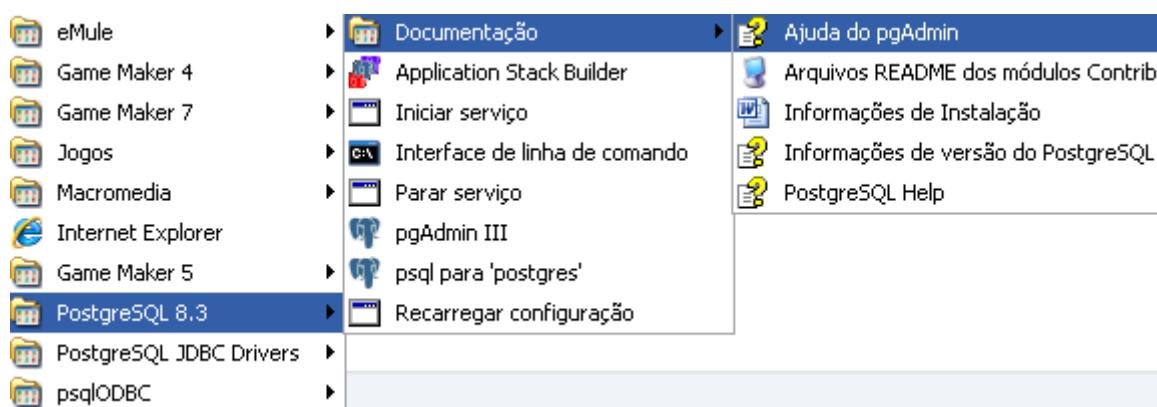
Apenas selecione o espelho e clique em Next e na próxima tela confirme em Next novamente e aguarde o download.



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

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 do PostgreSQL no Linux

Instalação no Debian e derivadas

```
sudo su  
apt-get install postgresql postgresql-doc;
```

Instalação num RedHat ou derivado

```
yum install postgresql postgresql-server
```

```
Criar o cluster  
service postgresql initdb
```

```
Iniciar  
service postgresql start
```

```
Manter na inicialização  
chkconfig --levels 235 postgresql on
```

Definições

Comandos DDL, DQL, DML, DCL, DTL, TCL?

Os comandos SQL são agrupados em cinco categorias. Segue abaixo suas definições e utilizações.

DCL - Data Control Language: comandos grant e revoke (administração dos usuários, grupos e privilégios)

DDL - Data Definition Language: comandos de definição e alteração de dados: create, drop, alter, rename, etc

DML - Data Manipulation Language: comandos para manipular dados: insert, update, delete

DQL - Data query Language: comandos para consulta de dados: select

Abordar primeiro o insert, update e delete
finalmente o Select, em várias formas:

Consultando dados

Order by

Distinct

Filtrando dados

where

limit

In

Between

Like

and e or

with

DDL – Data Definition Language - Linguagem de Definição de Dados.

Estes comandos são utilizados para definir a estrutura de banco de dados, criando ou removendo objetos.

CREATE- criar banco de dados, tabelas, colunas.

DROP - remover um objeto no banco de dados.

ALTER – altera a estrutura da base de dados

TRUNCATE – remover todos os registros de uma tabela, incluindo todos os espaços alocados para os registros são removidos. Limpa a tabela por completo. Semelhante ao parâmetro Purge de remoção de programas no Linux.

COMMENT – adicionar comentários ao dicionário de dados

RENAME – para renomear um objeto

DQL – Data Query Language - Linguagem de Consulta de Dados.

Utilizado para consultas dos dados.

SELECT- recuperar dados do banco de dados

DML – Data Manipulation Language - Linguagem de Manipulação de Dados.

Utilizados para o gerenciamento de dados dentro de objetos do banco.

INSERT – inserir dados em uma tabela

UPDATE – atualiza os dados existentes em uma tabela

DELETE – exclui registros de uma tabela,

CALL – chamar um subprograma PL / SQL

EXPLAIN PLAN – explicar o caminho de acesso aos dados
 LOCK TABLE – controle de concorrência

DCL – Data Control Language - Linguagem de Controle de Dados

Conjunto de comandos utilizados para controlar o nível de acesso de usuários.

GRANT – atribui privilégios de acesso do usuário a objetos do banco de dados

REVOKE – remove os privilégios de acesso aos objetos obtidos com o comando GRANT

DTL OU TCL – Transaction Control Language - Linguagem de Transação de Dados

São usados para gerenciar as mudanças feitas por instruções DML . Ele permite que as declarações sejam agrupadas em transações lógicas .

COMMIT – salvar o trabalho feito

SAVEPOINT – identificar um ponto em uma transação para que mais tarde você pode efetuar um ROLLBACK.

ROLLBACK – restaurar banco de dados ao original desde o último COMMIT

Estes comandos são os principais usados no gerenciamento, manutenção e consulta de um banco de dados relacional.

Fonte:

<http://www.vivaolinux.com.br/dica/Conhecendo-a-linguagem-SQL/>

<http://www.dellanio.com/diferenca-entre-comandos-ddl-dml-dcl-e-tcl/>

ALTER TABLE...

ALTER TABLE playground ADD last_maint date;

ALTER TABLE playground DROP last_maint;

Podemos

- adicionar um campo
- remover um campo
- adicionar uma constraint
- remover uma constraint
- alterar o valor default
- alterar o tipo de dados de um campo
- renomear um campo
- reomear uma tabela

Adicionar um Campo

alter table produtos add column descricao varchar(100) not null;

ou

... check(descricao <> "");

Obs.: o que foi usado no create table pode ser usado aqui no alter table.

Remover um Campo

alter table produtos drop column descricao;

Também removerá os dados do campo e suas constraints.

Para alterar campo referenciado por outra tabela precisamos adicionar cascade.

```
alter table produtos drop column descricao cascade;
```

Adicionar uma Constraint

```
alter table produtos add constraint check(nome <> "");  
alter table produtos add constraint nome unique(producto_id);  
alter table produtos add foreign key (produto_grupo_id) references produto_grupos;
```

Adicionar Not Null

```
alter table produtos alter column produto_id set not null;
```

Removendo uma Constraint

Para remover uma constraint você precisa conhecer seu nome. Se tiver um nome é fácil. Caso contrário precisará verificar com:

```
\d tabela
```

Anote o nome e use:

```
alter table produtos drop constraint nomevisto;
```

Alguns nomes precisarão de aspas duplas.

Remover um Not Null

```
alter table produtos alter column produto_id drop not null;
```

Obs.: não podemos dar nomes para a constraint not null.

Alterando o valor Default de um Campo

```
alter table produtos alter column preco set default 7.87;
```

Isso não afetará nenhum dos registros existentes, apenas os próximos.

Remover Valor Default

```
alter table produtos alter column preco drop default;
```

Alterar o Tipo de Dados de um Campo

```
alter table produtos alter column preco type numeric(8,2) not null;
```

Somente será bem sucedido se todos os registros existentes puderem ser convertidos para o novo tipo e tamanho de dados.

Para uma conversão mais complexa podemos usar um USING. O PostgreSQL também tentará converter os valores.

Renomera um Campo

```
alter table produtos rename column produto_id to id;
```

Renomeando uma Tabela

```
alter table produto rename to produtos;
```

Restrições/Constraints

As restrições são adicionadas para impor regras a nível de tabela, para agir sempre que um registro for inserido, alterado ou excluído. A operação somente será bem sucedida se atender a todas as restrições dos campos.

As restrições também impedem que um registro seja alterado dependendo de tabela relacionada. Assim como a exclusão de tabelas relacionadas precisam atender às restrições das foreign key.

Tipos de Restrições

not null - campo com not null obrigatoriamente deve ser preenchido. Cuidado, pois simplesmente a adição de um espaço em branco já atende às exigências do not null.

primary key - indica que um campo ou grupo de campos é exclusivo para todos os registros da tabela. A primary key é a combinação de duas constraints, a unique e a not null.

Cada tabela somente pode conter uma única primary key, mas uma primary key pode conter mais de um campo.

A adição de uma primary key automaticamente cria um índice do tipo btree no campo da pk.

A teoria dos bancos de dados relacionais diz que toda tabela deve conter uma chave primária, mas o PostgreSQL não obriga isso.

foreign key - especifica que o valor de um campo precisa ser o mesmo da tabela primária. Um campo ou grupo de campos que se repetem na tabela atual e que se ligam à primary key da tabela relacionada. Cada tabela pode conter uma ou mais foreign key. Usar not null em campos de fk, para garantir integridade.

Integridade referencial entre duas tabelas.

Exemplo:

produtos 1 ---- (vários) pedidos

```
create table produtos(      -- tabela referenciada
    id int primary key,
    nome varchar(50) not null,
    preco numeric(6,2)
);
```

Queremos garantir que somente tenhamos pedidos de produtos que existam.

```
create table pedidos( -- tabela referenciando
    id int primary key,
    produto_id int references produtos (id) not null,
    quantidade int not null
);
```

Assim como os campos

produtos.id - referenciado
pedidos.producto_id - referenciando

Uma fk também pode ser composta por mais de um campo, quando a pk referenciada também deve conter o mesmo número de campos e também o mesmo tipo de dados da fk

Outro exemplo

produtos
pedidos
pedido_itens

```
create table pedido_itens(
    ...
    produto_id int references produtos(id) not null,
    pedido_id int references pedidos (id) not null,
    ...
    primary key(produto_id, pedido_id)
);
```

unique - Muito importante. Assegura que este campo será exclusivo em todos os registros. Devemos usar unique not null, pois caso esqueça do not null os campos podem conter null e se repetirão.

Quando usamos unique not null criamos uma estrutura semelhante à primary key.

check - define uma condição que cada registro deve satisfazer.

```
create table produtos(
    numero int primary key,
    nome varchar(50) not null,
    descricao varchar(100) not null,
    preco numeric(8,2) not null
    constraint ck_preco_positivo
    check (preco >0),
    data date
);
```

Os valores que atendem ao check são o true e o null, por isso para maior segurança devemos usar o check com not null.

A constraint unique adiciona um índice btree ao campo.

Podemos adicionar restrições na criação da tabela ou adicionar, alterar ou remover usando

alter table

Exemplos

DISTINCT - mostrar somente valores distintos/exclusivos

Valor Default - entrar com o valor default em um campo é uma forma de impedir que o campo venha a conter valor NULL. Outra forma é adicionar uma constraint do tipo not null. O valor default pode ser uma constante ou uma expressão.

Exemplo:

```
preco numeric(6,2) default 9.99,
```

```
data date default now(),
```

O grande problema do valor default é que não haverá nenhuma crítica na cadastramento. Se o usuário não entrar nada no campo, o valor default será assumido e armazenado.

É preciso cuidado e melhor é evitar suaa doção.

Integridade Referencial

Implementando a política

- Quando alguém tentar remover um produto que permanece refenciado por um pedido (através dos itens de pedido) não permitir
- Quando alguém remover um pedido, seus itens devem ser removidos

```
create table produtos(...);
```

```
create table pedidos (...);
```

```
create table pedido_itens(
```

```
...
produto_id int references produtos(id) on delete restrict,
pedido_id int references pedidos(id) on delete cascade,
quantidade int not null,
primary key(produto_id, pedido_id)
);
```

RESTRICT - impede exclusão

CASCADE - exclue todos os itens relacionados

NO ACTION - se existir algum registro referenciado um erro será disparado. Este é o comportamento default.

SET NULL - torna null os referenciados como default

SET DEFAULT - torna o valor default

ON UPDATE

Uma Foreign Key somente pode se referenciar com campos que sejam Primary Key ou UNIQUE.

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 portáveis (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)
```

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)
```

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)
```

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)
```

Funções com Data e Hora

current_date

current_time

current_timestamp

date_part()

date_trunc()

extract()

new()

timeofday()

7 - Relacionamentos entre tabelas

1 - 1

1 - N (vários)

N (vários) - N (vários)

O último (N para N) precisa ser dividido em dois, assim:

N -> 1 e 1 <- N

Precisa nascer uma terceira tabela.

Integridade Referencial

- Não permitir que registros sejam inseridos na tabela referenciada sem estar na tabela atual
- Não permitir a exclusão de registros da tabela referenciada que estejam na tabela atual.

Ao adicionar ON DELETE CASCADE no relacionamento estamos dizendo ao SGBD que ao excluir um pedido também exclua todos os seus itens.

Ao adicionar ON UPDATE CASCADE, estamos dizendo que ao atualizar um pedido que o SGBD atualize seus itens.

Os relacionamentos podem também ser opcionais ou obrigatórios.

```
ALTER TABLE `tb_product` ADD FOREIGN KEY(`id_category`)
```

```
REFERENCES `tb_category`(`id_category`) ON DELETE CASCADE ON UPDATE
RESTRICT;
```

```
create table nome(
```

```
...
```

```
CONSTRAINT `customer_fk` FOREIGN KEY (`user_id`) REFERENCES `users` (`id`) ON
DELETE CASCADE ON UPDATE CASCADE
```

```
);
```

Vejamos como as instruções se comportam.

RESTRICT: vai restringir a algum conjunto de valores predeterminados.

NO ACTION: caso a condição seja satisfeita, nenhuma ação será executada.

CASCADE: caso uma operação afeta alguma linha, será verificado se essa operação desencadeou uma espécie de 'efeito dominó', ou seja, um efeito cascata: o que alterei em cima, vai alterar em baixo e assim por diante.

SET NULL: torna aquela instância para valores NULL.

SET DEFAULT: vai 'setar' os valores para o valor padrão correspondente. Ex.: valor numérico, o default seria 0; já para um campo do tipo caractere, o valor padrão seria o caractere vazio " " ou string vazia "".

Usando restrições de integridade referencial em cascata é possível definir as ações que o SQL Server toma quando o usuário tenta excluir ou atualizar uma chave para a qual apontam as chaves estrangeiras existentes.

As cláusulas REFERENCES das instruções CREATE TABLE e ALTER TABLE oferecem suporte às cláusulas ON DELETE e ON UPDATE. Ações em cascata também podem ser definidas usando-se a caixa de diálogo Relações de Chave Estrangeira:

[ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT }]

[ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT }]

NO ACTION é o padrão quando ON DELETE ou ON UPDATE não estão especificadas.

ON DELETE NO ACTION

Especifica que se uma tentativa for feita para exclusão de uma linha com uma chave referenciada por chaves estrangeiras em linhas existentes de outras tabelas, um erro é ativado e a instrução DELETE é revertida.

ON UPDATE NO ACTION

Especifica que se uma tentativa for feita para atualização de uma chave referenciada em uma linha cujas chaves sejam referenciadas por chaves estrangeiras em linhas existentes de outras tabelas, um erro é ativado e a instrução UPDATE é revertida.

CASCADE, SET NULL e SET DEFAULT permitem exclusões ou atualizações dos valores de chave que afetam as tabelas definidas para ter relações com as chaves estrangeiras e possam ser rastreadas até a tabela na qual a modificação foi feita. Se as ações referenciais em cascata também tiverem sido definidas em tabelas de destino, as ações em cascata especificadas também se aplicam àquelas linhas excluídas ou atualizadas. CASCADE não pode ser especificada para todas as chaves estrangeiras ou chaves primárias com a coluna timestamp.

ON DELETE CASCADE

Especifica que se uma tentativa for feita para exclusão de uma linha com uma chave referenciada por chaves estrangeiras em linhas existentes de outras tabelas, todas as linhas que contenham essas chaves estrangeiras serão igualmente excluídas.

ON UPDATE CASCADE

Especifica que se uma tentativa for feita para atualização de um valor de chave em uma linha, onde o valor de chave é referenciado pelas chaves estrangeiras em linhas existentes em outras tabelas, todos os valores que constituem a chave estrangeira serão igualmente atualizados no novo valor especificado para a chave.

Observação

CASCADE não poderá ser especificada se uma coluna timestamp integrar a chave estrangeira ou a chave referenciada.

ON DELETE SET NULL

Especifica que se uma tentativa for feita para exclusão de uma linha com uma chave referenciada por chaves estrangeiras em linhas existentes de outras tabelas, todos os valores que constituem a chave estrangeira nas linhas que são referenciadas serão definidos como NULL. Todas as colunas de chave estrangeira da tabela de destino devem ser anuláveis para que essa restrição seja executada.

ON UPDATE SET NULL

Especifica que se uma tentativa for feita para atualização de uma linha com uma chave referenciada por chaves estrangeiras em linhas existentes de outras tabelas, todos os valores que constituem a chave estrangeira nas linhas referenciadas serão definidos como NULL. Todas as colunas de chave estrangeira da tabela de destino devem ser anuláveis para que essa restrição seja executada.

ON DELETE SET DEFAULT

Especifica que se uma tentativa for feita para exclusão de uma linha com uma chave referenciada por chaves estrangeiras em linhas existentes de outras tabelas, todos os valores que constituem a chave estrangeira nas linhas referenciadas serão definidos como seus valores padrão. Todas as colunas de chave estrangeira da tabela de destino precisam ter uma definição padrão para que essa restrição seja executada. Se a coluna for anulável e não houver nenhum valor padrão explícito definido, NULL se tornará o valor padrão implícito para a coluna. Todos os valores não nulos definidos porque ON DELETE SET DEFAULT precisa conter valores correspondentes na tabela primária para poder manter a validade da chave estrangeira.

ON UPDATE SET DEFAULT

Especifica que se uma tentativa for feita para atualização de uma linha com uma chave referenciada por chaves estrangeiras em linhas existentes de outras tabelas, todos os valores que constituem a chave estrangeira nas linhas referenciadas serão definidos

como seus valores padrão. Todas as colunas de chave estrangeira da tabela de destino precisam ter uma definição padrão para que essa restrição seja executada. Se a coluna for anulável e não houver nenhum valor padrão explícito definido, NULL se tornará o valor padrão implícito para a coluna. Todos os valores não nulos definidos porque ON UPDATE SET DEFAULT precisa ter valores correspondentes na tabela primária para poder manter a validade da chave estrangeira.

Considere a restrição FK_ProductVendor_Vendor_VendorID na tabela Purchasing.ProductVendor em AdventureWorks2008R2. Essa restrição estabelece uma relação de chave estrangeira da coluna VendorID na tabela ProductVendor para a coluna de chave primária VendorID da tabela Purchasing.Vendor. Se ON DELETE CASCADE for especificada para a restrição, excluir a linha em Vendor, onde VendorID equivale a 100 também exclui as três linhas de ProductVendor onde VendorID equivale a 100. Se ON UPDATE CASCADE for especificada para a restrição, atualizar o valor VendorID da tabela Vendor de 100 para 155 também atualizada os valores VendorID nas três linhas de ProductVendor cujos valores VendorID equivalem atualmente a 100.

ON DELETE CASCADE não pode ser especificada para a tabela que tenha um gatilho INSTEAD OF DELETE. Para tabelas que tenham gatilhos INSTEAD OF UPDATE, não é possível especificar o seguinte: ON DELETE SET NULL, ON DELETE SET DEFAULT, ON UPDATE CASCADE, ON UPDATE SET NULL e ON UPDATE SET DEFAULT.

Várias ações em cascata

As instruções individuais DELETE ou UPDATE podem iniciar uma série de ações referenciais em cascata. Por exemplo, um banco de dados contém três tabelas: TableA, TableB e TableC. Uma chave estrangeira em TableB é definida com ON DELETE CASCADE de acordo com a chave primária em TableA. Uma chave estrangeira em TableC é definida com ON DELETE CASCADE contra a chave primária em TableB. Se uma instrução DELETE excluir linhas na TableA, a operação excluirá igualmente todas as linhas na TableB que tiverem chaves estrangeiras correspondentes a chaves primárias excluídas na TableA e, depois, excluirá todas as linhas na TableC que tenham chaves estrangeiras que correspondam a chaves primárias excluídas na TableB.

As séries de ações referenciais em cascata disparadas por uma única DELETE ou UPDATE devem formar uma árvore que não contenha referências circulares. Nenhuma tabela pode aparecer mais de uma vez na lista com todas as ações referenciais em cascata que resultem de DELETE ou UPDATE. Da mesma forma, a árvore de ações referenciais em cascata não precisa ter mais que um caminho para nenhuma tabela especificada. Toda ramificação da árvore termina quando encontra a tabela para a qual NO ACTION foi especificada ou é padrão.

Gatilhos e ações referenciais em cascata

As ações referenciais em cascata acionam os gatilhos de AFTER UPDATE ou AFTER DELETE da seguinte maneira:

Todas as ações referenciais em cascata causadas diretamente por DELETE ou UPDATE originais são executadas em primeiro lugar.

Se houver gatilhos AFTER definidos nas tabelas afetadas, esses gatilhos serão acionados depois que todas as ações referenciais em cascata forem executadas. Os gatilhos são acionados em ordem oposta à ordem da ação em cascata. Se houver vários gatilhos em uma única tabela, eles serão acionados em ordem aleatória, a menos que haja um gatilho dedicado final ou inicial para a tabela. Essa ordem é especificada usando-se `sp_settriggerorder`.

Se várias cadeias em cascata se originarem da tabela que era o destino direto de uma ação UPDATE ou DELETE, a ordem em que essas cadeias acionam seus respectivos gatilhos não é especificada. Porém, uma cadeia sempre aciona todos os seus gatilhos antes que outra cadeia inicie o acionamento.

Um gatilho AFTER em uma tabela que seja o destino direto de ações UPDATE ou DELETE é acionado independentemente de alguma linha ter sido ou não afetada. Não há nenhuma outra tabela afetada em cascata nesse caso.

Se algum dos gatilhos anteriores executar operações UPDATE ou DELETE em outras tabelas, essas ações poderão dar início a cadeias secundárias em cascata. Essas cadeias secundárias são processadas para todas as operações UPDATE ou DELETE em dado momento após o acionamento de todos os gatilhos em todas as cadeias primárias. Esse processo pode ser repetido recursivamente para operações UPDATE ou DELETE subsequentes.

Executar CREATE, ALTER, DELETE ou outras operações DDL (Data Definition Language) nos gatilhos pode fazer com que os disparadores DDL sejam acionados. Isso pode, subsequentemente, executar operações DELETE ou UPDATE que iniciam cadeias e gatilhos adicionais em cascata.

Se um erro for gerado em qualquer cadeia de ação referencial em cascata, um erro é ativado, nenhum gatilho AFTER é acionado naquela cadeia e a operação DELETE ou UPDATE que criou a cadeia é revertida.

Uma tabela com um gatilho INSTEAD OF não pode ter igualmente uma cláusula REFERENCES especificando uma ação em cascata. No entanto, um gatilho AFTER em uma tabela direcionada por uma ação em cascata poderá executar instruções INSERT, UPDATE ou DELETE em outra tabela ou exibição que acionem um gatilho INSTEAD OF definido naquele objeto.

Informações de catálogo de restrições referenciais em cascata

Entendendo a Herança de tabelas

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 um 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.

Sequência

Serial - é um inteiro que gera uma sequência automática somando + 1.

É um número sequencial usado nas primary key.

Quando usamos serial em um campo, o PostgreSQL adiciona automaticamente uma sequência.

8 - Tipos de Dados no PostgreSQL

O PostgreSQL suporta os tipos de dados padrões do SQL ANSI:

Numéricos

- integer
- arbitraria precisão
- ponto fluruante
- serial

Monetário

Character

Binário

- byea hex
- bytea escape

Data/Hora

- data/hora input
- data/hora output
- time zones
- interval input
- interval output

Boolean

Enumerated

Geométricos

- pontos
- linha
- boxes
- path
- polígonos
- círculos

Redes

- inet
- cidr
- inet vycidr
- macadd

Bit string

Text search

UUID

XML

JSON

Array

Compostos

Range

Object Identifier

Pseudo

E um rico conjunto de tipos de dados geométricos.

Além de poder ter novos tipos de dados criados pelo usuário com create type....

Os tipos especificados pelo SQL são:

bigint
 bit
 bit varying
 boolean
 char
 character varying
 character
 varchar
 date
 double precision
 integer
 interval
 numeric
 decimal
 real
 smallint
 time (with or without time zone)
 timestamp (with or without time zone)
 xml

Boolean/boolean

TRUE/FALSE

Valores possíveis

t/f

...

Dados do tipo Array

-- Quero poder entrar com mais de um telefone no campo telefone

create table servidores(

```

    id int primary key,
    nome varchar(50) not null,
    telefone varchar(10)[]
  );
```

\d servidores

insert into servidores values (

```
1, 'Ribamar FS', '{"988884444"}, {"999994444"}');
```

);

select * from servidores;

postgres=# select * from servidores;

id	nome	telefone
----	------	----------

--	--	--

1 | Ribamar FS | {{988884444},{999994444}}

CASTING - conversão de tipos de dados explicitamente

CAST ('string' as type)

Tipos Numéricos têm 2, 4 ou 8 bytes

Números com Precisão Arbitrária

Valores numéricos podem armazenar números com muitos dígitos e executar cálculos exatos. É especialmente indicado para armazenar valores monetários e outras quantidades onde a exatidão é requerida. A aritmética com valores numéricos é bem mais lenta que com tipos inteiros ou mesmo de ponto flutuante.

numeric (p,e)

p - precisão é o total de dígitos (a parte antes do ponto). > 0

e - escala, é a quantidade de decimais do número, depois do ponto. >= 0

Exemplo:

numeric(8,2)

precisão - 8

escala - 2

O tipo decimal é equivalente à numérico.

NaN - Not a Number

Tipos de Dados de Ponto Flutuante

Os tipos de dados real e double são inexatos e de variável precisão numérica.

- Quando for requerida exatidão (como dinheiro) use o tipo numeric.
- Comparando dois valores iguais de ponto flutuante nem sempre o resultado funciona como esperado.

Tipo de Dados Serial

Smallserial, serial e bigserial não são tipos reais de dados. São inteiros com uma sequência automática.

Tipo Money

'R\$1.200,00'

Tipo Character

varchar(n)

char(n) - este é mais lento

Tipos Binários

bytea

Tipos Data/Hora

As datas seguem o calendário Gregoriano.

Datas até 4713 dc.

epoch

infinity

-infinity

now

today

tomorrow

yesterday

Tipo Booleano

TRUE FALSE (preferidos)

't' 'f'

'true' 'false'

'y' 'n'

'yes' 'no'

'on' 'off'

'1' '0'

Tipos de Dados Enum

É um conjunto de valores ordenados.

Exemplo:

Os dias da semana

São criados usando

```
create type semana as enum ('seg','ter','qua','qui','sex','sab','dom');
```

Usando

```
create table pessoa(
```

```
    nome text,
```

```
    dia semana
```

```
);
```

```
insert into pessoa values ('João', 'ter');
```

```
select * from pessoa where semana = 'ter';
```

Tipos Geométricos

Representam objetos em 2 dimensões.

Ponto é a base dos demais tipos.

Pontos são especificados usando

(x,y)

x,y

Onde x e y são as coordenadas, como números em ponto flutuante.

Pontos são marcados usando a primeira sintaxe.

Segmento de Reta

Segmentos de reta são representados por pares de pontos.

São especificados usando uma das sintaxes:

[(x1,y1),(x2,y2)]

((x1,y1),(x2,y2))

(x1,y1),(x2,y2)

x1,y1, x2,y2

Onde x1,y1 é o ponto inicial e x2,y2 é o ponto final do segmento.

Segmentos são mostrados usando a primeira sintaxe.

Boxes / Caixas - são representados por pares de pontos dos cantos opostos do box.

Os pontos são especificados assim:

((x1,y1),(x2,y2))

(x1,y1),(x2,y2)

x1,y1, x2,y2

Onde (x_1, y_1) e (x_2, y_2) são os cantos opostos do box.

Boxes são exibidos usando a primeira sintaxe.

Os pontos devem ser

(x_1, y_1) - acima e à esquerda

(x_2, y_2) - abaixo e à direita

Não obrigatoriamente na ordem.

PATH/Caminho - são representados por uma lista de pontos conectados. Os paths podem ser abertos ou fechados. Abertos quando o primeiro e o último pontos não se conectam. Fechados quando o primeiro e o último ponto se conectam.

Paths são especificados usando:

$[(x_1, y_1), \dots, (x_n, y_n)]$

$((x_1, y_1), \dots, (x_n, y_n))$

$(x_1, y_1), \dots, (x_n, y_n)$

$x_1, y_1, \dots, x_n, y_n$

Onde (x_1, y_1) é o ponto inicial do primeiro segmento e (x_n, y_n) é o ponto final do último segmento que forma o path.

[] indicam um path aberto

() indicam um path fechado

Quando a última sintaxe é usada presume-se um path fechado.

A primeira sintaxe é a mais apropriada para mostrar paths.

Polígonos - são representados por uma lista dos pontos dos seus vértices. São similares a paths fechados, mas são armazenados de forma diferente e contam com rotinas próprias.

São especificados usando:

$[(x_1, y_1), \dots, (x_n, y_n)]$

$(x_1, y_1), \dots, (x_n, y_n)$

$(x_1, y_1), \dots, (x_n, y_n)$

$x_1, y_1, \dots, x_n, y_n$

São mostrados usando a primeira sintaxe.

Círculos - São representados por um ponto no centro e por um raio.

São especificados assim:

$\langle(x,y), r\rangle$

$((x,y), r)$

x, y, r

Onde (x,y) é o centro e r é o raio.

Círculos são mostrados usando a primeira sintaxe.

Tipos de Dados Endereço de Rede

É melhor usar um tipo de dados do PostgreSQL do que o formato texto, pois o postgresql critica seus tipos e têm funções específicas para eles.

INET/inet - manipula endereços IPv4 e IPv6

Se a máscara é 32 em IPv4 então não indica uma subrede mas apenas um host.

Formato

endereço/y

Se apenas y estiver ausente então 32 para IPv4 e 128 para IPv6. O valor representa apenas um único host.

CIDR/cidr - Lida com especificação de rede IPv4 ou IPv6. O formato segue cidr.

cidr input	cidr output	abrev
------------	-------------	-------

192.168.100.128/25 192.168.100.128/25

inet x cidr

A diferença é que **inet** aceita valores com bit não zero para a direita da máscara, já **cidr** não aceita.

macaddr - armazena endereços de MAC de placas de rede.

Inputs aceitam os seguintes formatos:

'08:00:2b:01:02:03'

'08-00-2b-01-02-03'

'08002b-010203'

'0800.2b01.0203'

'08002b010203'

Todos os exemplos acima especificam o mesmo MAC. Maiúsculas e minúsculas são

aceitas.

Como é composto de hexadecimais, então são válidos:

0-9

A-F

A primeira forma é a mais usada.

BIT Strings - são strings de 1 e 0.

bit(n) ou b'101 ou b'100 ou

variying(n)

Exemplo:

create table ...

insert into

select...

Tipo de Busca de Texto

O PostgreSQL tem dois tipos de dados que suportam full text search.

tsvector - representa um documento em uma forma otimizada para busca

tsquery - representa uma consulta de texto.

tsvector - é uma lista ordenada de distintos lexemes. A ordenação e a eliminação de duplicados é feita automaticamente no input.

select 'a fat cat sat on a mat and ate a fat rat'::tsvector;

Opcionalmente inteiro para as posições podem ser anexados as lexemes.

Exemplos

Tipo UUID - Armazena UUID como definido pela RFC 4122.

128 bit

Hexadecimais minúsculas em vários grupos separados por hífens.

Um grupo de 8 dígitos seguido por 3 grupos de 4 dígitos, seguidos por um grupo de 12 dígitos, perfazendo um total de 32 dígitos representando 128 bit.

Exemplo:

a0eebc99-9c0b-4ef8-bbbd-bbb9bd380a11

Tipo de Dados XML - pode ser usado para armazenar dados no formato XML. O uso deste tipo de dados requer que na instalação se tenha usado:

```
configure --with-libxml
```

Funções

xmlparse - para produzir um valor de tipo xml

xmlserialize - produz um valor de caracteres string de xml

Dados tipo JSON - pode ser usado para armazenar dados do tipo JSON.

Tipo Array - o PostgreSQL suporta campos do tipo array de comprimento variável e multidimensional.

```
create table salarios(
```

```
    pagamento int[],
```

```
    telefone text[2][9]
```

```
);
```

Acessando

```
select * from salarios where
```

```
    telefone[0][0] = '{85}{934912786}';
```

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

Comparação de Registro com Array

Todas as expressões retornam true/false:

in(expr)

not in(expr)

any/some(array)

all(array)

Exemplos

Concatenação

```
---- || -----
```

Tipos Compostos - Criação de um tipo composto por vários tipos existentes.

```
create type composto as(
```

```
    i int,
```

```
    nome text,
```

```
    preco numeric
```

```
);
```

```
create type tpreco as(
```

```

produto text,
quantidade int,
preco numeric(6,2)

);

create table produtos(
    id int primary key,
    preco tpreco
);

insert into produtos values(
    (1, ROW('Banana', 100, 2.75));

```

Podemos usar em tabelas e em funções

```
update produtos set preco = ROW('Goiaba', 20, 3.85) where ...
```

Range Types - representa uma faixa de valores de algum tipo.

Chamado faixa de subtipo.

Alguns ranges nativos:

int2range - range de smallint

int4range - range de int

int8range - range de bigint

numrange - range de numeric

Podemos criar novos ranges usando create type ...

Exemplos

OID - Object ID - são usados internamente pelo PostgreSQL como chave primária para várias tabelas de sistema.

O tipo OID representa um identificador de objeto.

É atualmente implementado como um inteiro de 4 bytes não sinalizado.

PSEUDO TIPOS - Não podem ser usados como nomes de campos nas nossas tabelas.

Exemplos:

any

anyenum

internal

record

void

opaque

9 - Agrupando Registros

Sintaxe

`select campo, função(campo)`

`from tabela`

`group by campo`

`having campo;`

Group by condensa em uma única linha todos os registros que possuem o mesmo valor do campo do group by.

`select sexo, count(*) from alunos`

`group by sexo;`

group by pode ser usada em mais de um campo

Alguns exemplos

Estas funções computam um resultado simples de um conjunto de valores.

avg

count(campo)

count(*)

max

min

stddev

sum

variance

Todas ignoram campos com valor null, exceto count(*)

`select função(campo) from tabela where clausula;`

SUM

`select salario, sum(salario) over (order by salario) from empregados;`

10 - EXPRESSÃO CASE

Similar ao comando IF ... THEN

Pode ser usada sempre que uma expressão for válida.

Cada condição é uma expressão que retorna um resultado booleano.

Se a condição resultante for true, o valor da expressão CASE é o resultado que segue o THEN e o restante não será processado. Se o resultado da condição for false o restante será examinado.

Se ELSE for omitido e nenhuma condição for true o resultado será null.

Sintaxe:

CASE WHEN condição THEN resultado

WHEN ...

ELSE resultado2;

END

Exemplo simples:

```
select * from teste;
```

```
a
```

```
---
```

```
1
```

```
2
```

```
3
```

```
select a,
```

```
  case when a = 1 then 'Um'
```

```
    when a = 2 then 'Dois'
```

```
    else 'Outro'
```

```
end
```

```
from teste;
```

```
a   case
```

```
1   Um
```

- 2 Dois
- 3 Outro

Exemplo para substituir a letra do sexo pela descrição:

```
select nome, (CASE sexo
    WHEN 'M' THEN 'Masculino'
    WHEN 'F' THEN 'Feminino'
    ELSE 'Sexo não informado'
END) AS sexo
```

from servidores;

Quantos alunos tem conceito Excelente, Bom, Regular e Insuficiente

```
select (case
    when nota_final between 9.00 and 10.00 then 'Excelente'
    when nota_final between 7.00 and 8.99 then 'Bom'
    when nota_final between 5.00 and 6.99 then 'Regular'
    when nota_final between 0.00 and 4.99 then 'Insuficiente'
    else 'Nota não informada'
end) as Conceito,
count(*) as quantos
```

from alunos group by 1;

Valores Armazenados

UNION, CASE e Construções Relacionadas

1) Se todas as entradas são do mesmo tipo e são não unknown, será resolvido como o tipo. Caso contrário, sobrescreve todos os domínios dos tipos da lista com seus underlying base type.

- 2) Se todas as entradas são de tipo unknown, resolve como tipo text. Caso contrário as entradas unknown serão ignoradas.
- 3) Se nem todas as entradas non-unknown são do mesmo tipo, falhará.
- 4) Escolher a primeira entrada non-unknown que é um tipo preferido na categoria, se for um.
- 5) De outra forma, escolha a última entrada unknown que permite todas as entradas precedentes non-unknown para ser implicitamente convertidas para esta.
- 6) Converter todas as entradas para o tipo selecionado. Falhará se não for uma conversão de uma dada entrada para o tipo selecionado.

CASTS - Conversão explícita de tipos

SQL é uma linguagem fortemente tipada.

Operador Fatorial !

```
select 40! as 'Fatorial de 40';
```

Concatenação ||

```
select text 'abc' || 'def' AS 'Teste';
```

Valor Absoluto @

```
select @ '-4' AS 'Absoluto';
```

Absoluto

4

11 - 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, intersecção 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,
 $(\text{consulta1 UNION consulta2}) \text{ UNION consulta3}$

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)
```

Detalhes em: <http://pgdocptbr.sourceforge.net/pg80/queries-union.html>
 Tutorial online: http://www.w3schools.com/sql/sql_union.asp

12 - Trabalhando com SQL

Quase todas estas dicas abaixo recebi na lista de PostgreSQL do <http://postgresql.org.br>.
 Sempre que lembrei concedi os devidos créditos.

1) Criar Tabela tendo outra outra como base e já importando todos os registros dessa outra:

CREATE TABLE tabelanova AS SELECT * FROM tabealexistente;

2) Inserindo com SELECT

Inserir todos os registros de uma tabela em outra:

INSERT INTO tabelaqueimporta SELECT * from tabelaqueexporta;

```

insert into engenharia.insumos (grupo,insumo,descricao,unidade) select
grupo,insumo,descricao, CAST(unidade AS int2) AS "unidade" from engenharia.apagar
insert into engenharia.insumos (grupo,insumo,descricao,unidade) select
grupo,insumo,descricao, cast(unidade AS INT2) AS unidade from engenharia.apagar
$conn = pg_connect("host=10.40.100.186 dbname=apoena user=_postgresql");
for($x=10;$x<=87;$x++){
$sql="update engenharia.precos set custo_produtoivo = (select custo_produtoivo from
engenharia.apagar where insumo='$x') where insumo='00' || '$x'";
$ret=pg_query($conn,$sql);
}

```

3) Atualizar um campo em todos os registros de uma tabela recebendo de outra tabela:

UPDATE servicos s SET custo = total FROM composicoes c
WHERE s.tabela = c.tabela AND s.servico = c.servico

Uso do Like e de Expressões Regulares

Registros:

Ribamar Ferreira de Sousa
João Pereira Brito

Usando LIKE e ILIKE

SELECT * FROM clientes WHERE nome LIKE 'Riba%'; // Retorna Ribamar Ferreira de Sousa

SELECT * FROM clientes WHERE nome LIKE 'riba%'; // Nada retorna

SELECT * FROM clientes WHERE nome ILIKE 'riba%'; // Retorna Ribamar Ferreira de Sousa

SELECT * FROM clientes WHERE nome NOT LIKE 'pedro'; // Retorna ambos os registros

Usando Expressões Regulares

SELECT * FROM clientes WHERE nome ~~ 'Riba%'; // Retorna Ribamar Ferreira de Sousa

SELECT * FROM clientes WHERE nome ~~ 'riba%'; // Nada retorna

SELECT * FROM clientes WHERE nome ~~* 'riba%'; // Retorna Ribamar Ferreira de Sousa

SELECT * FROM clientes WHERE nome !~~ 'pedro'; // Retorna ambos os registros

SELECT nome FROM clientes WHERE nome ~ 'Ribamar Ferreira de Sousa'; // Retorna Ribamar Ferreira de Sousa

SELECT * FROM clientes WHERE nome !~ 'jorge'; // Retorna ambos

4) Buscar nas tabelas de sistema do postgresql, todos as tabelas de um determinado schema, os campos que sejam do tipo boolean..

```
SELECT n.nspname AS Schema, c.relname AS Tabela, t.typname AS Tipo
FROM pg_class c
LEFT JOIN pg_namespace n ON n.oid = c.relnamespace
LEFT JOIN pg_type t ON t.oid = c.relttype
WHERE c.relkind = 'r'::"char"
AND t.typname = 'boolean';
```

5) Exemplos de Joins

Join com 4 tabelas

```
$w_sql = " TRUE ";
if ( $p_tabela != "") { $w_sql = $w_sql . " AND tabela ~~*". $p_tabela . ""; }
if ( $p_insumo_grupo != "") { $w_sql = $w_sql . " AND insumo_grupo ~~*". $p_insumo_grupo.""; }
if ( $p_insumo != "") { $w_sql = $w_sql . " AND insumo ~~*". $p_insumo . ""; }
if ( $p_fornecedor != "") { $w_sql = $w_sql . " AND fornecedor ~~*". $p_fornecedor.""; }

$w_sql="SELECT distinct on (p.tabela, p.insumo_grupo, p.insumo, p.fornecedor)
p.custo_proutivo, p.data_inclusao,
t.tabela, t.descricao as tabelad,
ig.grupo, ig.descricao as insumogd,
i.grupo, i.insumo, i.descricao as insumod,
f.codigo_fornecedor, f.razao_social as fornecedord
FROM $m_table as p, $m_table_tab as t, $m_table_ing as ig, $m_table_ins as i,
$m_table_for as f
WHERE p.tabela=t.tabela AND p.insumo_grupo=ig.grupo AND p.insumo=i.insumo AND
p.fornecedor=f.codigo_fornecedor
AND p.insumo_grupo = i.grupo ORDER BY p.tabela DESC, p.insumo_grupo;";

/*
p – $m_table (engenharia.precos)
i – $m_table_ins (engenharia.insumos)
ig – $m_table_ing (engenharia.insumos_grupos)
t – $m_table_tab (engenharia.tabela)
*/
```

6) Mudar Tipo de Dados de Campo – CAST (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);

7) Renomear Tabela

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

8) Tamanho de Tabela, Banco ou Todos os Bancos do SGBD:

Tamanho de Banco de Dados (postgresql 8.1 ou superior):

```
select pg_database_size('nomebanco');
```

Tamanho de Tabela

```
select pg_tablespace_size('nometabela');
```

Tamanho de todos os bancos de dados do SGBD:

```
select (sum(relpages) * 8) / 1024 || ' MB' as tamanho from pg_class where relowner > 1;
```

Ou

```
select (sum(relpages) / 2^7) :: int || ' MB' as tamanho from pg_class where relowner > 1;
```

9) Validação de e-mails

- 1 – select distinct(campo_email),campo_nome, campos_n from tabela where campo_email like '%@%.%'
- 2 – SELECT POSITION('@', 'ribafs@gmail.com') > 0
- 3 – select 'coutinho.php@gmail.com' ~ '@'
- 4 – select 'coutinho.php@gmail.com' like '%@%
- 5 – select if ('campo_email' like "%@%.%", "TRUE", "FALSE") as flag, campo_adcional from tabela
- 6 – select 'coutinho@gmail.com' similar to '%@%.%';

10) Temos um campo (insumo) com valores = 1, 2, 3, ... 87

Queremos atualizar para 0001, 0002, 0003, ... 0087

```
UPDATE equipamentos SET insumo = '000' || insumo WHERE LENGTH(insumo) = 1;  
UPDATE equipamentos SET insumo = '00' || insumo WHERE LENGTH(insumo) = 2;
```

Outra saída mais elegante ainda:

```
UPDATE equipamentos SET insumo = REPEAT('0', 4-LENGTH(insumo)) || insumo;
```

11) Retornar o número de usuários conectados

```
select count(*) from pg_stat_activity
```

`pg_stat_database` que apresenta para cada banco de dados o número de conexões.
 Eu particularmente acho que fica mais fácil de visualizar do que o `pg_stat_activity` quando se tem muitas conexões.

Mostrar uso dos índices dos bancos de dados:

```
select * from pg_statio_user_indexes;
```

```
select * from pg_stat_user_indexes;
```

Mostra estatística de uso das tabelas e manutenção:

```
select * from pg_stat_all_tables;
```

Mostra todas as tabelas do atual esquema do atual banco:

```
select * from pg_stat_user_tables;
```

`pg_stat_get_tuples_returned(oid)` bigint Number of rows read by sequential scans when argument is a table, or number of index entries returned when argument is an index

`pg_stat_get_tuples_fetched(oid)` bigint Number of table rows fetched by bitmap scans when argument is a table, or table rows fetched by simple index scans using the index when argument is an index

`pg_stat_get_tuples_inserted(oid)` bigint Number of rows inserted into table

`pg_stat_get_tuples_updated(oid)` bigint Number of rows updated in table

`pg_stat_get_tuples_deleted(oid)` bigint Number of rows deleted from table

`pg_stat_get_blocks_fetched(oid)` bigint Number of disk block fetch requests for table or index

`pg_stat_get_blocks_hit(oid)` bigint Number of disk block requests found in cache for table or index

`pg_stat_get_last_vacuum_time(oid)` timestamptz Time of the last vacuum initiated by the user on this table

`pg_stat_get_last_autovacuum_time(oid)` timestamptz Time of the last vacuum initiated by the autovacuum daemon on this table

`pg_stat_get_last_analyze_time(oid)` timestamptz Time of the last analyze initiated by the user on this table

`pg_stat_get_last_autoanalyze_time(oid)` timestamptz Time of the last analyze initiated by the autovacuum daemon on this table

This is controlled by configuration parameters that are normally set in `postgresql.conf`

The function `pg_stat_get_backend_idset` provides a convenient way to generate one row for each active server process. For example, to show the PIDs and current queries of all server processes:

```
SELECT pg_stat_get_backend_pid(s.backendid) AS procpid,
       pg_stat_get_backend_activity(s.backendid) AS current_query
  FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

Visualizar os processos do postgresql num UNIX:

```
ps auxww | grep ^postgres
```

Formato de retorno:

```
postgres: user database host activity
```

12) Corrigindo Estouro do Máximo de transações (2 bilhões)

Constatando:

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

age acusa mais de 2 bilhões

Tarcizio Meurer

- Execute um dumpall na base
- drop a base e o agrupamento de dados
- recrie o agrupamento
- recrie a base
- carregue os dados novamente.

13) Total de Registros de Todos os Bancos do SGBD (PHP):

```
<?php
$cconexao=pg_connect("host=127.0.0.1 user=postgres password=postabir");
$sql="SELECT datname AS banco FROM pg_database ORDER BY datname";
$consulta=pg_query($conexao,$sql);

$banco = array();
$c=0;
while ($data = @pg_fetch_object($consulta,$c)) {
$cons=$data->banco;

$banco[] .= $cons;
$c++;
}

$sql2="SELECT n.nspname as esquema,c.relname as tabela FROM pg_namespace n,
pg_class c
WHERE n.oid = c.relnamespace
and c.relkind = 'r' — no indices
and n.nspname not like 'pg\\_%' — no catalogs
and n.nspname != 'information_schema' — no information_schema
ORDER BY nspname, relname";

for ($x=0; $x < count($banco);$x++){
if ($banco[$x] != "template0" && $banco[$x] != "template1" && $banco[$x] != "postgres"){
$conexao2=pg_connect("host=127.0.0.1 dbname=$banco[$x] user=postgres"
```

```

password=postabir");
$consulta2=pg_query( $conexao2, $sql2 );

while ($data = pg_fetch_object($consulta2)) {
$esquematab=$data->esquema.''.$data->tabela;
$sql3="SELECT count(*) FROM $esquematab";
$consulta3=pg_query($conexao2,$sql3);
$res=@pg_fetch_array($consulta3);

print 'Banco.Esquema.Tabela -> '.$banco[$x].'.'.$data->esquema.'.'.$data->tabela.' -
Registro(s) - '.$res[0].';
$total += $res[0];
}

}
}

print "Total de Registro de todas as tabelas de todos os bancos ". $total;

?>

```

14) Uso da Constraint check

```

CREATE TABLE testes(
codigo serial primary key,
idade integer,
check (idade > 18 AND idade < 70)
)

```

Alternativas:

```

check (preco > desconto)
check (desconto > 0 AND preco > desconto)

```

Somente aceitar c ou e (simulando campo tipo enum do MySQL):
 tipo char(1) check (tipo ='c' OR tipo='e')

Para este cria-se uma combo com values 'c' e 'e'.

15) Manutenção do PostgreSQL:

No CRON:

```
/home/pgsql/bin/psql -c "vacuum full analyse" -d dadosadv -U postgres
```

Consultas no Prompt do SO:

```
psql -U postgres -d banco -c "SELECT * FROM clientes"
```

Manutenção em Tabela
vacuum analize tabela;

Reindexar Banco, tabela ou índice
reindex database banco;

Exibir plano de consulta
explain select * from tabela;

Exibir todos os parâmetros de runtime
show all;

16) Consulta com Dias Úteis

Só para constar aqui vai uma expressão SQL que fornece os dias úteis de um período. Considerei que existe uma tabela com o registro dos feriados e outros dias que não devem ser considerados (emendas, pontos facultativos, etc):

```
SELECT dia FROM
(SELECT ('2007-10-01'::date+s.a*'1 day'::interval) AS dia
FROM generate_series(0, '2007-10-31'::date -
'2007-10-01'::date, 1) AS s(a)) foo
WHERE EXTRACT(DOW FROM dia) BETWEEN 1 AND 5
EXCEPT
SELECT dia FROM tab_feriado;
Osvaldo (na lista postgresql-br)
```

17) Update em uma chave primária sem causar duplicação de chave

```
UPDATE teste SET coluna1 = t_aux.coluna1+1
FROM (
SELECT coluna1
FROM teste
ORDER BY coluna1 DESC
) t_aux
WHERE teste.coluna1 = t_aux.coluna1;
Osvaldo (na lista postgresql-br)
```

18) Como saber se existe uma transação ativa

select pg_stat_activity;

Dica do João Paulo.

19) Inserir data como valor default:

Pode usar também o current_date ou o localtimestamp.

```
insert into tabela(data) values ((select current_date));
```

ou

```
insert into tabela(data) values ((select localtimestamp));
```

20) Ler último saldo de tabela

Tenho o seguinte conteúdo numa tabela de contas:

Lancto	CCorrente	Banco	OP	Data Lan	Valor	Saldo
1	12345-6	002	C	19/11/2007	1000.00	1000.00
2	12345-6	002	C	19/11/2007	2000.00	3000.00
3	12345-6	002	D	19/11/2007	100.00	2900.00
4	23450-6	001	C	19/11/2007	2000.00	3000.00
5	23450-6	001	D	19/11/2007	100.00	2900.00

Preciso retornar sempre o último SALDO registrado.

Como nunca vou saber a data exata da período de consulta.

Estou executando:

```
SELECT saldoatual FROM lanban WHERE contacorrente = '12345-6' and datalan <=
'2007/12/01' ORDER BY datalan DESC LIMIT 1
```

Retona o Saldo: 1000.00, preciso pegar o ultimo saldo da conta 12345-6: que é 2900.00.

Isso porque tabelas são conjuntos de dados. O padrão SQL *não* garante a ordem dos dados. Mesmo se ele garantisse, um simples UPDATE podia mudar a ordem dos dados e o seu SELECT não retornaria o valor desejado.

> Alguém tem alguma dica?

>

O campo 'Lancto' é do tipo serial? Se for poderias utilizar:

```
SELECT saldoatual FROM lanban WHERE contacorrente = '12345-6' ORDER BY
" Lancto " DESC LIMIT 1.
```

Dica do Euler Taveira de Oliveira

21) Formato de moeda

O correto seria:

```
to_char(1030.52,'9G999D99')
```

mas o resultado é: 1,030,52

como você pode observar existe um problema no

separador de milhar (indicado pelo G) que é considerado como , e não como . que seria o esperado.

Uma maneira de contornar (não muito elegante) é:
to_char(1030.52,'9'."999D99')

Corrigido na versão 8.3

22) Saber o Tamanho de Tabela e de Índices

pg_relation_size()
pg_total_relation_size()

-Leo

—
Leonardo Cezar

23) Último Saldo

Fernando Brombatti

A situação é a seguinte. Não se sabe se o serial citado (por N razões) vai ser o último valor existente. Nada me garante que estes dados não sofreram algum UPDATE. Sendo assim, recomendo:

1) alterar o campo DATE para TIMESTAMP

2) alterar o query:

SELECT lan.saldoatual

FROM lanban lan

WHERE lan.contacorrente = '12345-6' AND lan.datalan = (SELECT MAX(maxlan.datalan)

FROM lanban maxlan

WHERE maxlan.contacorrente = lan.contacorrente)

Isso faz com que no primeiro SQL eu traga os lançamentos da conta e no segundo eu trago a máxima data de lançamento para a mesma conta. Como as contas são iguais, trago a máxima data da conta atual, logo tenho o saldo atual.

É confuso, mas é o mais seguro (podem haver UPDATES neste caso também, mas aí não se depende de um serial).

Para este query funcionar bem necessita mais um índice em datalan ao menos.

Nos nossos sistemas da prefeitura nunca usamos saldos desta forma, pois aí se é removido algum registro a informação não fica correta.

Espero não ter confundido tanto.

24) Encontrando tabela de sistema

Para localizar informações desse tipo existe o information_schema (conforme citado pelo Leandro). Utilizando o catalogo poupa voce de

futuras dores de cabeça quando por exemplo houver alguma alteração estrutural em tabelas do sistema em versões futuras. As views do catalogo deverão permanecer com o máximo de compatibilidade entre versões (segundo padrão SQL).

Além de ser mais simples:

```
SELECT *
FROM information_schema.tables
WHERE table_name = 'foobar';
```

Infelizmente não possuímos referencias a outros banco de dados (banco.schema.tabela), portanto o comando deverá ser executado em todos seus bancos para localizar a tabela ou um programeta bash parecido com isso:

```
$ ARG=$1 || "foo" && for DATABASE in `psql -U postgres -c "\l" \
| cut -d"," -f1 | egrep '^(\ [a-z])'
do
psql -U postgres -d $DATABASE -Atc \
"SELECT 'O banco de dados $DATABASE possui a tabela: $RG'
FROM information_schema.tables
WHERE table_name = '$ARG'";
done;
```

Abraço!

-Leo

25) Como Localizar e Deletar registros duplicados

1.Select para localizar duplicados

```
select campo,campo1,count(*)
from tabela group by campo,campo1 having count(*) > 1
```

2.Deletar duplicados:

```
delete from tab p1
where rowid < (select max(rowid)
from tab1 p2
where p1.primary_key = p2.primary_key);
```

26) Inserir registros em uma específica posição

```
> Hi, how are you? maybe you know how SQL insert data
> bellow or above in database tabe? example insert
> data from position table 5 thanks
>
```

No, I don't known.

But if you make a copy from table,
create a new table with same structure,
insert a new register,
import register from old table, then first register
are this last register inserted.

27) Timezones do PostgreSQL (lista pgbr-geral)

No POSTGRESQL.CONF tem o timezone onde você pode colocar algo do tipo:

TIMEZONE=BRAZIL/EAST esta é minha configuração, ou seja, de minha região.

Analise.

Wandrey

Outra —————

Na maioria dos casos é criado um link do diretório de timezones do S.O. (/usr/share/zoneinfo//usr/share/zoneinfo/) para o diretório de Timezones do Postgres (\$PGDIR/share/timezone)Que possui seu próprio sistema de controle de timezone, se não me engano a partir d versão 8)

—

Att:

Thiago Risso

28) Inserir Número Aleatório em Tabela

```
CREATE TABLE page (
    id SERIAL PRIMARY KEY,
    about TEXT NULL,
);
```

```
ALTER TABLE page ADD myrand NUMERIC NOT NULL DEFAULT RANDOM();
```

```
UPDATE page SET myrand = DEFAULT;
```

```
SELECT id FROM page WHERE myrand >= RANDOM() ORDER BY myrand LIMIT 1;
```

This approach has some problems:

- * If the number you pick is greater than the largest number in the myrand column, you will not find any matching rows.
- * The gaps between the random values in the myrand column are not uniform, and thus the rows selected are not random. Imagine a table with two rows and myrand values of 0.8 and 0.9. If the random number compared to myrand is .8 or less, the first row is chosen. But the second row is only chosen if the value picked is between .8 and .9
- * If more than one row has the exact same number, it is likely that one of them will never get picked.

Mais detalhes em: <http://people.planetpostgresql.org/greg/index.php?archives/118-guid.htm...>

29) Desabilitar Triggers

Vinicio Santos – MSI escreveu:

Thiago Boufleuhr escreveu:

Como faço para desabilitar as triggers em uma sessão no PLSQL ?

Thiago Boufleuhr

```
ALTER TABLE [NOME DA TABELA]
DISABLE TRIGGER [NOME DA TRIGGER]
```

Ou

```
ALTER TABLE [NOME DA TABELA]
DISABLE TRIGGER ALL
```

ALERTA:

William Leite Araújo: MUITO CUIDADO AO USAR “DISABLE TRIGGER ALL”

As constraints de chave estrangeira são controladas via TRIGGER. Caso desabilite todos os gatilhos, a checagem da integridade referencial (chaves estrangeiras) serão desabilitadas!

30) Codificação de Caracteres

Euler Taveira de Oliveira

>Evandro Ricardo Silvestre wrote: Codificação de caracteres do cliente e do servidor podem ser diferentes. Se a codificação do cliente é diferente da codificação do servidor, o servidor PostgreSQL tenta fazer uma conversão antes de armazenar/retornar os dados. Um problema que existia é que a aplicação cliente (no caso abaixo o psql) não avisava se a codificação informada ao servidor (client_encoding) era a mesma do ambiente (terminal).

Bem vindo ao psql 8.3.0, o terminal iterativo do PostgreSQL.

Digite: \copyright para mostrar termos de distribuição

\h para ajuda com comandos SQL

\? para ajuda com comandos do psql

\g ou terminar com ponto-e-vírgula para executar a consulta

\q para sair

template1=# show client_encoding;

client_encoding

LATIN1

(1 registro)

template1=# show server_encoding;

server_encoding

LATIN1

(1 registro)

template1=# select upper('áéíóú');

upper

ÁÉÍÓÚ

(1 registro)

template1=# set client_encoding to 'utf-8';

SET

template1=# show client_encoding;

client_encoding

utf-8

(1 registro)

template1=# select upper('áéíóú');

ERRO: sequência de bytes é inválida para codificação “UTF8”: 0xe1e9ed

DICA: Este erro pode acontecer também se a sequência de bytes não corresponde a codificação esperado pelo servidor, que é controlada por “client_encoding”.

ERRO: sequência de bytes é inválida para codificação “UTF8”: 0xe1e9ed

DICA: Este erro pode acontecer também se a sequência de bytes não corresponde a codificação esperado pelo servidor, que é controlada por “client_encoding”.

[trocando a codificação de caracteres do terminal e digitando novamente]

template1=# select upper('áéí');

upper

ÁÉÍ

(1 registro)

31) Como visualizar as consultas correntes no Postgres

Colaboração: Frederico Palma

Data de Publicação: 16 de fevereiro de 2008

É necessário habilitar o stats_command_string no postgresql.conf:

stats_command_string = true

Essa configuração pode ser alterada em um banco que está ativo sem a necessidade de reiniciá-lo e sem afetar as conexões abertas para recarregar as configurações. Envie um SIGHUP ou use o comando:

pg_ctl reload

Quando stats_command_string está ativo a tabela pg_stat_activity armazena todas consultas correntes.

Realizando a consulta:

```
SELECT datname,procpid,current_query FROM pg_stat_activity
```

Teremos a lista dos bancos de dados utilizados com seus respectivos processos (PID) referente às consultas.

```
SELECT datname,procpid,current_query FROM pg_stat_activity ORDER BY procpid;
```

```
datname | procpid | current_query
```

datname	procpid	current_query
mydatabase1	2587	< IDLE >
mydatabase2	15726	SELECT * FROM users WHERE id=123 ;
mydatabase3	15851	< IDLE >

Publicado originalmente na Dicas-L – <http://www.dicas-l.com.br/dicas-l/20080216.php>

13 - Utilizando SQL para selecionar, filtrar e agrupar registros

- 1) A linguagem SQL
- 2) Principais Palavras-Chave e Identificadores
- 3) Trabalhando e Manipulando Valores Nulos
- 4) Utilizando Comentários
- 5) Entendendo os Tipos de Dados
- 6) Utilizando Expressões e Constantes
- 7) Ocultando Linhas Duplicadas em uma Consulta (DISTINCT)
- 8) Limitando o Resultado do Select
- 9) Utilizando o Comando Case
- 10) Substituindo Valores Nulos para Formatar dados Retornados na Consulta

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-élé".

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) 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>

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.

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 */
```

Tipos de Dados

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 bissexto, 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;
```

6) Utilizando Expressões e Constantes

Expressões SQL em: http://db.apache.org/derby/docs/dev/pt_BR/ref/rrefsqlj19433.html

Precedência dos operadores (decrescente)

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>

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.

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;
```

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

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>

LIMIT - Retorna somente uma quantidade especificada de registros em uma consulta

O order by complementa.

LIMIT com OFFSET - pular certa quantidade (OFFSET) antes de mostrar uma quantidade (limit).

```
select nome from servidores order by nome limit 15 offset 6;
```

Pula os 6 primeiros e traz os próximos 15.

14 - Desvendando o SELECT

Cláusula FROM

Comentários

Cláusula JOIN

Cláusula WHERE

Exemplos de utilização

Cláusula GROUP BY

Cláusula HAVING

Cláusula ORDER BY

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.

SELECT INTO

Cria uma nova tabela partindo de uma ou mais tabelas existentes.

A nova tabela conterá apenas os campos do select. Se * todos, mas se especificados apenas os especificados.

```
select * into informatica from disciplinas where codigo = 'inf';
```

Este comando não traz as constraints, por conta disso o create table é preferido. Usando devemos após a criação adicionar as constraints manualmente.

15 - Entendendo e Utilizando sub-consultas

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).

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 seqüência); 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

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

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.

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 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

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.

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).

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 preço 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);
```

Sub Consultas

Quando temos um select dentro de outro select para recuperar registros filtrados de outra consulta.

Mostrar disciplinas que possuem o mesmo nr. de créditos de Álgebra:

```
select disciplinas from disciplinas
  where creditos = (select creditos
                      from disciplinas
                     where descricao = 'Álgebra');
```

Todas as expressões retornam true/false

```
exists(subqry)
in(subqry)
not(subqry)
any(subqry)
all(subqry)
```

16 - Reuso de Código: Utilizando Funções

- 1) Introdução às Funções - 1
- 2) Utilizando Funções matemáticas - 2
- 3) Utilizando Funções de data e hora - 4
- 4) Utilizando Funções de Texto (Strings) - 7
- 5) Utilizando Funções de Conversão de Tipos (Casts) - 10
- 6) Utilizando Outras Funções - 11
- 7) Entendendo as Funções de Agregação - 14

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>

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.

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)

Extrai 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);
```

Sumar 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', 'YYMMDD')
```

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, millennium 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 subtrair 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:

Cusará 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:

Cusará Erro.

Como testar estes exemplos no PostgreSQL?

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

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('UNIFOR');

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 Brasão`
`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 equivale ao ILIKE

!~~ equivale ao NOT LIKE

!~~* equivale 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

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 portáveis 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);
```

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 Geométricos

```
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(maraddr '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) 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

Funções Nativas

Funções Matemáticas

- abs
- cbrt
- ceil
- ceiling
- degree
- div
- exp
- floor
- ln
- log
- mod
- pi
- power
- radians
- random
- round
- setseed
- sign
- sqrt
- trunc
- width_bucket

numeric

Arredondamento

```
select round (4,3);
```

```
round
```

```
----
```

```
4.000
```

Funções Trigonométricas

acos

asin

atan

atan2

cos

cot

sin

tan

Funções para Strings

Manipulação de strings, que são char, varchar, text e bit string

bit_length

lower

octet_length

overlay

position

substring

trim

upper

ascii

btrim

chr

concat

concat_ws

convert

convert_from

decode

encode

format

initcap

left

length

lpad

ltrim

```

md5
pg_client_encoding
quote_ident
quote_literal
quote_nullable
regexp_matches
regexp_replace
regexp_split_to_array
regexp_split_to_table
repeat
replace
reverse
right
rpad
rtrim
split_part
strpos
substr
to_ascii
to_hex
translate

```

Exemplo de substring

```
select substr('12345',3);
```

```
substr
```

```
-----
```

```
345
```

Funções String Binários

Para o tipo bytea

Funções para Bit Strings

Patern Matching

```

like
similarto
posix-stile

```

LIKE

```

'abc' like 'abc' - true
'abc' like 'a%' - true
'abc' like '_b_' - true
'abc' like 'c' - false

```

ILIKE - case sensitive

SIMILAR TO - parece com like mas interpreta o resultado de acordo com SQL.

SIMILAR TO aceita:

| - alternativa entre dois valores

* - repetição de item anterior zero ou mais vezes

+ - repetição de item anterior uma ou mais vezes

? - repetição de item anterior zero ou uma vez

{m} - repetição de item anterior extamente m vezes

{m,} - repetição de item anterior m ou mais vezes

{m,n} - repetição de item anterior pelo menos m e não mais que n vezes

() - podem ser usados para agrupar itens

[...] - especifica uma classe de caracteres

POSIX Regular Expressões

Tabela de operadores

Funções para Formatação de Datas

A configuração do estilo de datas pode ser feita pelo postgresql.conf para todos os bancos de dados ou pelo comando set no banco atual.

17 - Consultando dados em múltiplas tabelas

- 1) Utilizando Apelidos para as tabelas - 1
- 2) Cruzando dados entre tabelas distintas - 2
- 3) Entendendo os Tipos de Join disponíveis - 2
- 4) Trabalhando com CROSS JOIN - 2
- 5) Trabalhando com INNER e OUTER JOINs - 4
- 6) Trabalhando com NATURAL JOIN - 5

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**.

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.

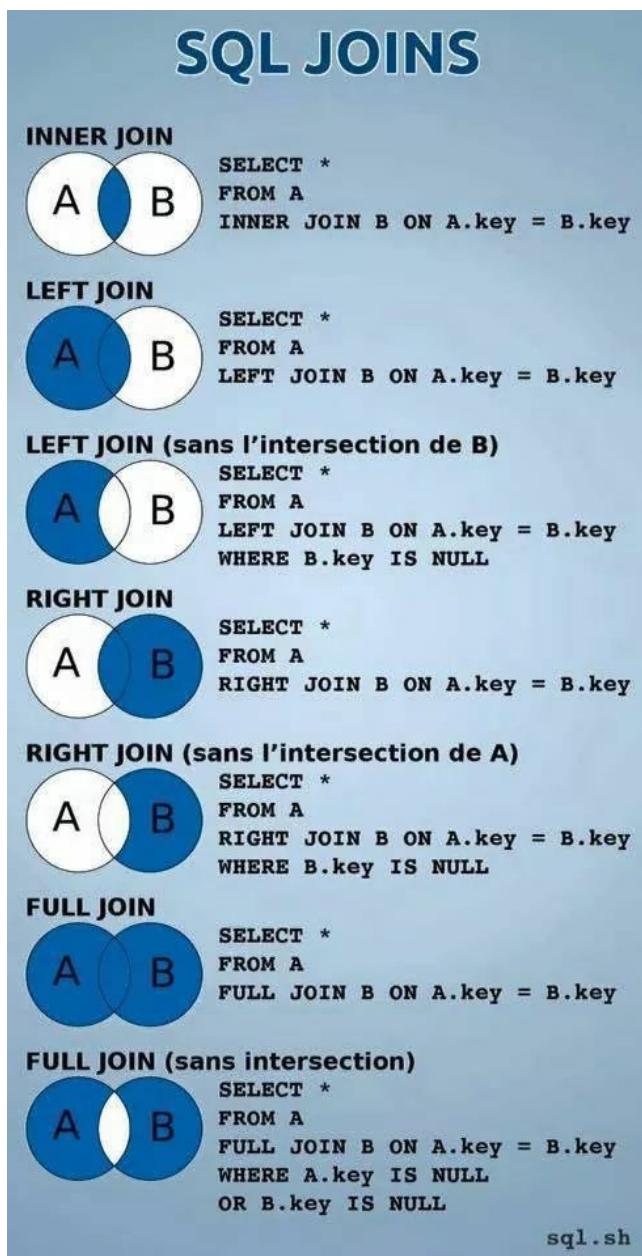
FULL OUTER JOIN

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
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**.

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;
```

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)

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

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 AxB.

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) 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 buscara 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 = ????

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

Last Name	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>

18 - Visões (views)

Tutorial sobre Views no PostgreSQL

View – uma view é uma consulta que recebe um nome e é armazenada no banco em que é criada.

Em que elas são úteis? Elas economizam grande quantidade de digitação e esforço e apresentam somente os dados que desejamos.

```
postgres=# \h create view
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] VIEW name [ ( column_name [, ...] ) ]
AS query
```

Pela sintaxe vemos que podemos usar:

- OR REPLACE para ao criar, se já existir sobrescrever a existente
- TEMP ou TEMPORARY para criar uma view temporária, com tempo de vida restrito à seção atual. Ao fechar o cliente (psql, pgAdmin, etc) a view criada é extinta.
- Podemos criar apelidos para os campos da tabela na view com as column_name, ...

Como a view praticamente é uma consulta, ela pode conter tudo que podemos usar em uma consulta em SQL.

Exemplo simples:

Caso uma empresa precise da relação mensal de aniversariantes de seus clientes, então é útil criar uma view com o seguinte conteúdo:

```
create table clientes(nome char(45) primary key, data_nasc date);
```

```
insert into clientes values ('João', '1986-05-29');
insert into clientes values ('Emanuela', '1978-05-29');
insert into clientes values ('Gerardo', '1967-05-29');
insert into clientes values ('Pedro', '1969-02-25');
-- to_char(current_date, 'DD/MM');
```

```
SET DATESTYLE TO sql,dmy;
```

```
CREATE OR REPLACE VIEW v_clientes_aniversariantes AS
SELECT nome FROM clientes
WHERE to_char(data_nasc, 'DD/MM')=to_char(current_date, 'DD/MM');
```

Executando:

```
select * from v_clientes_aniversariantes;
```

Usando Datas:

```
SELECT '10/01/2005'::DATE - '01/01/2000'::DATE
SELECT NOW() - '2001/1/1'
SELECT (current_date - '1956-08-03')/365
SELECT 'today'::date;
SELECT '12:16'::time;
SELECT '12:16:32.43'::time with time zone;
SELECT 'now'::time with time zone;
SELECT '2001-01-12'::timestamp;
```

Criando Uma View

```
CREATE VIEW recent_shipments
AS SELECT count(*) AS num_shipped, max(ship_date), title
FROM shipments
JOIN editions USING (isbn)
NATURAL JOIN books AS b (book_id)
GROUP BY b.title
ORDER BY num_shipped DESC;
```

Usando Uma View

```
SELECT * FROM recent_shipments;
SELECT * FROM recent_shipments
ORDER BY max DESC
LIMIT 3;
```

Destruindo Uma View

```
DROP VIEW nomeview;
```

Criar as Tabelas que servirão de Base

```
CREATE TABLE client (
    clientid SERIAL NOT NULL PRIMARY KEY,
    clientname VARCHAR(255)
);
```

```
CREATE TABLE clientcontact (
    contactid SERIAL NOT NULL PRIMARY KEY,
    clientid int CONSTRAINT client_contact_check REFERENCES client(clientid),
    name VARCHAR(255),
    phone VARCHAR(255),
    fax VARCHAR(255),
    emailaddress VARCHAR(255)
);
```

```
CREATE VIEW client_contact_list AS
SELECT client.clientid, clientname, name, emailaddress FROM client, clientcontact
WHERE client.clientid = clientcontact.clientid;
```

O nome da visão deve ser distinto do nome de qualquer outra visão, tabela, seqüência ou índice no mesmo esquema.

A visão não é materializada fisicamente. Em vez disso, a consulta é executada toda vez que a visão é referenciada em uma consulta.

Fazer livre uso de visões é um aspecto chave de um bom projeto de banco de dados SQL.

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

Atualmente, as visões são somente para leitura: o sistema não permite inserção, atualização ou exclusão em uma visão. É possível obter o efeito de uma visão atualizável criando regras que reescrevem as inserções, etc. na visão como ações apropriadas em outras tabelas. Para obter informações adicionais consulte o comando CREATE RULE.

`CREATE VIEW vista AS SELECT 'Hello World';`

é ruim por dois motivos: o nome padrão da coluna é ?column?, e o tipo de dado padrão da coluna é unknown. Se for desejado um literal cadeia de caracteres no resultado da visão deve ser utilizado algo como `CREATE VIEW vista AS SELECT text 'Hello World' AS hello;`

Veja capítulo 4 do Livro "Practical PostgreSQL"

Supondo que uma consulta seja de particular interesse para uma aplicação, mas que não se deseja digitar esta consulta toda vez que for necessária, então é possível criar uma view baseada na consulta, atribuindo um nome a esta consulta pelo qual será possível referenciá-la como se fosse uma tabela comum.

```
CREATE VIEW minha_view AS
  SELECT cidade, temp_min, temp_max, prcp, data, localizacao
    FROM clima, cidades
   WHERE cidade = nome;
SELECT * FROM minha_view;
```

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

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

Views

Objeto armazenado no banco de dados e destinado a visualizar informações de uma ou mais tabelas, criando com um select.

Uma view não guarda registros, pois estes permanecem nas tabelas de origem.

Uma view pode se referir a uma tabela ou a outra view e também usar funções.

Sempre que são executadas seus dados são atualizados com as alterações das tabelas.

Objetivos:

- Filtrar informações para os programadores;
- Simplificar consultas;
- Cada usuário ou grupo pode acessar somente aquilo que está sob seu domínio;
- Alterações futuras nas tabelas não requerem alterações no código do programador.

Criando:

```
create view nome as select * from tabela;
```

```
create view nome (campo1, campo2, campo3) as
    select campoa, campob, campoc from tabela;
```

campoa receberá campo1, campob receberá campo2 e campoc receberá campo3.

Executando uma view:

```
select * from nomeview;
```

Também podemos passar parâmetros:

```
select * from nomeview where campoa=2;
```

\dp – visualizar views e outros objetos

Que são VIEWS?

São uma maneira simples de executar e exibir dados selecionados de consultas complexas em bancos.

Em que elas são úteis?

Elas economizam grande quantidade de digitação e esforço e apresentam somente os dados que desejamos.

Criando Uma View

```
CREATE VIEW recent_shipments
AS SELECT count(*) AS num_shipped, max(ship_date), title
FROM shipments
JOIN editions USING (isbn)
NATURAL JOIN books AS b (book_id)
GROUP BY b.title
ORDER BY num_shipped DESC;
```

Usando Uma View

```
SELECT * FROM recent_shipments;
SELECT * FROM recent_shipments
    ORDER BY max DESC
    LIMIT 3;
```

Destruindo Uma View

```
DROP VIEW nomeview;
```

Criar as Tabelas que servirão de Base

```
CREATE TABLE client (
    clientid SERIAL NOT NULL PRIMARY KEY,
    clientname VARCHAR(255)
);
```

```
CREATE TABLE clientcontact (
    contactid SERIAL NOT NULL PRIMARY KEY,
    clientid int CONSTRAINT client_contact_check REFERENCES client(clientid),
    name VARCHAR(255),
    phone VARCHAR(255),
    fax VARCHAR(255),
    emailaddress VARCHAR(255)
);
```

```
CREATE VIEW client_contact_list AS
SELECT client.clientid, clientname, name, emailaddress FROM client, clientcontact
WHERE client.clientid = clientcontact.clientid;
```

Estando no psql e digitando \d podemos visualizar também as views.

O nome da visão deve ser distinto do nome de qualquer outra visão, tabela, seqüência ou índice no mesmo esquema.

A visão não é materializada fisicamente. Em vez disso, a consulta é executada toda vez que a visão é referenciada em uma consulta.

Fazer livre uso de visões é um aspecto chave de um bom projeto de banco de dados SQL.

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

Atualmente, as visões são somente para leitura: o sistema não permite inserção, atualização ou exclusão em uma visão. É possível obter o efeito de uma visão atualizável criando regras que reescrevem as inserções, etc. na visão como ações apropriadas em outras tabelas. Para obter informações adicionais consulte o comando CREATE RULE.

```
CREATE VIEW visao AS SELECT 'Hello World';
```

é ruim por dois motivos: o nome padrão da coluna é ?column?, e o tipo de dado padrão da coluna é unknown. Se for desejado um literal cadeia de caracteres no resultado da visão deve ser utilizado algo como CREATE VIEW visao AS SELECT text 'Hello World' AS hello;

Veja capítulo 4 do Livro "Practical PostgreSQL"

Supondo que uma consulta seja de particular interesse para uma aplicação, mas que não se deseja digitar esta consulta toda vez que for necessária, então é possível criar uma view baseada na consulta, atribuindo um nome a esta consulta pelo qual será possível referenciá-la como se fosse uma tabela comum.

```
CREATE VIEW minha_view AS
  SELECT cidade, temp_min, temp_max, prcp, data, localizacao
    FROM clima, cidades
   WHERE cidade = nome;
```

```
SELECT * FROM minha_visao;
```

Fazer livre uso de visões é um aspecto chave de um bom projeto de banco de dados SQL.

As visões permitem encapsular, atrás de interfaces que não mudam, os detalhes da estrutura das tabelas, que podem mudar na medida em que as aplicações evoluem.

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

Views

Juntamente com as funções armazenadas (stored procedures) as views são boas alternativas para tornar o código mais simples e o aplicativo mais eficientes, já que parte do processamento feito pelo código agora já está pronto e debugado no banco, o que torna o código mais rápido e eficiente. O uso de views e de funções armazenadas em bancos é semelhante ao uso de funções e classes no código.

19 - 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 seqüência de exemplos os tipos de junções de tabelas possíveis no PostgreSQL.

Completo em <https://imasters.com.br/artigo/2867/postgresql/juncao-entre-tabelas-no-postgresql>

Parte 2 - <https://imasters.com.br/artigo/2870/postgresql/juncao-entre-tabelas-no-postgresql-parte-02>

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/

JOIN

Recuperar dados de duas ou mais tabelas.

Caso o nome de algum campo se repita entre as tabelas, ele precisará ser prefixado com o nome da tabela.

NATURAL JOIN

Faz a junção de todos os campos com o mesmo nome em ambas as tabelas.

Exemplos:

JOIN USING

Usado para especificar colunas/campos.

JOIN ON

Usado para especificar campos quando não existem campos repetidos.

JOIN DE VÁRIAS TABELAS

NON EQUI JOINS

OUTER JOINS - pode ser usado para recuperar dados de uma tabela, mesmo que não existam linhas correspondentes para satisfazer a junção.

Exemplos

Tipos de OUTER JOINS:

LEFT OUTER JOIN

RIGHT OUTER JOIN

FULL OUTER JOIN

A palavra OUTER é opcional.

LEFT OUTER JOIN - Retorna registros que satisfazem a igualdade da junção, como também os registros da tabela do lado esquerdo que não satisfazem.

JOIN USING com OUTER JOIN

JOIN ON com OUTER JOIN

NATURAL JOIN com OUTER JOIN

RIGHT OUTER JOIN - Retorna os registros que satisfazem a igualdade da junção, como também retorna os registros do lado direito que não satisfazem.

FULL OUTER JOIN - retornam os registros que satisfazem a igualdade da junção e todos os outros que não satisfazem. Produto cartesiano?

SELF JOIN (AUTO JOIN) - Usado quando se deseja unir uma tabela com ela mesma. Analisa a tabela duas vezes.

EXEMPLOS DE JOINS SOFISTICADOS

```
CREATE TABLE logtable(id serial, tstamp timestamp, message text,mestype int4);
```

```
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;
```

Inner Join - <https://www.javatpoint.com/postgresql-join>
<http://www.postgresqltutorial.com/postgresql-inner-join/>

Left join - <https://www.javatpoint.com/postgresql-left-join>
<http://www.postgresqltutorial.com/postgresql-left-join/>

Right join - <https://www.javatpoint.com/postgresql-right-join>

Full outer join - <https://www.javatpoint.com/postgresql-full-join>
<http://www.postgresqltutorial.com/postgresql-full-outer-join/>

Cross join - <https://www.javatpoint.com/postgresql-cross-join>
<http://www.postgresqltutorial.com/postgresql-cross-join/>

Natural join - <http://www.postgresqltutorial.com/postgresql-natural-join/>

group by
having

union
intersect
except

Funções de agregação - <https://www.postgresql.org/docs/9.6/static/functions-aggregate.html>

Uma view é uma consulta armazenada ou uma tabela lógica. A view não armazena registros, apenas tem as consultas armazenadas. Os registros permanecem nas tabelas de origem.

Sintaxe

create [or replace] view nomeview as subconsulta;

Exemplo:

```
create view v_alunas
  as select nome from alunos
    where sexo = 'F';
```

Usando

```
select * from v_alunas;
```

Uma boa prática é preceder o nome das views com v_ para evitar confusão.

Views - <https://www.javatpoint.com/postgresql-view>
<https://www.postgresql.org/docs/9.6/static/rules-views.html>

20 - Transações

Entendendo a execução das transações e os tipos de isolamento no banco de dados

- 1) Uso de Transações no PostgreSQL
- 2) Entendendo os Níveis de isolamento
- 3) Usando Savepoints para controlar etapas das transações

1) 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.

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 verbose 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.

2) Entendendo os Níveis de isolamento

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 quatro 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.

2.1. 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 efetuarem 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*.

2.2. 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 pareça 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.

3) Usando Savepoints para controlar etapas das transações

SAVEPOINT

Nome

SAVEPOINT -- define um novo ponto de salvamento na transação corrente

Sinopse

SAVEPOINT nome_do_ponto_de_salvamento

Descrição

O comando SAVEPOINT estabelece um novo ponto de salvamento na transação corrente.

O ponto de salvamento é uma marca especial dentro da transação que permite desfazer todos os comandos executados após o seu estabelecimento, restaurando o estado da transação ao que era quando o ponto de salvamento foi estabelecido. [1] [2] [3]

Parâmetros

nome_do_ponto_de_salvamento

O nome a ser dado ao novo ponto de salvamento.

Observações

Para desfazer até o ponto de salvamento deve ser utilizado o comando [ROLLBACK TO SAVEPOINT](#). Para destruir um ponto de salvamento, mantendo os efeitos dos comandos executados após este ter sido estabelecido, deve ser utilizado o comando [RELEASE SAVEPOINT](#).

Os pontos de salvamento somente podem ser estabelecidos dentro de um bloco de transação. Podem haver vários pontos de salvamento definidos dentro de uma transação.

Exemplos

Para estabelecer um ponto de salvamento e, posteriormente, desfazer o efeito de todos os comandos executados após o seu estabelecimento:

```
BEGIN;
    CREATE TEMPORARY TABLE tabela1 (col1 int) ON COMMIT DROP;
    INSERT INTO tabela1 VALUES (1);
    SAVEPOINT meu_ponto_de_salvamento;
    INSERT INTO tabela1 VALUES (2);
    ROLLBACK TO SAVEPOINT meu_ponto_de_salvamento;
    INSERT INTO tabela1 VALUES (3);
    SELECT * FROM tabela1;
COMMIT;

col1
-----
 1
 3
(2 linhas)
```

A transação acima inseriu os valores 1 e 3, mas não o 2.

Para estabelecer e, posteriormente, destruir um ponto de salvamento:

```
BEGIN;
    CREATE TEMPORARY TABLE tabela1 (col1 int) ON COMMIT DROP;
    INSERT INTO tabela1 VALUES (3);
    SAVEPOINT meu_ponto_de_salvamento;
    INSERT INTO tabela1 VALUES (4);
    RELEASE SAVEPOINT meu_ponto_de_salvamento;
    SELECT * FROM tabela1;
COMMIT;

col1
-----
 3
 4
(2 linhas)
```

A transação acima inseriu tanto o 3 quanto o 4.

Compatibilidade

O padrão SQL requer que um ponto de salvamento seja destruído, automaticamente, quando é estabelecido um outro ponto de salvamento com o mesmo nome. No PostgreSQL o ponto de salvamento é mantido, embora somente o mais recente seja utilizado ao se desfazer ou liberar; a liberação do ponto de salvamento mais recente torna o ponto de salvamento mais antigo acessível novamente para os comandos ROLLBACK TO SAVEPOINT e RELEASE SAVEPOINT. Fora isso, o comando SAVEPOINT está em conformidade total com o padrão SQL.

Consulte também

[BEGIN](#), [COMMIT](#), [RELEASE SAVEPOINT](#), [ROLLBACK](#), [ROLLBACK TO SAVEPOINT](#)

Referências

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

<http://pgdocptbr.sourceforge.net/pg80/transaction-iso.html>

<http://pgdocptbr.sourceforge.net/pg80/sql-set-transaction.html>

<https://www.postgresql.org/docs/9.6/static/transaction-iso.html>

<https://www.postgresql.org/docs/9.6/static/sql-set-transaction.html>

Transações

As transações têm como objetivo evitar inconsistências em alguns tipos de operações, quando um bloco de comandos é efetivado ou inteiramente cancelado em bloco. Para a execução das transações temos os comandos:

COMMIT

ROLLBACK

SAVEPOINT

COMMIT - efetiva todos os comandos do bloco

ROLLBACK - cancela todos os comandos

SAVEPOINT - salva todos os comandos somente até onde se encontra o SAVEPOINT

Trabalhando com Nulos

Como a presença de NULLs geralmente torna mais frágil um modelo de dados, como também ferem os princípios do modelo relacional e é um assunto pouco debatido, resolvi estudar e experimentar o uso do NULL em várias consultas para conhecer melhor seu comportamento. Este não é um assunto exclusivo do SGBD PostgreSQL, mas que diz respeito a todos os SGBDs.

Expondo e analisando diversos cenários onde o NULL aparece e pode complicar a nossa vida: em foreign keys, agregações, distinct, etc. Como também mostrando alternativas ao seu uso. O conhecimento do comportamento do NULL nos leva a viver melhor com ele.

O NULL é como a SQL lida com valores inexistentes, desconhecidos, não aplicáveis ou perdidos.

NULL é global em termos de tipos de dados e não se restringe a um único tipo de dados.

Valor Padrão

Quando criamos uma tabela e em um determinado campo não adicionamos nenhuma constraint, por padrão o SQL dos principais SGBDs (inclusive o PostgreSQL) adiciona NULL como default.

De forma que quando se insere registro nesta tabela e não se adiciona nenhum valor neste campo, será inserido o valor NULL. Geralmente ao usar assim na inclusão:

```
create table nulos(chave serial primary key, nulo int);
INSERT INTO nulos (chave, nulo) VALUES (1, 1);
INSERT INTO nulos (chave, nulo) VALUES (2, NULL);
INSERT INTO nulos (chave, nulo) VALUES (3, DEFAULT);
INSERT INTO nulos (nulo) VALUES (DEFAULT);
```

Em todos estes exemplos será adicionado NULL no campo nulo.

NULL se Propaga

Regra geral: NULL se propaga, o que significa que com quem NULL se combina o resultado será um NULL, apenas com uma exceção, para FALSE.

O Que NULL não é:

- NULL não é zero,
- Não é string vazia
- Nem string de comprimento zero.
- Não é zero
- Não é vazio

Para evitar confusão recomenda-se usar um flag char(1).

Com valores como 'T' e 'F' ou 't' e 'f'.

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 se concatene com NULL gerará NULL, com exceção de FALSE, que gerará FALSE.

STRING || NULL -- NULL

Usos:

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

Tabelas Verdade**AND**

AND	TRUE	UNKNOWN	FALSE
TRUE	TRUE	UNKNOWN	FALSE
UNKNOWN	UNKNOWN	UNKNOWN	FALSE
FALSE	FALSE	FALSE	FALSE

OR

OR	TRUE	UNKNOWN	FALSE
TRUE	TRUE	TRUE	TRUE
UNKNOWN	TRUE	UNKNOWN	UNKNOWN
FALSE	TRUE	UNKNOWN	FALSE

NOT

<EXP>	NOT
TRUE	FALSE
UNKNOWN	UNKNOWN
FALSE	TRUE

Nessas tabelas UNKNOWN significa TRUE ou FALSE mas não se sabe qual dos dois. Caso se remova o UNKNOWN dessas tabelas elas voltam a ser tabelas boolean normais.

Tabela verdade com operador IS

IS	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	FALSE
FALSE	FALSE	TRUE	FASLE
UNKNOWN	FALSE	FALSE	TRUE

Experimentando:

```
create table booleanos(texto text, valor boolean);
```

```
insert into booleanos(texto, valor) values ('TRUE', TRUE);
insert into booleanos(texto, valor) values ('FALSE', FALSE);
insert into booleanos(texto, valor) values ('NULL', NULL);
```

```
select texto,valor from booleanos;
select * from booleanos where valor = true;
select * from booleanos where valor = false;
select * from booleanos where valor is null;
```

```

select * from booleanos where valor = true OR valor=false;
select * from booleanos where valor = true OR valor=false OR valor is null;

select * from booleanos where valor in( select valor from booleanos);
select * from booleanos where valor in( select valor from booleanos where valor=true or
valor=false);
select * from booleanos where valor in( select valor from booleanos where valor=true or valor
is null);
select * from booleanos where valor in( select valor from booleanos where valor=false or valor is
null);
select * from booleanos where valor in( select valor from booleanos where valor is null);

select (false or false);
select (false or true);
select (true or true);
select (true or null);
select (true and null);
select (true and not null);
select (false or null);
select (false or not null);
select (false and null);
select (null or null);
select (null and null);
select (null or not null);
select (null and not null);

```

A comparação com NULL

resulta num estado *desconhecido* chamado UNKNOWN.

Por exemplo, a expressão SQL

"Cidade = 'Porto Alegre'"

retorna FALSE para um registro contendo "Rio de Janeiro" no campo Cidade,
mas retorna UNKNOWN para um registro contendo NULL no mesmo campo.

Usando a lógica ternária, o SQL consegue usar UNKNOWN para resolver expressões
booleanas. Considerando a expressão

"Cidade = 'Porto Alegre' OR Balanco < 0.0".

Essa expressão retorna TRUE para qualquer registro contendo um valor negativo no
campo Balanco.

A mesma expressão retorna TRUE para qualquer registro contendo "Porto Alegre" no
campo Cidade.

Já FALSE é retornado somente para um registro contendo explicitamente uma cadeia
diferente de "Porto Alegre" e cujo campo Balanco é explicitamente não negativo.

Em qualquer outro caso, o retorno é UNKNOWN.

Na linguagem de manipulação de dados do SQL, um retorno TRUE duma expressão
inicia uma ação, enquanto UNKNOWN ou FALSE não iniciam ações.

Dessa forma a lógica ternária é transformada em binária para o utilizador.

Sobre NULLs:

- **NULLs** acarretam sérias dificuldades e devem ser evitados.
- Uma relação que contém nulos não é uma relação. (Date)
- Nulos são um erro e nunca deveriam ter sido adotados. (Date)
- A presença de nulls é corrigida com a normalização e consequente criação de outras tabelas.

FK (Chave Estrangeira) - permite nulos, mas se um campo for nulo estará satisfeita a constraint em consequência em consequência violada a integridade.

Recomendação: sempre usar NOT NULL nos campos da FK.

Comparar NULL com outro NULL

Requer operador especial, o operador IS e não podem ser comparados com os operadores =, >, <, >=, etc, pois retornará UNKNOWN.

Teste com Null

```
create table nula2(c1 int primary key, c2 int check(c2 > 0), c3 int);
insert into nula2(c1,c2,c3) values (1,default,4); -- Será válido.
insert into nula2(c1,c2,c3) values (2,-3,4); -- Não válido
```

```
select * from nula2;
c1 | c2 | c3
----+---+---
1 |  | 4
(1 registro)
```

Uma "incoerência" no comportamento do nulo, que reforça a recomendação de se evitar seu uso.

Veja que o campo c2, como permite null, aceitou o valor NULL, mesmo com a check > 0.

Uso de NULL em Alguns Cenários

Com Chaves Estrangeiras:

FK (para relacionamentos 1 – N ou N – 1)

```
create table clientes(cpf char(11) primary key, nome char(45)not null );
```

```
create table pedidos(
produto int primary key,
valor numeric(12,2) not null,
data date not null,
cpf_cliente char(11),
constraint cpf_fk foreign key (cpf_cliente) references clientes(cpf)
);
```

```
insert into clientes(cpf, nome) values ('12345678901', 'João Pereira Brito');
```

```
insert into pedidos(produto, valor, data, cpf_cliente) values (1, 37.45, '2008-04-26',
'12345678901');
insert into pedidos(produto, valor, data, cpf_cliente) values (2, 87.45, '2008-04-27',
DEFAULT);
```

```
select * from pedidos;
produto | valor | data | cpf_cliente
-----+-----+-----+
 1 | 37.45 | 2008-04-26 | 12345678901
 2 | 87.45 | 2008-04-27 |
(2 registros)
```

Nesse caso espera-se normalmente que pedidos sejam cadastrados somente para clientes já cadastrados, mas se for permitida a FK como NULL isso não é garantido.

NOT IN

Do Artigo: NULLs vs. NOT IN()

```
select * from tabela where id not in (select campo from tabela2);
```

Mesmo com alguns ids que não encontram-se na tabela2, ainda assim nada é retornado.

```
CREATE TABLE objects (id INT4 PRIMARY KEY);
```

```
CREATE TABLE secondary (object_id INT4);
```

```
INSERT INTO objects (id) VALUES (1), (2), (3);
```

```
INSERT INTO secondary (object_id) VALUES (NULL), (2), (4);
```

```
SELECT * FROM objects WHERE id NOT IN (SELECT object_id FROM secondary);
```

```
SELECT id, id NOT IN (SELECT object_id FROM secondary) FROM objects;
```

```
SELECT id, id NOT IN (SELECT object_id FROM secondary WHERE object_id IS NOT
NULL) FROM objects;
```

COUNT

O count(*) conta com NULLs.
COUNT(campo) não conta com nulls.

```
create table nulos(texto text not null, campo int);
insert into nulos(texto, campo) values ('campo1', 1);
insert into nulos(texto, campo) values ('campo2', DEFAULT);
insert into nulos(texto, campo) values ('campo3', 3);
insert into nulos(texto, campo) values ('campo4', NULL);

select count(*) from nulos
select count(texto) from nulos
select count(campo) from nulos
```

SUM

```
select * from nulos
select sum(campo) from nulos
```

AVG

```
select * from nulos
select avg(campo) from nulos
```

MAX

```
select * from nulos
select max(campo) from nulos
```

MIN

```
select * from nulos
select min(campo) from nulos
```

SUM, AVG, MAX, MIN não consideram os NULLs.

GROUP BY

```
select count(*) from nulos group by texto
select count(*),campo from nulos group by campo; -- agrupa campo 1 e 3 (nulos)
```

ORDER BY

```
select * from nulos order by campo;
select * from nulos order by campo desc;
```

Obs.: os NULL são os maiores, os primeiros quando na ordem DESC.

DISTINCT

```
select distinct(texto) from nulos;
select distinct(campo) from nulos; -- Traz inclusive os nulos, mas uma só vez
select distinct(count(campo)) from nulos; -- Como o count(campo) não traz nulos, retornará somente 2
```

NULL Se Propaga:

- Operações aritméticas com NULL gera NULL
- Operações de strings com NULL gera NULL
- Operações booleans com NULL gera NULL com TRUE e FALSE com FALSE

Exemplos de uso:

- Cadastro de funcionários, campo dependente
- Cadastro de exames laboratorias, campo do diagnóstico

Do Artigo: None, nil, Nothing, undef, NA, and SQL NULL

```
=> -- aggregate with one NULL input
=> select sum(column1) from (values(NULL::int)) t;
sum
-----
```

(1 row)

```
=> -- aggregate with two inputs, one of them NULL
=> select sum(column1) from (values(1),(NULL)) t;
sum
-----
1
(1 row)
```

```
=> -- aggregate with no input
=> select sum(column1) from (values(1),(NULL)) t where false;
sum
-----
```

```
(1 row)
```

```
=> -- + operator
=> select 1 + NULL;
?column?
-----
(1 row)
```

Boleanos

```
create table booleanos(entrada text, saida boolean);
insert into booleanos values('t','t'),('true','true'),('TRUE','TRUE'),('TRUEs/aspas',TRUE),
('y','y'), ('yes','yes'), ('1','1');
insert into booleanos values('f','f'),('false','false'),('FALSE','FALSE'),('FALSEs/aspas',FALSE),
('n','n'), ('no','no'), ('0','0');
```

Caso use o psql a consulta abaixo retornará tudo t ou f na saída. Já o pgadmin mostrará tudo TRUE ou FALSE.

```
select * from booleanos;
```

O mais interessante é perguntar ao psql se tem algum TRUE por lá e ele responder que sim:

```
testes=# select * from booleanos where saida='TRUE';
entrada | saida
-----+-----
t | t
true | t
TRUE | t
```

```
TRUEs/aspas | t
y | t
yes | t
1 | t
(7 rows)
```

Então, agora de fato percebi que o psql (realmente, acho mais confiável e acabo confundindo com o próprio PG), o psql é só um cliente e pode exibir de uma forma, o pgadmin de outra e pelo visto qualquer uma das formas de entrada pode ser vista na saída.

Como Evitar NULLs

Campos como telefone, CPF, Inscrição Estadual e outros são criados permitindo NULL pelo fato de que esses não são obrigatórios para todos os registros.

Nesses casos, quando um cliente não tem telefone, o campo é deixado sem valor (NULL) gerando as várias dificuldades decorrentes do NULL.

Algumas providências ajudam a reduzir esses problemas. Podemos definir uma entrada para o caso do cliente não ter telefone, como por exemplo a palavra 'sem'. Todo cliente que não tenha telefone recebe a entrada 'sem'.

Com isso podemos criar um domínio que valide o campo telefone com expressões regulares.

Também podemos definir um índice parcial e único que será aplicado somente para as entradas válidas de telefone (exceto as entradas com 'sem') e usar este índice no domínio.

Assim podemos proceder para praticamente todos os casos onde haja necessidade de se deixar como opcional algum campo.

Outro exemplo Prático

Temos uma tabela de login e nela um campo expira_em para abrigar a data em que o login do usuário expira.

Acontece que alguns usuários nunca expiram, portanto para estes o campo conterá sempre NULL.

Esta é uma saída e pelo visto a mais cômoda, portanto com tendência de ser a mais adotada por quem tem pressa.

Uma alternativa para evitar usar NULL seria criar uma segunda tabela, somente para os usuários que não expiram.

```
create table login_expira
(
    login char(12) primary key,
    senha char(32) not null,
    expira_em date not null
);
create table login_nao_expira
(
    login char(12) primary key,
    senha char(32) not null
);
```

Criar então, para facilitar a inclusão, uma função que insere condicionalmente.

```
-- Quando não expira entrar com NULL para a data
create or replace function insere_login(text, text, date) returns void as
$$
begin
if ($3 isnull) then
    insert into login_nao_expira(login, senha) values ($1, $2);
else
    insert into login_expira(login, senha, expira_em) values ($1, $2, $3);
end if;
return;
end;
$$
language 'plpgsql';
```

```
-- Quando não expirar entrar NULL para data
select insere_login('ribafs', 'senha',NULL);
select insere_login('login2', 'senha2','2008-12-31');

select * from login_nao_expira;
select * from login_expira;
```

Neste caso a aplicação não cadastrará os registros diretamente nas tabelas mas indiretamente através desta função.

Referências:

- Livro Instant SQL Programming de Joe Celko.
- Wikipédia – http://pt.wikipedia.org/wiki/Lógica_ternária
- Artigo - [None, nil, Nothing, undef, NA, and SQL NULL](#)
- Artigo - [NULLs vs. NOT IN\(\)](#)

Valores Nulos

Nulo interagindo com qualquer valor usando qualquer operação resulta nulo.

NULL não é = NULL

COALESCE - função que retorna o primeiro de seus argumentos que não é nulo.
Será retornado NULL somente se todos os argumentos forem nulos.

select coalesce (nome, descricao, '(nenhum')...

descricao somente será avaliada se nome for null.

Caso nome não seja null ele será retornado e os demais ignorados.

- Testa nome, se \neq null o retorna e para
- Se nome for NULL testa descricao. Se descricao \neq null retorna descricao e para
- Se descricao for NULL retorna nenhum.

NULLIF - retorna NULL se valor1 for igual a valor2, caso contrário retorna valor1

Pode ser usado para executar a operação inversa da COALESCE.

Exemplo

select nullif (valor1, '(nada')...

Se valor1 = nada retorna NULL

Se \neq retorna valor1

GREATEST e LEAST - funções que selecionam o maior e o menor valor de uma lista

de qualquer quantidade de expressões.

Valores nulos nas listas serão ignorados. O resultado será nulo somente se todas as expressões forem nulas.

Utilizando Operadores

- 1) Introdução aos operadores - 1
- 2) Entendendo os Operadores de texto - 4
- 3) Entendendo as Expressões regulares - 5
- 4) Entendendo os Operadores matemáticos - 10
- 5) Entendendo a importância da Conversão de tipos - 11

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";
expressão
-----
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 (decrescente)

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

Operador/Elemento	Associatividade	Descrição
		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>

2) Entendendo os Operadores de Texto (Strings)

Funções e operadores para cadeias de caracteres (Strings)

Esta seção descreve as funções e operadores disponíveis para examinar e manipular valores cadeia de caracteres. Neste contexto, cadeia de caracteres inclui todos os valores dos tipos character, character varying e text. A menos que seja dito o contrário, todas as funções relacionadas abaixo trabalham com todos estes tipos, mas se deve tomar cuidado com os efeitos em potencial do preenchimento automático quando for utilizado o tipo character. De modo geral, as funções descritas nesta seção também trabalham com dados de tipos que não são cadeias de caracteres, convertendo estes dados primeiro na representação de cadeia de caracteres. Algumas funções também existem em forma nativa para os tipos cadeia de bits.

O SQL define algumas funções para cadeias de caracteres com uma sintaxe especial, onde certas palavras chave, em vez de vírgulas, são utilizadas para separar os argumentos. Os detalhes estão na [Tabela 9-6](#). Estas funções também são implementadas utilizando a sintaxe regular de chamada de função (Consulte a [Tabela 9-7](#)).

Tabela 9-6. Funções e operadores SQL para cadeias de caracteres: vide site oficial em:

<http://pgdocptbr.sourceforge.net/pg80/functions-string.html>

<http://www.postgresql.org/docs/8.3/interactive/functions-string.html>

Exemplo 9-1. Conversão de letras minúsculas e maiúsculas acentuadas

Abaixo estão mostradas duas funções para conversão de letras. A função maiusculas converte letras minúsculas, com ou sem acentos, em maiúsculas, enquanto a função minusculas faz o contrário, ou seja, converte letras maiúsculas, com ou sem acentos em minúsculas [1] .

```
=> \!chcp 1252
Active code page: 1252
=> CREATE FUNCTION maiusculas(text) RETURNS text AS '
'>     SELECT translate( upper($1),
'>         text ''áéíóúàèìòùãõâéîôõäëïöç'',
'>         text ''ÁÉÍÓÚÀÈÌÒÙÃÕÂÉÎÔÕÄËÏÖÜÇ'')
'> ' LANGUAGE SQL STRICT;

=> SELECT maiusculas('à ação seqüência');

    maiusculas
-----
À AÇÃO SEQÜÊNCIA
=> CREATE FUNCTION minusculas(text) RETURNS text AS '
'>     SELECT translate( lower($1),
'>         text ''ÁÉÍÓÚÀÈÌÒÙÃÕÂÉÎÔÕÄËÏÖÜÇ'',
'>         text ''áéíóúàèìòùãõâéîôõäëïöç'')
'> ' LANGUAGE SQL STRICT;

=> SELECT minusculas('À AÇÃO SEQÜÊNCIA');

    minusculas
-----
à ação seqüência
```

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'	<i>verdade</i>
'abc' LIKE 'a%'	<i>verdade</i>
'abc' LIKE '_b_'	<i>verdade</i>
'abc' LIKE 'c'	<i>falso</i>

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 \sim equivale ao LIKE, enquanto \sim^* corresponde ao ILIKE.

Também existem os operadores $\sim\sim$ e $\sim\sim^*$, 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 (s o compar veis ao . e ao .* das express es regulares POSIX).

Al m destas funcionalidades tomadas emprestada do LIKE, o **SIMILAR TO** suporta os seguintes metacaracteres para correspond ncia com p drio pegos emprestado das express es regulares POSIX:

- | representa altern ncia (uma das duas alternativas).
- * representa a repeti o do item anterior zero ou mais vezes.
- + representa a repeti o do item anterior uma ou mais vezes.
- Os par nteses () podem ser utilizados para agrupar itens em um  nico item l gico.
- A express o de colchetes [...] especifica uma classe de caracteres, do mesmo modo que na express o regular POSIX.

Deve ser observado que as repeti es limitadas (? e {...}) n o est o dispon veis, embora existam no POSIX. Al m disso, o ponto (.) n o  o 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 cl usula 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 fun o substring com tr s par metros, substring(cadeia_de_caracteres FROM p drio FOR caractere_de_escape), permite extrair a parte da cadeia de caracteres que corresponde ao p drio da express o regular SQL:1999. Assim como em SIMILAR TO, o p drio especificado deve corresponder a toda a cadeia de caracteres, sen o a fun o falha e retorna nulo. Para indicar a parte do p drio que deve ser retornada em caso de sucesso, o p drio deve conter duas ocorr ncias do caractere de escape seguidas por aspas ("").  e retornado o texto correspondente   parte do p drio 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 seqüência 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 (aquele 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
```

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	Exempl o	Resultad o
+	adição	2 + 3	5
-	subtração	2 - 3	-1
*	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

Operador	Descrição	Exemplo	Resultado
!!	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>

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/analisador (*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 sobre carregados 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 despender 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.

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 ('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 attypmod; a interpretação de attypmod 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.

Tipos de Operadores

== Operadores Aritméticos

Mathematical Operators

Operator	Description	Example	Result
+	addition	2 + 3	5
-	subtraction	2 - 3	-1
*	multiplication	2 * 3	6
/	division	4 / 2	2
%	modulo	5 % 4	1
^	exponentiation	2.0 ^ 3.0	8
/	square root	/ 25.0	5
/	cube root	/ 27.0	3
!	factorial	5 !	120
!!	factorial	!! 5	120
@	absolute value	@ -5.0	5
&	bitwise AND	91 & 15	11
	bitwise OR	32 3	35
#	bitwise XOR	17 # 5	20
~	bitwise NOT	~1	-2
<<	bitwise shift left	1 << 4	16
>>	bitwise shift right	8 >> 2	2

Precedência controlada com parêntesis, como em matemática.

== Operadores de Comparação

```
=
>
>=
<
<=
!= ou <>
```

Usados na cláusula WHERE

Todos estão disponíveis para todos os tipos de dados

Operador BETWEEN

a between x and y;

a NOT BETWEEN x AND y

Operador IN

na lista

```
where cor in('verde', 'vermelho', 'azul', 'preto');
```

Operador LIKE

Recuperar valores parecidos com a string de pesquisa quando você não sabe o valor exato.

LIKE '%975';

Pode retornar, se existirem:

1975

2975

3975

...

9975

LIKE 'A%' - começa com A

'%A%' - Contém A em qualquer posição

'_A%' - Contém A na segunda posição

'A%' - Termina com A

'A%B%C' - Começa com A, tem B em qualquer posição e termina com C

NOT LIKE '%A' - Não contém A no início

Operador ILIKE - busca case sensitiva.

Operador SIMILAR TO

Semelhante ao LIKE mas com algumas facilidades a mais.

% e - semelhante ao LIKE

[A-F] - de A a F

[AEF] - A, E ou F

[^AEF] - Qualquer coisa diferente de A, E ou F

| e || - ou é concatenação

Operador IS NULL

Trata de campos com valor NULL

Como NULL não é uma string vazia, nem é zero nem podemos usar com = para tratar com NULL precisamos usar IS NULL.

```
select nome, nascimento from servidores where nascimento is null;
```

Não podemos usar ... nascimento = null;

Nem ... nascimento = 0;

Nem ... nascimento = ";

Operadores Lógicos

AND, OR e NOT

Tabela Verdade

a	b	a AND b	a OR b
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

a	NOT a
TRUE	FALSE
FALSE	TRUE
NULL	NULL

21 - Catálogo do Sistema

- 1) Tabelas de Sistema
- 2) Informações sobre as tabelas de sistema
- 3) Exemplos práticos

Todo SGBD precisa ter seu catálogo de sistema (metadados, dicionário de dados), onde armazena pelo menos:

- Nomes das tabelas
- Nomes dos campos
- Tipos de dados de cada campo
- As constraints
- Informações sobre os índices
- Privilégios de acesso dos elementos

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)

Exibir registro aleatoriamente

```

SELECT *
FROM mctvaramb
ORDER BY random()
limit 1
  
```

Fonte: <http://pgviavel.blogspot.com/search/label/fun%C3%A7%C3%B5es>

O Catálogo de sistema está disponível no PostgreSQL desde a versão 7.4.

1) Tabelas de Sistema do PostgreSQL 8.3

Catalog Name	Purpose
pg_aggregate	aggregate functions
pg_am	index access methods
pg_amop	access method operators
pg_amproc	access method support procedures
pg_attrdef	column default values
pg_attribute	table columns ("attributes")

Catalog Name	Purpose
pg_authid	authorization identifiers (roles)
pg_auth_members	authorization identifier membership relationships
pg_autovacuum	per-relation autovacuum configuration parameters
pg_cast	casts (data type conversions)
pg_class	tables, indexes, sequences, views ("relations")
pg_constraint	check constraints, unique constraints, primary key constraints, foreign key constraints
pg_conversion	encoding conversion information
pg_database	databases within this database cluster
pg_depend	dependencies between database objects
pg_description	descriptions or comments on database objects
pg_enum	enum label and value definitions
pg_index	additional index information
pg_inherits	table inheritance hierarchy
pg_language	languages for writing functions
pg_largeobject	large objects
pg_listener	asynchronous notification support
pg_namespace	schemas
pg_opclass	access method operator classes
pg_operator	operators
pg_opfamily	access method operator families
pg_pltemplate	template data for procedural languages
pg_proc	functions and procedures
pg_rewrite	query rewrite rules
pg_shdepend	dependencies on shared objects
pg_shdescription	comments on shared objects
pg_statistic	planner statistics
pg_tablespace	tablespaces within this database cluster
pg_trigger	triggers
pg_ts_config	text search configurations
pg_ts_config_map	text search configurations' token mappings
pg_ts_dict	text search dictionaries
pg_ts_parser	text search parsers
pg_ts_template	text search templates
pg_type	data types

Mais detalhes em:

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

<http://pgdocptbr.sourceforge.net/pg80/catalogs.html>

http://www.cs.umu.se/kurser/TDBC86/H06/Slides/16_OH_catalog.pdf (trazendo informações sobre o catálogo do PostgreSQL, Oracle e ODBC).

É um conjunto de tabelas e views que armazenam informações sobre os bancos de dados.

Cada banco criado tem seu catálogo e existem algumas tabelas que guardam informações sobre todos os bancos, como o pg_database.

Todo SGBD precisa ter seu catálogo de sistema (metadados, dicionário de dados), onde armazena pelo menos:

- Nomes das tabelas
- Nomes dos campos
- Tipos de dados de cada campo
- As constraints
- Informações sobre os índices
- Privilégios de acesso dos elementos

Informações sobre as tabelas de sistema

Para visualizar a estrutura de cada uma destas relações, podemos usar o psql. Exemplo:
\d pg_database

Para visualizar o catálogo completo, que está no esquema 'pg_catalog', usar:
\dS

Para visualizar a estrutura de uma das relações do esquema pg_catalog:
\d pg_catalog.pg_table

2) Informações sobre as tabelas de sistema

Para visualizar a estrutura de cada uma destas relações, podemos usar o psql.
Exemplo:

\d pg_database

Observar que existe um esquema ()

Para visualizar o catálogo completo, que está no esquema 'pg_catalog', usar:

```
\dS
```

Um total de 74 na versão 8.3 do PostgreSQL.

Para visualizar a estrutura de uma das relações do esquema pg_catalog:

```
\d pg_catalog.pg_table
```

serão listados os campos, seus tipos e abaixo uma definição de view.

Retornando todas as tabelas de sistema de um usuário:

```
select * from pg_catalog.pg_tables where tableowner='dba1';
```

```
select * from pg_catalog.pg_tables where tableowner='postgres';
```

3) Exemplos práticos e úteis

Uso do disco pela Tabela

```
\c dba_projeto
```

```
vacuum analyze clientes;
```

```
SELECT relfilename, relpages FROM pg_class WHERE relname = 'clientes';
```

relfilename é o arquivo, com nome usando números num diretório do 'base'.

relpages é o número de páginas ocupado pela relação (tabela). Lembrar que cada página ocupa 8KB. relpages somente é atualizado por VACUUM, ANALYZE e uns poucos comandos de DDL como CREATE INDEX). O valor de relfilename possui interesse caso se deseje examinar diretamente o arquivo em disco da tabela.

Podemos então saber algo com mais detalhes assim:

```
\c dba_projeto
```

```
vacuum analyze clientes;
```

```
SELECT relfilename AS arquivo, relpages*8 AS tamanho_em_kb FROM pg_class WHERE relname = 'clientes';
```

No diretório do tutorial existe o arquivo [syscat.sql](#) contendo várias consultas interessantes aos catálogos do sistema:

<http://pgdocptbr.sourceforge.net/pg80/syscat.sql>

```
-- syscat.sql-
-- Exemplos de consultas aos catálogos do sistema

-- Portions Copyright (c) 1996-2003, PostgreSQL Global Development Group
-- Portions Copyright (c) 1994, Regents of the University of California

-- Primeiro definir o caminho de procura do esquema como pg_catalog,
-- para não ser necessário qualificar todo objeto do sistema.
--

SET SEARCH_PATH TO pg_catalog;

--

-- Listar o nome de todos os administradores de banco de dados e o seus bancos
de dados.
--

SELECT usename, datname
  FROM pg_user, pg_database
 WHERE usesysid = datdba
 ORDER BY usename, datname;

--

-- Listar todas as classes definidas pelo usuário
--

SELECT n.nspname, c.relname
  FROM pg_class c, pg_namespace n
 WHERE c.relnamespace=n.oid
   AND c.relkind = 'r'          -- sem índices, visões, etc
   AND n.nspname not like 'pg\%_%'    -- sem catálogos
   AND n.nspname != 'information_schema' -- sem information_schema
 ORDER BY nspname, relname;

-- Listar todos os índices simples (ou seja, àqueles que são definidos
-- sobre uma referência de coluna simples)
--

SELECT n.nspname AS schema_name,
       bc.relname AS class_name,
       ic.relname AS index_name,
       a.attname
  FROM pg_namespace n,
       pg_class bc,        -- classe base
       pg_class ic,        -- classe índice
       pg_index i,
       pg_attribute a      -- atributo na base
 WHERE bc.relnamespace = n.oid
   AND i.indrelid = bc.oid
   AND i.indexrelid = ic.oid
```

```

AND i.indkey[0] = a.attnum
AND i.indnatts = 1
AND a.attrelid = bc.oid
ORDER BY schema_name, class_name, index_name, attname;

-- 
-- Listar os atributos definidos pelo usuário e seus tipos
-- para todas as classes definidas pelo usuário
--
SELECT n.nspname, c.relname, a.attname, format_type(t.oid, null) AS typname
FROM pg_namespace n, pg_class c,
     pg_attribute a, pg_type t
WHERE n.oid = c.relnamespace
    AND c.relkind = 'r'          -- sem índices
    AND n.nspname not like 'pg\%_%' -- sem catálogos
    AND n.nspname != 'information_schema' -- sem information_schema
    AND a.attnum > 0           -- sem atributo de sistema
    AND not a.attisdropped      -- sem colunas removidas
    AND a.attrelid = c.oid
    AND a.atttypid = t.oid
ORDER BY nspname, relname, attname;

-- 
-- Listar todos os tipos base definidos pelo usuário (sem incluir os tipos matriz)
--
SELECT n.nspname, u.usename, format_type(t.oid, null) AS typname
FROM pg_type t, pg_user u, pg_namespace n
WHERE u.usesysid = t.typowner
    AND t.typnamespace = n.oid
    AND t.typeid = '0'::oid      -- sem tipos complexos
    AND t.typelem = '0'::oid     -- sem matrizes
    AND n.nspname not like 'pg\%_%' -- sem catálogos

    AND n.nspname != 'information_schema' -- sem information_schema
ORDER BY nspname, usename, typname;

-- 
-- Listar todas as funções de agregação e os tipos em que podem ser aplicadas
--
SELECT n.nspname, p.proname, format_type(t.oid, null) AS typname
FROM pg_namespace n, pg_aggregate a,
     pg_proc p, pg_type t
WHERE p.pronamespace = n.oid
    AND a.aggfnoid = p.oid
    AND p.proargtypes[0] = t.oid
ORDER BY nspname, proname, typname;

-- 
-- Restaurar o caminho de procura
--
RESET SEARCH_PATH;

```

Retornar o número de usuários conectados

```
select count(*) from pg_stat_activity;
select count(*) from pg_stat_database;
```

`pg_stat_database` que apresenta para cada banco de dados o número de conexões.
Fica mais fácil de visualizar do que o `pg_stat_activity` quando se tem muitas conexões.

Mostrar uso dos índices:

```
select * from pg_statio_user_indexes;
select * from pg_stat_user_indexes;
```

Mostra estatística de uso de todas as tabelas e manutenção:

```
select * from pg_stat_all_tables;
```

Mostra todas as tabelas e informações do atual esquema do atual banco:

```
select * from pg_stat_user_tables;
```

Veja só o retorno: `relid | schemaname | relname | seq_scan | seq_tup_read | idx_scan | idx_tup_fetch | n_tup_ins | n_tup_upd | n_tup_del | last_vacuum | last_autovacuum | last_analyze | last_autoanalyze`

A função `pg_stat_get_backend_idset` provê uma conveniente maneira de gerar um registro/linha para cada processo ativo no servidor. Por exemplo, para **exibir os PIDs e as atuais consultas de todos os processos do servidor**:

```
SELECT pg_stat_get_backend_pid(s.backendid) AS procpid,
       pg_stat_get_backend_activity(s.backendid) AS current_query
  FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

Visualizar os processos do postgresql num UNIX:

```
ps auxww | grep ^post
```

Mostrar todas as tabelas (inclusive de sistema) a quantidade de registros:

```
select relpages*8192 from pg_class;
```

Funções para Administração do Sistema

Definindo configurações

A função `current_setting` retorna o valor corrente da definição `nome_da_definição`.
Corresponde ao comando SQL `SHOW`. Por exemplo:

```
SELECT current_setting('datestyle');
```

A função `set_config` define o parâmetro `nome_da_configuração` como `novo_valor`. Se o parâmetro `é_local` for `true`, então o novo valor se aplica somente à transação corrente. Se for desejado que o novo valor seja aplicado à sessão corrente, deve ser utilizado `false`. Esta função corresponde ao comando SQL `SET`. Por exemplo:

```
SELECT set_config('log_statement_stats', 'off', false);
```

Funções de Sinais para o Servidor

```
pg_cancel_backend(pid)
pg_reload_conf()
pg_rotate_logfile()
```

Se for bem-sucedida a função retorna 1, caso contrário retorna 0. O ID do processo (`pid`) de um servidor ativo pode ser encontrado a partir da coluna `procpid` da visão `pg_stat_activity`, ou listando os processos do `postgres` no servidor através do comando do Unix `ps`.

Funções que Retornam o Tamanho de Objetos

Name	Return Type	Description
<code>pg_column_size(any)</code>	<code>int</code>	Number of bytes used to store a particular value (possibly compressed)
<code>pg_tablespace_size(oid)</code>	<code>bigint</code>	Disk space used by the tablespace with the specified OID
<code>pg_tablespace_size(name)</code>	<code>bigint</code>	Disk space used by the tablespace with the specified name
<code>pg_database_size(oid)</code>	<code>bigint</code>	Disk space used by the database with the specified OID
<code>pg_database_size(name)</code>	<code>bigint</code>	Disk space used by the database with the specified name
<code>pg_relation_size(oid)</code>	<code>bigint</code>	Disk space used by the table or index with the specified OID
<code>pg_relation_size(text)</code>	<code>bigint</code>	Disk space used by the table or index with the specified name. The table name may be qualified with a schema name
<code>pg_total_relation_size(oid)</code>	<code>bigint</code>	Total disk space used by the table with the specified OID, including indexes and toasted data

Name	Return Type	Description
pg_total_relation_size(text)	bigint	Total disk space used by the table with the specified name, including indexes and toasted data. The table name may be qualified with a schema name
pg_size.pretty(bigint)	text	Converts a size in bytes into a human-readable format with size units

pg_column_size shows the space used to store any individual data value.

pg_tablespace_size and pg_database_size accept the OID or name of a tablespace or database, and return the total disk space used therein.

pg_relation_size accepts the OID or name of a table, index or toast table, and returns the size in bytes.

pg_total_relation_size accepts the OID or name of a table or toast table, and returns the size in bytes of the data and all associated indexes and toast tables.

pg_size.pretty can be used to format the result of one of the other functions in a human-readable way, using kB, MB, GB or TB as appropriate.

Funções de Acesso a Arquivos

Name	Return Type	Description
pg_ls_dir(dirname text)	setof text	List the contents of a directory
pg_read_file(filename text, offset bigint, length bigint)	text	Return the contents of a text file
pg_stat_file(filename text)	record	Return information about a file

pg_ls_dir returns all the names in the specified directory, except the special entries ". " and "...".

pg_read_file returns part of a text file, starting at the given offset, returning at most length bytes (less if the end of file is reached first). If offset is negative, it is relative to the end of the file.

pg_stat_file returns a record containing the file size, last accessed time stamp, last modified time stamp, last file status change time stamp (Unix platforms only), file creation timestamp (Windows only), and a boolean indicating if it is a directory. Typical usages include:

```
SELECT * FROM pg_stat_file('filename');
```

```
SELECT (pg_stat_file('filename')).modification;
```

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

Consultando a Estrutura de uma Tabela através do Catálogo

SELECT

```
    rel.nspname, rel.relname, attrs.attname, "Type", "Default", attrs.attnotnull
```

FROM (

```
    SELECT c.oid, n.nspname, c.relname
```

```
    FROM pg_catalog.pg_class c
```

```
    LEFT JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace
```

```
    WHERE pg_catalog.pg_table_is_visible(c.oid) ) rel
```

JOIN (

```
    SELECT a.attname, a.attrelid, pg_catalog.format_type(a.atttypid, a.atttypmod)
```

as "Type",

```
    (SELECT substring(d.adsrc for 128) FROM pg_catalog.pg_attrdef d
```

```
    WHERE d.adrelid = a.attrelid AND d.adnum = a.attnum AND a.atthasdef)
```

as "Default",

a.attnotnull, a.attnum

```
    FROM pg_catalog.pg_attribute a WHERE a.attnum > 0 AND NOT a.attisdropped )
```

attrs

```
    ON (attrs.attrelid = rel.oid )
```

```
    WHERE relname = 'clientes' ORDER BY attrs.attnum;
```

Função em PIPgSQL:

```
CREATE OR REPLACE FUNCTION Dados_Tabela(varchar(30))
RETURNS SETOF tabela_estrutura AS '
DECLARE
r tabela_estrutura%ROWTYPE;
rec RECORD;
vTabela alias for $1;
eSql TEXT;
```

```
BEGIN
```

```

eSql := '''SELECT
    CAST(rel.nspname AS TEXT), CAST(rel.relname AS TEXT) ,
    CAST(attrs.attname AS TEXT), CAST("Type" AS TEXT),
    CAST("Default" AS TEXT), attrs.attnotnull
    FROM
        (SELECT c.oid, n.nspname, c.relname
        FROM pg_catalog.pg_class c
        LEFT JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace
        WHERE pg_catalog.pg_table_is_visible(c.oid) ) rel
    JOIN
        (SELECT a.attname, a.attrelid,
        pg_catalog.format_type(a.atttypid, a.atttypmod) as "Type",
        (SELECT substring(d.adsrc for 128) FROM
        pg_catalog.pg_attrdef d
        WHERE d.attrelid = a.attrelid AND d.adnum = a.attnum AND
        a.attasdef)
        as "Default", a.attnotnull, a.attnum
        FROM pg_catalog.pg_attribute a
        WHERE a.attnum > 0 AND NOT a.attisdropped ) attrs
    ON (attrs.attrelid = rel.oid )
    WHERE relname LIKE '%%' || vTabela || '%'
    ORDER BY attrs.attnum''';
FOR r IN EXECUTE eSql
LOOP
RETURN NEXT r;
END LOOP;
IF NOT FOUND THEN
    RAISE EXCEPTION ''Tabela % não encontrada'', vTabela;
END IF;
RETURN;
END
'
LANGUAGE 'plpgsql';

```

Usando:

```
SELECT * FROM Dados_Tabela('clientes');
```

Fonte:

http://imasters.uol.com.br/artigo/2137/postgresql/consultando_a_estrutura_de uma_tabela/

Retornando Informações sobre uma Tabela

```
SELECT pg_attribute.attnum AS index,
attname AS field,
typename AS type,
atttypmod-4 as length,
NOT attnotnull AS "null",
```

```
adsrc AS def
FROM pg_attribute,
pg_class,
pg_type,
pg_attrdef
WHERE pg_class.oid=attrelid
AND pg_type.oid=atttypid
AND attnum>0
AND pg_class.oid=adrelid
AND adnum=attnum
AND atthasdef='t'
AND lower(relname)='clientes'
UNION
SELECT pg_attribute.attnum AS index,
attname AS field,
typename AS type,
atttypmod-4 as length,
NOT attnotnull AS "null",
" AS def
FROM pg_attribute,
pg_class,
pg_type
WHERE pg_class.oid=attrelid
AND pg_type.oid=atttypid
AND attnum>0
AND atthasdef='f'
AND lower(relname)='clientes';
```

Use o comando \x no psql antes de executar a consulta caso queira exibir os registros em sequência, \x novamente para voltar ao normal.

Fonte:

http://imasters.uol.com.br/artigo/1283/postgresql/informacoes_atraves_do_catalogo_do_sistema/

Podemos listar tabelas, índices, sequências e vies usando os comandos do psql:

\d{t|i|s|v}

Abaixo veremos mais algumas funções que, ao contrário, usarão o SQL para receber essas informações.

Execute o trecho do script dba_projeto para criação do banco 'catalogo' da Aula 8.

Listando Tabelas do Banco Atual

```
SELECT relname
  FROM pg_class
 WHERE relname !~ '^(pg_|sql_)'
   AND relkind = 'r';

-- using INFORMATION_SCHEMA:

SELECT table_name
  FROM information_schema.tables
 WHERE table_type = 'BASE TABLE'
   AND table_schema NOT IN
     ('pg_catalog', 'information_schema');
```

Listando Views do Banco Atual

```
-- with postgresql 7.2:

SELECT viewname
  FROM pg_views
 WHERE viewname !~ '^pg_';

-- with postgresql 7.4 and later:

SELECT viewname
  FROM pg_views
 WHERE schemaname NOT IN
     ('pg_catalog', 'information_schema')
   AND viewname !~ '^pg_';

-- using INFORMATION_SCHEMA:

SELECT table_name
  FROM information_schema.tables
 WHERE table_type = 'VIEW'
   AND table_schema NOT IN
     ('pg_catalog', 'information_schema')
   AND table_name !~ '^pg_';

-- or
```

```
SELECT table_name
  FROM information_schema.views
 WHERE table_schema NOT IN ('pg_catalog', 'information_schema')
   AND table_name !~ '^pg_';
```

Listando Todos os Usuários

```
SELECT username
  FROM pg_user;
```

Retornando os nomes dos Campos de uma Tabela

```
SELECT a.attname
  FROM pg_class c, pg_attribute a, pg_type t
 WHERE c.relname = 'test2'
   AND a.attnum > 0
   AND a.attrelid = c.oid
   AND a.atttypid = t.oid;
```

-- Usando INFORMATION_SCHEMA:

```
SELECT column_name
  FROM information_schema.columns
 WHERE table_name = 'test2';
```

Informações Detalhadas sobre os Campos de uma Tabela

```
SELECT a.attnum AS ordinal_position,
       a.attname AS column_name,
       t.typname AS data_type,
       a.attlen AS character_maximum_length,
       a.atttypmod AS modifier,
       a.attnotnull AS notnull,
       a.atthasdef AS hasdefault,
       col_description(a.attrelid, a.attnum) as field_comment
  FROM pg_class c,
       pg_attribute a,
       pg_type t
 WHERE c.relname = 'test2'
   AND a.attnum > 0
   AND a.attrelid = c.oid
   AND a.atttypid = t.oid
 ORDER BY a.attnum;
```

-- Usando INFORMATION_SCHEMA:

```
SELECT ordinal_position,
       column_name,
       data_type,
       column_default,
       is_nullable,
       character_maximum_length,
       numeric_precision
  FROM information_schema.columns
```

```
WHERE table_name = 'test2'
ORDER BY ordinal_position;
```

Retornando os Nomes de Índices de uma Tabela

```
SELECT relname
  FROM pg_class
 WHERE oid IN (
    SELECT indexrelid
      FROM pg_index, pg_class
     WHERE pg_class.relname='test2'
       AND pg_class.oid=pg_index.indrelid
       AND indisunique != 't'
       AND indisprimary != 't'
);

```

Retornando as Constraints de uma Tabela

```
SELECT conname
  FROM pg_constraint, pg_class
 WHERE pg_constraint.conrelid = pg_class.oid
   AND relname = 'test2';

-- with INFORMATION_SCHEMA:

SELECT constraint_name, constraint_type
  FROM information_schema.table_constraints
 WHERE table_name = 'test2';
```

Recebendo Informações Detalhadas sobre as Constraints de uma Tabela

```
SELECT c.conname AS constraint_name,
       CASE c.contype
         WHEN 'c' THEN 'CHECK'
         WHEN 'f' THEN 'FOREIGN KEY'
         WHEN 'p' THEN 'PRIMARY KEY'
         WHEN 'u' THEN 'UNIQUE'
       END AS "constraint_type",
       CASE WHEN c.condeferrable = 'f' THEN 0 ELSE 1 END AS is_deferrable,
       CASE WHEN c.condeferred = 'f' THEN 0 ELSE 1 END AS is_deferred,
       t.relname AS table_name,
       array_to_string(c.conkey, ' ') AS constraint_key,
       CASE confupdtype
         WHEN 'a' THEN 'NO ACTION'
         WHEN 'r' THEN 'RESTRICT'
         WHEN 'c' THEN 'CASCADE'
         WHEN 'n' THEN 'SET NULL'
         WHEN 'd' THEN 'SET DEFAULT'
       END AS on_update,
       CASE confdeltype
         WHEN 'a' THEN 'NO ACTION'
         WHEN 'r' THEN 'RESTRICT'
         WHEN 'c' THEN 'CASCADE'
         WHEN 'n' THEN 'SET NULL'
         WHEN 'd' THEN 'SET DEFAULT'
       END AS on_delete,
       CASE confmatchtype
```

```

        WHEN 'u' THEN 'UNSPECIFIED'
        WHEN 'f' THEN 'FULL'
        WHEN 'p' THEN 'PARTIAL'
    END AS match_type,
    t2.relname AS references_table,
    array_to_string(c.confkey, ' ') AS fk_constraint_key
FROM pg_constraint c
LEFT JOIN pg_class t ON c.conrelid = t.oid
LEFT JOIN pg_class t2 ON c.confrelid = t2.oid
WHERE t.relname = 'testconstraints2'
    AND c.conname = 'testconstraints_id_fk';

-- with INFORMATION_SCHEMA:

SELECT tc.constraint_name,
       tc.constraint_type,
       tc.table_name,
       kcu.column_name,
       tc.is_deferrable,
       tc.initially_deferred,
       rc.match_option AS match_type,
       rc.update_rule AS on_update,
       rc.delete_rule AS on_delete,
       ccu.table_name AS references_table,
       ccu.column_name AS references_field
FROM information_schema.table_constraints tc
LEFT JOIN information_schema.key_column_usage kcu
    ON tc.constraint_catalog = kcu.constraint_catalog
    AND tc.constraint_schema = kcu.constraint_schema
    AND tc.constraint_name = kcu.constraint_name
LEFT JOIN information_schema.referential_constraints rc
    ON tc.constraint_catalog = rc.constraint_catalog
    AND tc.constraint_schema = rc.constraint_schema
    AND tc.constraint_name = rc.constraint_name
LEFT JOIN information_schema.constraint_column_usage ccu
    ON rc.unique_constraint_catalog = ccu.constraint_catalog
    AND rc.unique_constraint_schema = ccu.constraint_schema
    AND rc.unique_constraint_name = ccu.constraint_name
WHERE tc.table_name = 'testconstraints2'
    AND tc.constraint_name = 'testconstraints_id_fk';

```

Listando as Sequências

```

SELECT relname
  FROM pg_class
 WHERE relkind = 'S'
   AND relnamespace IN (
        SELECT oid
          FROM pg_namespace
         WHERE nspname NOT LIKE 'pg_%'
           AND nspname != 'information_schema'
);

```

Listando Todas as Triggers

```

SELECT trg.tgname AS trigger_name
  FROM pg_trigger trg, pg_class tbl
 WHERE trg.tgrelid = tbl.oid
   AND tbl.relname !~ '^pg_';
-- or
SELECT tgname AS trigger_name
  FROM pg_trigger
 WHERE tgname !~ '^pg_';

-- with INFORMATION_SCHEMA:

SELECT DISTINCT trigger_name
  FROM information_schema.triggers
 WHERE trigger_schema NOT IN
 ('pg_catalog', 'information_schema');

```

Listando Somente as Triggers de uma Tabela

```

SELECT trg.tgname AS trigger_name
  FROM pg_trigger trg, pg_class tbl
 WHERE trg.tgrelid = tbl.oid
   AND tbl.relname = 'newtable';
-- with INFORMATION_SCHEMA:

SELECT DISTINCT trigger_name
  FROM information_schema.triggers
 WHERE event_object_table = 'newtable'
   AND trigger_schema NOT IN
 ('pg_catalog', 'information_schema');

```

Recebendo Informações Detalhadas sobre as Triggers

```

SELECT trg.tgname AS trigger_name,
      tbl.relname AS table_name,
      p.proname AS function_name,
CASE trg.tgtype & cast(2 as int2)
    WHEN 0 THEN 'AFTER'
    ELSE 'BEFORE'
END AS trigger_type,
CASE trg.tgtype & cast(28 as int2)
    WHEN 16 THEN 'UPDATE'
    WHEN 8 THEN 'DELETE'
    WHEN 4 THEN 'INSERT'
    WHEN 20 THEN 'INSERT, UPDATE'
    WHEN 28 THEN 'INSERT, UPDATE, DELETE'
    WHEN 24 THEN 'UPDATE, DELETE'
    WHEN 12 THEN 'INSERT, DELETE'
END AS trigger_event
  FROM pg_trigger trg,
       pg_class tbl,
       pg_proc p
 WHERE trg.tgrelid = tbl.oid
   AND trg.tgfoid = p.oid
   AND tbl.relname !~ '^pg_';

```

```
-- with INFORMATION_SCHEMA:

SELECT *
  FROM information_schema.triggers
 WHERE trigger_schema NOT IN
   ('pg_catalog', 'information_schema');
```

Listar as Funções

```
SELECT proname
  FROM pg_proc pr,
       pg_type tp
 WHERE tp.oid = pr.prorettype
   AND pr.proisagg = FALSE
   AND tp.typname <> 'trigger'
   AND pr.pronamespace IN (
     SELECT oid
       FROM pg_namespace
      WHERE nspname NOT LIKE 'pg_%'
        AND nspname != 'information_schema'
);
-- with INFORMATION_SCHEMA:

SELECT routine_name
  FROM information_schema.routines
 WHERE specific_schema NOT IN
   ('pg_catalog', 'information_schema')
   AND type_udt_name != 'trigger';
```

Outra mais detalhada:

```
CREATE OR REPLACE FUNCTION public.function_args(
  IN funcname character varying,
  IN schema character varying,
  OUT pos integer,
  OUT direction character,
  OUT argname character varying,
  OUT datatype character varying)
RETURNS SETOF RECORD AS $$DECLARE
  rettype character varying;
  argtypes oidvector;
  allargtypes oid[];
  argmodes "char"[];
  argnames text[];
  mini integer;
  maxi integer;
BEGIN
  /* get object ID of function */
  SELECT INTO rettype, argtypes, allargtypes, argmodes, argnames
    CASE
      WHEN pg_proc.proretset
      THEN 'setof' || pg_catalog.format_type(pg_proc.prorettype, NULL)
      ELSE pg_catalog.format_type(pg_proc.prorettype, NULL) END,
      pg_proc.proargtypes,
      pg_proc.proallargtypes,
      pg_proc.proargmodes,
```

```

    pg_proc.proargnames
  FROM pg_catalog.pg_proc
    JOIN pg_catalog.pg_namespace
      ON (pg_proc.pronamespace = pg_namespace.oid)
 WHERE pg_proc.prorettype <> 'pg_catalog.cstring'::pg_catalog.regtype
   AND (pg_proc.proargtypes[0] IS NULL
     OR pg_proc.proargtypes[0] <> 'pg_catalog.cstring'::pg_catalog.regtype)
   AND NOT pg_proc.proisagg
   AND pg_proc.proname = funcname
   AND pg_namespace.nspname = schema
   AND pg_catalog.pg_function_is_visible(pg_proc.oid);

/* bail out if not found */
IF NOT FOUND THEN
  RETURN;
END IF;

/* return a row for the return value */
pos = 0;
direction = 'o'::char;
argname = 'RETURN VALUE';
datatype = rettype;
RETURN NEXT;

/* unfortunately allargtypes is NULL if there are no OUT parameters */
IF allargtypes IS NULL THEN
  mini = array_lower(argtypes, 1); maxi = array_upper(argtypes, 1);
ELSE
  mini = array_lower(allargtypes, 1); maxi = array_upper(allargtypes, 1);
END IF;
IF maxi < mini THEN RETURN; END IF;

/* loop all the arguments */
FOR i IN mini .. maxi LOOP
  pos = i - mini + 1;
  IF argnames IS NULL THEN
    argname = NULL;
  ELSE
    argname = argnames[i];
  END IF;
  IF allargtypes IS NULL THEN
    direction = 'i'::char;
    datatype = pg_catalog.format_type(argtypes[i], NULL);
  ELSE
    direction = argmodes[i];
    datatype = pg_catalog.format_type(allargtypes[i], NULL);
  END IF;
  RETURN NEXT;
END LOOP;

RETURN;
END $$ LANGUAGE plpgsql STABLE STRICT SECURITY INVOKER;
COMMENT ON FUNCTION public.function_args(character varying, character
varying)
IS $$For a function name and schema, this procedure selects for each
argument the following data:
- position in the argument list (0 for the return value)
- direction 'i', 'o', or 'b'
- name (NULL if not defined)
- data type$$;

```

Do artigo de Lorenzo Alberton. Extracting metadata from PostgreSQL using INFORMATION_SCHEMA :

http://www.alberton.info/postgresql_meta_info.html

Que também traz artigos sobre os catálogos do Firebird, Oracle e SQL Server.

Saber quantidade de registros no banco inteiro:

```
SELECT sum(C.reltuples)::int FROM pg_class C WHERE c.relkind = 'r'::"char";
```

Fonte: Blog da Kenia Milene

<http://keniamilene.wordpress.com/2007/09/18/contar-registros-em-banco-postgresql/>

Listando os bancos e sua codificação

```
SELECT datname, pg_encoding_to_char(encoding) FROM pg_database;
```

Exibindo codificações e versões

```
SELECT name, setting FROM pg_settings
WHERE name ~ 'encoding|^lc_|version';
```

Mostrar data da última execução do Vacuum e do Autovacuum

```
select relname, last_vacuum, last_autovacuum from
pg_stat_all_tables where schemaname like 'public';
```

Vide o capítulo "Table Statistics" do Livro PostgreSQL The Comprehensive Guide, da Sam's em: <http://www.iphelp.ru/faq/15/ch04lev1sec4.html>

Exibir Todas as Tabelas e seus Registros

```
select
    n.nspname as esquema,
    c.relname as tabela, c.reltuples::int as registros
from pg_class c
left join pg_namespace n on n.oid = c.relnamespace
left join pg_tablespace t on t.oid = c.reltablespace
where c.relkind = 'r'::char
    and nspname not in('information_schema','pg_catalog', 'pg_toast')
order by n.nspname,registros;
```

Outra:

```
select relname as tabelas, reltuples
from pg_class
```

where relkind = 'r'::char and relname not like 'pg_%' and relname not like 'sql_%' order by reltuples;

Outra solução:

VACUUM ANALYZE (em todo o banco)

```
SELECT sum(reltuples) as qtd_linhas
FROM pg_class
WHERE relkind = 'r';
```

criar uma view com a instrução SQL

Rodrigo Hjort - <http://icewall.org/~hjort> – na lista pgbr-geral

Boa fonte de leitura sobre o Catálogo do Sistema do PostgreSQL:
System Tables no capítulo 6 do livro: PostgreSQL Developer's Handbook de Ewald Geschwinde, Hans-Jürgen Schönig.

```
\c banco
vacuum analyze clientes;
SELECT relfilename AS arquivo, relpages*8 AS tamanho_em_kb FROM pg_class WHERE
relname = 'clientes';
```

Listar o nome de todos os administradores de banco de dados e o seus bancos de dados.

```
SELECT usename, datname
  FROM pg_user, pg_database
 WHERE usesysid = datdba
 ORDER BY usename, datname;
```

Mostrar todas as tabelas (inclusive de sistema) a quantidade de registros:
select relpages*8192 from pg_class;

Listando Tabelas do Banco Atual

```
SELECT relname
  FROM pg_class
 WHERE relname !~ '^(pg_|sql_)'
   AND relkind = 'r';
```

-- using INFORMATION_SCHEMA:

```
SELECT table_name
  FROM information_schema.tables
 WHERE table_type = 'BASE TABLE'
   AND table_schema NOT IN
 ('pg_catalog', 'information_schema');
```

Listando Views do Banco Atual

```
-- with postgresql 7.2:
SELECT viewname
  FROM pg_views
 WHERE viewname !~ '^pg_';

-- with postgresql 7.4 and later:
SELECT viewname
  FROM pg_views
 WHERE schemaname NOT IN
   ('pg_catalog', 'information_schema')
 AND viewname !~ '^pg_';

-- using INFORMATION_SCHEMA:
SELECT table_name
  FROM information_schema.tables
 WHERE table_type = 'VIEW'
   AND table_schema NOT IN
     ('pg_catalog', 'information_schema')
 AND table_name !~ '^pg_';
-- or
SELECT table_name
  FROM information_schema.views
 WHERE table_schema NOT IN ('pg_catalog', 'information_schema')
 AND table_name !~ '^pg_';
```

Listando Todos os Usuários

```
SELECT usename
  FROM pg_user;
```

Retornando os nomes dos Campos de uma Tabela

```
SELECT a.attname
  FROM pg_class c, pg_attribute a, pg_type t
 WHERE c.relname = 'test2'
   AND a.attnum > 0
   AND a.attrelid = c.oid
   AND a.atttypid = t.oid;
```

-- Usando INFORMATION_SCHEMA:

```
SELECT column_name
  FROM information_schema.columns
 WHERE table_name = 'test2';
```

Informações Detalhadas sobre os Campos de uma Tabela

```
SELECT a.attnum AS ordinal_position,
       a.attname AS column_name,
       t.typname AS data_type,
       a.attlen AS character_maximum_length,
```

```

a.atttypmod AS modifier,
a.attnotnull AS notnull,
a.atthasdef AS hasdefault,
col_description(a.attrelid, a.attnum) as field_comment
FROM pg_class c,
pg_attribute a,
pg_type t
WHERE c.relname = 'test2'
AND a.attnum > 0
AND a.attrelid = c.oid
AND a.atttypid = t.oid
ORDER BY a.attnum;

```

-- Usando INFORMATION_SCHEMA:

```

SELECT ordinal_position,
column_name,
data_type,
column_default,
is_nullable,
character_maximum_length,
numeric_precision
FROM information_schema.columns
WHERE table_name = 'test2'
ORDER BY ordinal_position;

```

Retornando os Nomes de Índices de uma Tabela

```

SELECT relname
FROM pg_class
WHERE oid IN (
    SELECT indexrelid
    FROM pg_index, pg_class
    WHERE pg_class.relname='test2'
    AND pg_class.oid=pg_index.indrelid
    AND indisunique != 't'
    AND indisprimary != 't'
);

```

Retornando as Constraints de uma Tabela

```

SELECT conname
FROM pg_constraint, pg_class
WHERE pg_constraint.conrelid = pg_class.oid
AND relname = 'test2';

```

-- with INFORMATION_SCHEMA:

```

SELECT constraint_name, constraint_type
FROM information_schema.table_constraints
WHERE table_name = 'test2';

```

Recebendo Informações Detalhadas sobre as Constraints de uma Tabela

```

SELECT c.conname AS constraint_name,

```

```

CASE c.contype
  WHEN 'c' THEN 'CHECK'
  WHEN 'f' THEN 'FOREIGN KEY'
  WHEN 'p' THEN 'PRIMARY KEY'
  WHEN 'u' THEN 'UNIQUE'
END AS "constraint_type",
CASE WHEN c.condeferrable = 'f' THEN 0 ELSE 1 END AS is_deferrable,
CASE WHEN c.condeferred = 'f' THEN 0 ELSE 1 END AS is_deferred,
t.relname AS table_name,
array_to_string(c.conkey, ' ') AS constraint_key,
CASE confupdtype
  WHEN 'a' THEN 'NO ACTION'
  WHEN 'r' THEN 'RESTRICT'
  WHEN 'c' THEN 'CASCADE'
  WHEN 'n' THEN 'SET NULL'
  WHEN 'd' THEN 'SET DEFAULT'
END AS on_update,
CASE confdeltype
  WHEN 'a' THEN 'NO ACTION'
  WHEN 'r' THEN 'RESTRICT'
  WHEN 'c' THEN 'CASCADE'
  WHEN 'n' THEN 'SET NULL'
  WHEN 'd' THEN 'SET DEFAULT'
END AS on_delete,
CASE confmatchtype
  WHEN 'u' THEN 'UNSPECIFIED'
  WHEN 'f' THEN 'FULL'
  WHEN 'p' THEN 'PARTIAL'
END AS match_type,
t2.relname AS references_table,
array_to_string(c.confkey, ' ') AS fk_constraint_key
FROM pg_constraint c
LEFT JOIN pg_class t ON c.conrelid = t.oid
LEFT JOIN pg_class t2 ON c.conrelid = t2.oid
  WHERE t.relname = 'testconstraints2'
    AND c.conname = 'testconstraints_id_fk';

```

-- with INFORMATION_SCHEMA:

```

SELECT tc.constraint_name,
  tc.constraint_type,
  tc.table_name,
  kcu.column_name,
  tc.is_deferrable,
  tc.initially_deferred,
  rc.match_option AS match_type,
  rc.update_rule AS on_update,
  rc.delete_rule AS on_delete,
  ccu.table_name AS references_table,
  ccu.column_name AS references_field
FROM information_schema.table_constraints tc

```

```

LEFT JOIN information_schema.key_column_usage kcu
    ON tc.constraint_catalog = kcu.constraint_catalog
    AND tc.constraint_schema = kcu.constraint_schema
    AND tc.constraint_name = kcu.constraint_name
LEFT JOIN information_schema.referential_constraints rc
    ON tc.constraint_catalog = rc.constraint_catalog
    AND tc.constraint_schema = rc.constraint_schema
    AND tc.constraint_name = rc.constraint_name
LEFT JOIN information_schema.constraint_column_usage ccu
    ON rc.unique_constraint_catalog = ccu.constraint_catalog
    AND rc.unique_constraint_schema = ccu.constraint_schema
    AND rc.unique_constraint_name = ccu.constraint_name
WHERE tc.table_name = 'testconstraints2'
    AND tc.constraint_name = 'testconstraints_id_fk';

```

Listando as Sequências

```

SELECT relname
    FROM pg_class
   WHERE relkind = 'S'
    AND relnamespace IN (
        SELECT oid
            FROM pg_namespace
           WHERE nspname NOT LIKE 'pg_%'
             AND nspname != 'information_schema'
);

```

Listando Todas as Triggers

```

SELECT trg.tgname AS trigger_name
    FROM pg_trigger trg, pg_class tbl
   WHERE trg.tgrelid = tbl.oid
     AND tbl.relname !~ '^pg_';
-- or
SELECT tgname AS trigger_name
    FROM pg_trigger
   WHERE tgname !~ '^pg_';

```

-- with INFORMATION_SCHEMA:

```

SELECT DISTINCT trigger_name
    FROM information_schema.triggers
   WHERE trigger_schema NOT IN
        ('pg_catalog', 'information_schema');

```

Listando Somente as Triggers de uma Tabela

```

SELECT trg.tgname AS trigger_name
    FROM pg_trigger trg, pg_class tbl
   WHERE trg.tgrelid = tbl.oid
     AND tbl.relname = 'newtable';

```

-- with INFORMATION_SCHEMA:

```
SELECT DISTINCT trigger_name
  FROM information_schema.triggers
 WHERE event_object_table = 'newtable'
   AND trigger_schema NOT IN
 ('pg_catalog', 'information_schema');
```

Recebendo Informações Detalhadas sobre as Triggers

```
SELECT trg.tgname AS trigger_name,
       tbl.relname AS table_name,
       p.proname AS function_name,
       CASE trg.tgtype & cast(2 as int2)
         WHEN 0 THEN 'AFTER'
         ELSE 'BEFORE'
       END AS trigger_type,
       CASE trg.tgtype & cast(28 as int2)
         WHEN 16 THEN 'UPDATE'
         WHEN 8 THEN 'DELETE'
         WHEN 4 THEN 'INSERT'
         WHEN 20 THEN 'INSERT, UPDATE'
         WHEN 28 THEN 'INSERT, UPDATE, DELETE'
         WHEN 24 THEN 'UPDATE, DELETE'
         WHEN 12 THEN 'INSERT, DELETE'
       END AS trigger_event
  FROM pg_trigger trg,
       pg_class tbl,
       pg_proc p
 WHERE trg.tgrelid = tbl.oid
   AND trg.tgfoid = p.oid
   AND tbl.relname !~ '^pg_';
-- with INFORMATION_SCHEMA:
SELECT *
  FROM information_schema.triggers
 WHERE trigger_schema NOT IN
 ('pg_catalog', 'information_schema');
```

Listar as Funções

```
SELECT proname
  FROM pg_proc pr,
       pg_type tp
 WHERE tp.oid = pr.prorettype
   AND pr.proisagg = FALSE
   AND tp.typname <> 'trigger'
   AND prpronamespace IN (
     SELECT oid
       FROM pg_namespace
      WHERE nspname NOT LIKE 'pg_%'
        AND nspname != 'information_schema'
   );
```

-- with INFORMATION_SCHEMA:

```
SELECT routine_name
  FROM information_schema.routines
 WHERE specific_schema NOT IN
      ('pg_catalog', 'information_schema')
    AND type_udt_name != 'trigger';
```

Saber quantidade de registros no banco inteiro:

```
SELECT sum(C.reltuples)::int FROM pg_class C WHERE c.relkind = 'r'::"char";
```

Fonte: Blog da Kenia Milene

<http://keniamilene.wordpress.com/2007/09/18/contar-registros-em-banco-postgresql/>

Listando os bancos e sua codificação

```
SELECT datname, pg_encoding_to_char(encoding) FROM pg_database;
```

Exibindo codificações e versões

```
SELECT name, setting FROM pg_settings
 WHERE name ~ 'encoding|^lc_|version';
```

Mostrar data da última execução do Vacuum e do Autovacuum

```
select relname, last_vacuum, last_autovacuum from pg_stat_all_tables where
 schemaname like 'public';
```

Vide o capítulo "Table Statistics" do Livro PostgreSQL The Comprehensive Guide, da Sam's em:

<http://www.iphelp.ru/faq/15/ch04lev1sec4.html>

Exibir Todas as Tabelas e seus Registros

```
select
  n.nspname as esquema,
  c.relname as tabela, c.reltuples::int as registros
from pg_class c
 left join pg_namespace n on n.oid = c.relnamespace
 left join pg_tablespace t on t.oid = c.reltablespace
where c.relkind = 'r'::char
  and nspname not in('information_schema','pg_catalog', 'pg_toast')
 order by n.nspname,registros;
```

Outra:

```
select relname as tabelas, reltuples
from pg_class
where relkind = 'r'::char and relname not like 'pg_%' and relname not like 'sql_%' order by
reltuples;
```

Outra solução:

```
VACUUM ANALYZE (em todo o banco)
SELECT sum(reltuples) as qtd_linhas
FROM pg_class
WHERE relkind = 'r';
```

criar uma view com a instrução SQL

Rodrigo Hjort - <http://icewall.org/~hjort> – na lista pgbr-geral

Uso do disco pela tabela

```
\c banco
vacuum analyze clientes;
SELECT relfilename AS arquivo, relpages*8 AS tamanho_em_kb FROM pg_class WHERE
relname = 'clientes';
```

Listar o nome de todos os administradores de banco de dados e o seus bancos de dados.

```
SELECT usename, datname
  FROM pg_user, pg_database
 WHERE usesysid = datdba
 ORDER BY usename, datname;
```

Mostrar todas as tabelas (inclusive de sistema) a quantidade de registros:
select relpages*8192 from pg_class;

Listando Tabelas do Banco Atual

```
SELECT relname
  FROM pg_class
 WHERE relname !~ '^(pg_|sql_)'
   AND relkind = 'r';
```

-- using INFORMATION_SCHEMA:

```
SELECT table_name
  FROM information_schema.tables
 WHERE table_type = 'BASE TABLE'
   AND table_schema NOT IN
 ('pg_catalog', 'information_schema');
```

Listando Views do Banco Atual

-- with postgresql 7.2:

```
SELECT viewname
  FROM pg_views
 WHERE viewname !~ '^pg_';
```

```
-- with postgresql 7.4 and later:
SELECT viewname
  FROM pg_views
 WHERE schemaname NOT IN
   ('pg_catalog', 'information_schema')
  AND viewname !~ '^pg_';

-- using INFORMATION_SCHEMA:
SELECT table_name
  FROM information_schema.tables
 WHERE table_type = 'VIEW'
  AND table_schema NOT IN
   ('pg_catalog', 'information_schema')
  AND table_name !~ '^pg_';

-- or
SELECT table_name
  FROM information_schema.views
 WHERE table_schema NOT IN ('pg_catalog', 'information_schema')
  AND table_name !~ '^pg_';
```

Listando Todos os Usuários

```
SELECT username
  FROM pg_user;
```

Retornando os nomes dos Campos de uma Tabela

```
SELECT a.attname
  FROM pg_class c, pg_attribute a, pg_type t
 WHERE c.relname = 'test2'
  AND a.attnum > 0
  AND a.attrelid = c.oid
  AND a.atttypid = t.oid;
```

-- Usando INFORMATION_SCHEMA:

```
SELECT column_name
  FROM information_schema.columns
 WHERE table_name = 'test2';
```

22 - 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.

23 - 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))'
#	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)'

Operador	Descrição	Exemplo
>>	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)	box	caixa de interseção	box_intersect(box '((0,0),(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))')

Função	Tipo retornado	Descrição	Exemplo
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))')
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)')

Função	Tipo retornado	Descrição	Exemplo
<code>polygon(npts, circle)</code>	<code>polygon</code>	círculo em polígono de npts-pontos	<code>polygon(12, circle '((0,0),2.0))'</code>
<code>polygon(path)</code>	<code>polygon</code>	caminho em polígono	<code>polygon(path '((0,0),(1,1),(2,0))')</code>

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.

Detalhes e matéria completa 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.

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'::polygon);
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

Obs.: Atualmente o conteúdo não mais se encontra no site acima.

Informações Detalhadas sobre os Campos de uma Tabela

```
SELECT a.attnum AS ordinal_position,
       a.attname AS column_name,
       t.typname AS data_type,
       a.attlen AS character_maximum_length,
       a.atttypmod AS modifier,
       a.attnotnull AS notnull,
       a.atthasdef AS hasdefault,
       col_description(a.attrelid, a.attnum) as field_comment
  FROM pg_class c,
       pg_attribute a,
       pg_type t
 WHERE c.relname = 'test2'
   AND a.attnum > 0
   AND a.attrelid = c.oid
   AND a.atttypid = t.oid
 ORDER BY a.attnum;
```

-- Usando INFORMATION_SCHEMA:

```
SELECT ordinal_position,
       column_name,
       data_type,
       column_default,
       is_nullable,
       character_maximum_length,
       numeric_precision
  FROM information_schema.columns
 WHERE table_name = 'test2'
 ORDER BY ordinal_position;
```

Retornando os Nomes de Índices de uma Tabela

```

SELECT relname
  FROM pg_class
 WHERE oid IN (
    SELECT indexrelid
      FROM pg_index, pg_class
     WHERE pg_class.relname='test2'
       AND pg_class.oid=pg_index.indrelid
       AND indisunique != 't'
       AND indisprimary != 't'
);

```

Retornando as Constraints de uma Tabela

```

SELECT conname
  FROM pg_constraint, pg_class
 WHERE pg_constraint.conrelid = pg_class.oid
   AND relname = 'test2';

```

-- with INFORMATION_SCHEMA:

```

SELECT constraint_name, constraint_type
  FROM information_schema.table_constraints
 WHERE table_name = 'test2';

```

Recebendo Informações Detalhadas sobre as Constraints de uma Tabela

```

SELECT c.conname AS constraint_name,
       CASE c.contype
         WHEN 'c' THEN 'CHECK'
         WHEN 'f' THEN 'FOREIGN KEY'
         WHEN 'p' THEN 'PRIMARY KEY'
         WHEN 'u' THEN 'UNIQUE'
       END AS "constraint_type",
       CASE WHEN c.condeferrable = 'f' THEN 0 ELSE 1 END AS is_deferrable,
       CASE WHEN c.condeferred = 'f' THEN 0 ELSE 1 END AS is_deferred,
       t.relname AS table_name,
       array_to_string(c.conkey, ' ') AS constraint_key,
       CASE confupdtype
         WHEN 'a' THEN 'NO ACTION'
         WHEN 'r' THEN 'RESTRICT'
         WHEN 'c' THEN 'CASCADE'
         WHEN 'n' THEN 'SET NULL'
         WHEN 'd' THEN 'SET DEFAULT'
       END AS on_update,
       CASE confdeltype
         WHEN 'a' THEN 'NO ACTION'
         WHEN 'r' THEN 'RESTRICT'
         WHEN 'c' THEN 'CASCADE'
         WHEN 'n' THEN 'SET NULL'
         WHEN 'd' THEN 'SET DEFAULT'
       END AS on_delete,
       CASE confmatchtype

```

```

        WHEN 'u' THEN 'UNSPECIFIED'
        WHEN 'f' THEN 'FULL'
        WHEN 'p' THEN 'PARTIAL'
    END AS match_type,
    t2.relname AS references_table,
    array_to_string(c.confkey, ' ') AS fk_constraint_key
FROM pg_constraint c
LEFT JOIN pg_class t ON c.conrelid = t.oid
LEFT JOIN pg_class t2 ON c.confrelid = t2.oid
WHERE t.relname = 'testconstraints2'
AND c.conname = 'testconstraints_id_fk';

```

-- with INFORMATION_SCHEMA:

```

SELECT tc.constraint_name,
       tc.constraint_type,
       tc.table_name,
       kcu.column_name,
       tc.is_deferrable,
       tc.initially_deferred,
       rc.match_option AS match_type,
       rc.update_rule AS on_update,
       rc.delete_rule AS on_delete,
       ccu.table_name AS references_table,
       ccu.column_name AS references_field
FROM information_schema.table_constraints tc
LEFT JOIN information_schema.key_column_usage kcu
      ON tc.constraint_catalog = kcu.constraint_catalog
     AND tc.constraint_schema = kcu.constraint_schema
     AND tc.constraint_name = kcu.constraint_name
LEFT JOIN information_schema.referential_constraints rc
      ON tc.constraint_catalog = rc.constraint_catalog
     AND tc.constraint_schema = rc.constraint_schema
     AND tc.constraint_name = rc.constraint_name
LEFT JOIN information_schema.constraint_column_usage ccu
      ON rc.unique_constraint_catalog = ccu.constraint_catalog
     AND rc.unique_constraint_schema = ccu.constraint_schema
     AND rc.unique_constraint_name = ccu.constraint_name
WHERE tc.table_name = 'testconstraints2'
    AND tc.constraint_name = 'testconstraints_id_fk';

```

Listando as Sequências

```

SELECT relname
FROM pg_class
WHERE relkind = 'S'
    AND relnamespace IN (
        SELECT oid
        FROM pg_namespace
        WHERE nspname NOT LIKE 'pg_%'
            AND nspname != 'information_schema'

```

);

Listando Todas as Triggers

```
SELECT trg.tgname AS trigger_name
  FROM pg_trigger trg, pg_class tbl
 WHERE trg.tgrelid = tbl.oid
   AND tbl.relname !~ '^pg_';
-- or
SELECT tname AS trigger_name
  FROM pg_trigger
 WHERE tname !~ '^pg_';
```

-- with INFORMATION_SCHEMA:

```
SELECT DISTINCT trigger_name
  FROM information_schema.triggers
 WHERE trigger_schema NOT IN
 ('pg_catalog', 'information_schema');
```

Listando Somente as Triggers de uma Tabela

```
SELECT trg.tgname AS trigger_name
  FROM pg_trigger trg, pg_class tbl
 WHERE trg.tgrelid = tbl.oid
   AND tbl.relname = 'newtable';
```

-- with INFORMATION_SCHEMA:

```
SELECT DISTINCT trigger_name
  FROM information_schema.triggers
 WHERE event_object_table = 'newtable'
   AND trigger_schema NOT IN
 ('pg_catalog', 'information_schema');
```

Recebendo Informações Detalhadas sobre as Triggers

```
SELECT trg.tgname AS trigger_name,
      tbl.relname AS table_name,
      p.proname AS function_name,
      CASE trg.tgtype & cast(2 as int2)
        WHEN 0 THEN 'AFTER'
        ELSE 'BEFORE'
      END AS trigger_type,
      CASE trg.tgtype & cast(28 as int2)
        WHEN 16 THEN 'UPDATE'
        WHEN 8 THEN 'DELETE'
        WHEN 4 THEN 'INSERT'
        WHEN 20 THEN 'INSERT, UPDATE'
        WHEN 28 THEN 'INSERT, UPDATE, DELETE'
        WHEN 24 THEN 'UPDATE, DELETE'
        WHEN 12 THEN 'INSERT, DELETE'
      END AS trigger_event
  FROM pg_trigger trg,
       pg_class tbl,
```

```

pg_proc p
WHERE trg.tgrelid = tbl.oid
  AND trg.tgfoid = p.oid
  AND tbl.relname !~ '^pg_';
-- with INFORMATION_SCHEMA:
SELECT *
  FROM information_schema.triggers
 WHERE trigger_schema NOT IN
  ('pg_catalog', 'information_schema');

```

Listar as Funções

```

SELECT proname
  FROM pg_proc pr,
       pg_type tp
 WHERE tp.oid = pr.prorettype
   AND pr.proisagg = FALSE
   AND tp.typname <> 'trigger'
   AND prpronamespace IN (
    SELECT oid
      FROM pg_namespace
     WHERE nspname NOT LIKE 'pg_%'
       AND nspname != 'information_schema'
);

```

-- with INFORMATION_SCHEMA:

```

SELECT routine_name
  FROM information_schema.routines
 WHERE specific_schema NOT IN
  ('pg_catalog', 'information_schema')
   AND type_udt_name != 'trigger';

```

Saber quantidade de registros no banco inteiro:

```
SELECT sum(C.reltuples)::int FROM pg_class C WHERE c.relkind = 'r'::"char";
```

Fonte: Blog da Kenia Milene

<http://keniamilene.wordpress.com/2007/09/18/contar-registros-em-banco-postgresql/>

Listando os bancos e sua codificação

```
SELECT datname, pg_encoding_to_char(encoding) FROM pg_database;
```

Exibindo codificações e versões

```
SELECT name, setting FROM pg_settings
 WHERE name ~ 'encoding|^lc_|version';
```

Mostrar data da última execução do Vacuum e do Autovacuum

```
select relname, last_vacuum, last_autovacuum from pg_stat_all_tables where
schemaname like 'public';
```

Vide o capítulo "Table Statistics" do Livro PostgreSQL The Comprehensive Guide, da Sam's em:

<http://www.iphelp.ru/faq/15/ch04lev1sec4.html>

Exibir Todas as Tabelas e seus Registros

```
select
    n.nspname as esquema,
    c.relname as tabela, c.reltuples::int as registros
from pg_class c
    left join pg_namespace n on n.oid = c.relnamespace
    left join pg_tablespace t on t.oid = c.reltablespace
where c.relkind = 'r'::char
    and nspname not in('information_schema','pg_catalog', 'pg_toast')
    order by n.nspname,registros;
```

Outra:

```
select relname as tabelas, reltuples
from pg_class
where relkind = 'r'::char and relname not like 'pg_%' and relname not like 'sql_%' order by
reltuples;
```

Outra solução:

```
VACUUM ANALYZE (em todo o banco)
SELECT sum(reltuples) as qtd_linhas
FROM pg_class
WHERE relkind = 'r';
```

Mostrar todas as tabelas úteis de um banco com seus registros:

```
select relname, reltuples from pg_class where relname !~ '^(pg_|sql_)' and relkind = 'r';
```

```
SELECT ordinal_position,
column_name,
data_type,
column_default,
is_nullable,
character_maximum_length,
numeric_precision
FROM information_schema.columns
WHERE table_name = 'clientes' and column_name='nome'
ORDER BY ordinal_position;
```

RECEBER PRIMARY KEY

```

SELECT kcu.column_name AS pk
  FROM information_schema.table_constraints tc
LEFT JOIN information_schema.key_column_usage kcu
    ON tc.constraint_name = kcu.constraint_name
   WHERE tc.table_name = 'clientes'
     AND constraint_type = 'PRIMARY KEY';

```

RECEBER MAIS INFOS

```

SELECT tc.constraint_name,
       tc.constraint_type,
       tc.table_name,
       kcu.column_name,
       tc.is_deferrable,
       tc.initially_deferred,
       rc.match_option AS match_type,
       rc.update_rule AS on_update,
       rc.delete_rule AS on_delete,
       ccu.table_name AS references_table,
       ccu.column_name AS references_field
  FROM information_schema.table_constraints tc
LEFT JOIN information_schema.key_column_usage kcu
    ON tc.constraint_catalog = kcu.constraint_catalog
   AND tc.constraint_schema = kcu.constraint_schema
   AND tc.constraint_name = kcu.constraint_name
LEFT JOIN information_schema.referential_constraints rc
    ON tc.constraint_catalog = rc.constraint_catalog
   AND tc.constraint_schema = rc.constraint_schema
   AND tc.constraint_name = rc.constraint_name
LEFT JOIN information_schema.constraint_column_usage ccu
    ON rc.unique_constraint_catalog = ccu.constraint_catalog
   AND rc.unique_constraint_schema = ccu.constraint_schema
   AND rc.unique_constraint_name = ccu.constraint_name
   WHERE tc.table_name = 'clientes';

```

```

select column_name from information_schema.constraint_column_usage where
table_name = '$table' and constraint_name = '{$table}_pkey';

```

MOSTRAR FOREIGN KEY

```

SELECT
  kcu.column_name
FROM
  information_schema.table_constraints AS tc
  JOIN information_schema.key_column_usage AS kcu ON tc.constraint_name =
kcu.constraint_name
  JOIN information_schema.constraint_column_usage AS ccu ON ccu.constraint_name =
tc.constraint_name

```

```
WHERE constraint_type = 'FOREIGN KEY' AND tc.table_name='pedidos';
```

Extracting META information from PostgreSQL (INFORMATION_SCHEMA)

http://www.alberton.info/postgresql_meta_info.html

Exibir todos os registros de todas as tabelas de um banco:

-- Como o count(*) é pesado...

-- Executar analyze no banco

\c banco

ANALYZE;

-- Consulta

```
select sum(reltuples) as registros from pg_class where relkind='r'
and relname not like 'pg%' and relname not like 'sql_%';
```

-- View

create view todos_regs as

```
select sum(reltuples) as registros from pg_class where relkind='r'
and relname not like 'pg%' and relname not like 'sql_%';
```

-- Usando a view

```
select * from todos_regs;
```

Dica do Rodrigo Hjort na lista PostgreSQL Brasil

```
<?php
```

```
$conexao=pg_connect("host=127.0.0.1 user=postgres password=postabir");
```

```
$sql="SELECT datname AS banco FROM pg_database ORDER BY datname";
$consulta=pg_query($conexao,$sql);
```

```
$banco = array();
$c=0;
while ($data = @pg_fetch_object($consulta,$c)) {
    $cons=$data->banco;

    $banco[] .= $cons;
    $c++;
}
```

```
$sql2="SELECT n.nspname as esquema,c.relname as tabela FROM pg_namespace n,
pg_class c
```

```
WHERE n.oid = c.relnamespace
and c.relkind = 'r' -- no indices
and n.nspname not like 'pg\\_%' -- no catalogs
```

```

and n.nspname != 'information_schema' -- no information_schema
ORDER BY nspname, relname";

for ($x=0; $x < count($banco);$x++){
    if ($banco[$x] != "template0" && $banco[$x] != "template1" && $banco[$x] != "postgres")
{
    $conexao2=pg_connect("host=127.0.0.1 dbname=$banco[$x] user=postgres
password=postabir");
    $consulta2=pg_query( $conexao2, $sql2 );

    while ($data = pg_fetch_object($consulta2)) {
        $esquematab=$data->esquema.''.$data->tabela;
        $sql3="SELECT count(*) FROM $esquematab";
        $consulta3=pg_query($conexao2,$sql3);
        $res=@pg_fetch_array($consulta3);

        print 'Banco.Esquema.Tabela -> '.$banco[$x].'.'.$data->esquema.''.$data-
>tabela.' - Registro(s) - '.$res[0].'  
';
        $total += $res[0];
    }
}
print "Total de Registro de todas as tabelas de todos os bancos ". $total;

?>

```

24 - PostgreSQL Dicas

COALESCE: Trate decisões envolvendo campos nulos!

A função COALESCE permite que se selecione, entre dois ou mais parâmetros, o primeiro valor não nulo, retornando nulo caso todos os valores passados como parâmetro sejam nulos. É um recurso que pode ser utilizado para dar mais elegância ao tratamento de valores nulos e ao mesmo tempo reduzir o tamanho das consultas, tornando-as mais fáceis de manter.

A sintaxe é bem simples, pois a função COALESCE recebe uma lista de valores como parâmetro, separados por vírgula.

Exemplo 1: Apenas um parâmetro fornecido

```
postgres=# SELECT COALESCE(1);
coalesce
-----
1
(1 registro)
```

Exemplo 2: Dois parâmetros fornecidos

```
postgres=# SELECT COALESCE(null,2);
coalesce
-----
2
(1 registro)
```

Exemplo 3: Três parâmetros fornecidos

```
postgres=# SELECT COALESCE(1,2,3);
coalesce
-----
1
(1 registro)
```

Exemplo 4: O primeiro parâmetro é nulo.

```
postgres=# SELECT COALESCE(null,2,3);
coalesce
-----
```

2

(1 registro)

Exemplo 5: Os dois primeiros parâmetros são nulos.

```
postgres=# SELECT COALESCE(null,null,3);
coalesce
-----
3
(1 registro)
```

Exemplo 6: Exemplo com cinco parâmetros e valor do tipo data.

```
postgres=# SELECT COALESCE(null, null, null, null, current_date);
coalesce
-----
2014-05-02
(1 registro)
```

Exemplo 7: Exemplo utilizando campos de uma tabela como parâmetro.

```
postgres=# CREATE TEMP TABLE TBL_COA(campo1 INTEGER, campo2 INTEGER,
campo3 INTEGER);
postgres=# INSERT INTO TBL_COA VALUES (null,1,2);
postgres=# INSERT INTO TBL_COA VALUES (null,null,2);
postgres=# INSERT INTO TBL_COA VALUES (null,null,null);

postgres=# SELECT COALESCE(campo1,campo2, campo3) FROM TBL_COA;
coalesce
-----
1
2

(3 registros)
```

Exemplo 8: Implementação das condições do exemplo anterior utilizando a cláusula CASE. O plano de execução é o mesmo, mas o código fica bem mais complexo.

```
postgres=# SELECT
postgres# CASE WHEN campo1 IS NOT NULL THEN campo1
postgres# WHEN campo1 IS NULL AND campo2 IS NOT NULL THEN campo2
postgres# WHEN campo1 IS NULL AND campo2 IS NULL AND campo3 IS NOT NULL
THEN campo3
```

```
postgres=# ELSE null END AS simula_coalesce
postgres=# FROM TBL_COA;
simula_coalesce
-----
1
2

(3 registros)
```

* Conclusões

- O uso de COALESCE pode tornar seu código mais enxuto e fácil de manter;
- Pode substituir a cláusula CASE no tratamento de valores NULOS, embora não haja ganho de desempenho.

explain.depesz: Encontre a Causa da Lentidão em suas Consultas!

O site <http://explain.depesz.com/> disponibiliza uma ferramenta bastante útil, que formata, extrai e apresenta de forma relativamente simples o conteúdo dos planos de execução do postgresql, permitindo uma análise mais fácil do mesmo. A promessa deste aplicativo é ajudar a encontrar a causa da lentidão nas consultas realizadas.

Site da ferramenta

Para utilizar a ferramenta, basta acessar o site, colar na caixa de texto o resultado do comando EXPLAIN

Simples e Útil: Visões Materializadas no PostgreSQL!

Visões materializadas são recursos introduzidos na versão 9.3 do postgresql. Enquanto visões tradicionais reexecutam uma consulta sempre que são referenciadas, visões materializadas dispensam este esforço pelos seus dados já estarem guardados desde a sua criação ou do último refresh (atualização de visão). Pode-se dizer que uma visão materializada é um objeto que contém o resultado de uma consulta, facilitando o acesso aos dados nela contidos.

A principal justificativa para se utilizar visões materializadas é a **aceleração de consultas em grandes massas de dados**. É importante observar que em sistemas com pouco espaço em disco e discos lentos, visões materializadas podem ter pouco efeito ou até

impacto negativo por sobrecarregar ainda mais o hardware.

Para ter ainda mais desempenho, é possível criar índices para visões materializadas.

No postgresql, a atualização de uma visão materializada é feita através do comando REFRESH MATERIALIZED VIEW, enquanto que a mudança do código da consulta da visão é feita através do comando ALTER MATERIALIZED VIEW. A exclusão de visões materializadas é feita com o comando DROP MATERIALIZED VIEW.

Sintaxe básica:

```
CREATE MATERIALIZED VIEW nome_tabela
[ (nome_coluna [, ...] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) ]
[ TABLESPACE nome_tablespace ]
AS consulta
[ WITH [ NO ] DATA ]
```

Exemplo 1: Criação de uma visão materializada simples

```
CREATE MATERIALIZED VIEW lista_tabelas AS
SELECT tablename FROM PG_TABLES ORDER BY tablename;
```

Exemplo 2: Refresh dos dados de uma visão materializada, pelo comando REFRESH MATERIALIZED VIEW.

```
REFRESH MATERIALIZED VIEW lista_tabelas;
```

Exemplo 3: Exclusão de uma visão materializada utilizando DROP MATERIALIZED VIEW. O comando drop view não exclui visões materializadas, e sim gera erro.

```
DROP MATERIALIZED VIEW lista_tabelas;
```

Exemplo 4: Criação de uma visão materializada sem dados. Neste caso, o comando REFRESH MATERIALIZED VIEW pode ser utilizado para popular a visão armazenada.

```
CREATE MATERIALIZED VIEW lista_tabelas_nodata AS
SELECT tablename FROM PG_TABLES ORDER BY tablename WITH NO DATA;
```

Exemplo 5: Criação de uma visão materializada simples, explicitando o tablespace utilizado

```
CREATE MATERIALIZED VIEW lista_indices TABLESPACE pg_default AS
SELECT schemaname, tablename, indexname, tablespace FROM
PG_INDEXES;
```

Exemplo 6: Criação de uma visão materializada simples, utilizando o storage parameter fillfactor. São aceitos todos os tipos de storage parameters de uma tabela padrão, exceto OIDs, pois visões materializadas não apresentam identificador OID para cada registro.

```
CREATE MATERIALIZED VIEW lista_indices_fill50 WITH (fillfactor = 50) AS
SELECT schemaname, tablename, indexname, tablespace FROM
PG_INDEXES;
```

Exemplo 7: Para saber quantas visões armazenadas você tem no seu servidor, utilize a visão **PG_MATVIEWS**.

```
SELECT * FROM PG_MATVIEWS;
```

Exemplo 8: Alteração de visão materializada. O comando ALTER MATERIALIZED VIEW apresenta uma sintaxe mais elaborada, merecendo mais espaço em um texto futuro.

```
ALTER MATERIALIZED VIEW lista_tabelas RENAME TO lista_relacoes;
```

* Conclusões

Visões materializadas são uma boa opção para aumento de performance sob certas condições.

Também facilitam a importação de visões materializadas disponíveis em outros SGBDs como ORACLE e SQL SERVER.

A implementação de visões materializadas é relativamente fácil e bastante útil.

Você já utilizou esta funcionalidade nos seus projetos? Qual foi a sua opinião?

[Beltrano: Base de Dados em Português para o PostgreSQL](#)

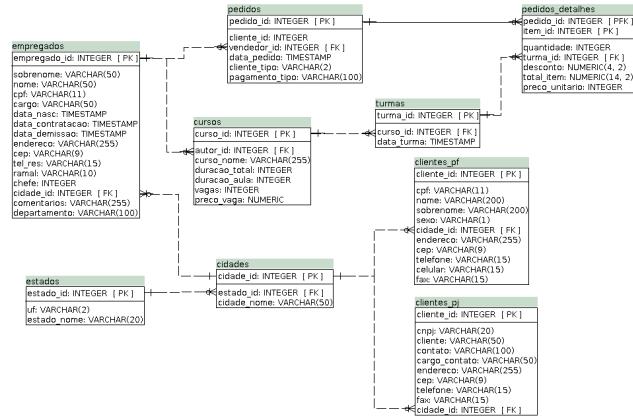
Às vezes ideias simples melhoram a nossa vida. Compartilhar projetos simples pode ajudar muita gente. Este é o caso do Beltrano, **base de dados em português desenvolvida para aplicações OLAP e OLTP e compatível com o postgresql**, pelo analista [Fábio de Salles](#), bastante conhecido na comunidade pentaho. O banco simula uma aplicação simples, com empregados, cursos e pedidos, ideal para treinamentos e o ensino de bancos de dados e tecnologias.

O autor criou [projeto completo no sourceforge](#), com a modelagem na ferramenta power architect, e os bancos de dados produzidos disponibilizados para download.

Adicionalmente, colocou um [tutorial com o passo a passo para a criação das bases no postgresql](#).

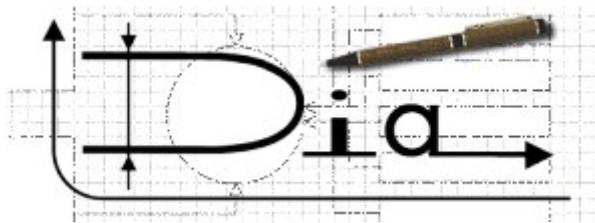
Você é livre para conhecer, baixar, utilizar e melhorar o Beltrano!

Abaixo, modelagem da base OLTP:



Faça Você Mesmo: Diagramas de Classe com as Tabelas do Catálogo do PostgreSQL

Automatizar tarefas complexas não é uma tarefa fácil. Dentro da área de banco de dados a elaboração e manutenção da modelagem do banco de dados é uma tarefa das mais difíceis. Os modelos simplesmente não refletem a realidade, ou são incompletos, por limitações das ferramentas ou pela baixa prioridade atribuída à modelagem. Esta postagem mostra as etapas para se fazer a geração automática de diagramas de classe com base no modelo de dados. Foram modelados os esquemas PG_CATALOG e INFORMATION_SCHEMA do postgresql, através do [editor de diagramas DIA](#), que pode ser baixada no sourceforge e funciona tanto no windows quanto no linux.



A escolha da ferramenta de modelagem se deu pelo fato da mesma permitir a geração de diagramas complexos através de scripts em Python. Infelizmente, a documentação encontrada não é das melhores, o que dificulta este tipo de empreitada.

Basicamente, foram seguidas as seguintes etapas para a geração dos diagramas:

- 1 - Consultas aos metadados do postgresql, relativas aos esquemas desejados, gerando como resultado um script python;
- 2 - Salvamento deste script em um arquivo texto;

- 3 - Ajuste manual do script para alterar detalhes como textos apresentados e remoção de espaços;
- 4 - Execução do script python de dentro da ferramenta DIA, para a criação do diagrama;
- 5 - Ajuste manual de alguma imperfeição estética no diagrama gerado.

Antes de saber como fazer, veja o resultado obtido com o mínimo de alterações manuais:



Figura 1 - PG_CATALOG



Figura 2 - INFORMATION_SCHEMA

1 - Geração e Edição de Script Python - PG_CATALOG

A geração do script python para a criação do diagrama foi feita utilizando-se sql no esquema do banco de dados desejado.

Abaixo coloco a consulta de geração do diagrama do esquema pg_catalog. É mais simples do que parece, sendo composto por uma sequência de várias consultas, unidas por UNION ALL, gerando linhas de texto com o script:

```
SELECT CAST ('diagram = dia.new("PG_CATALOG.dia")' AS TEXT) as OBJETO
UNION ALL
SELECT CAST ('data = diagram.data' AS TEXT) as OBJETO
UNION ALL
SELECT CAST ('display = diagram.display()' AS TEXT) as OBJETO
UNION ALL
SELECT CAST ('layer = data.active_layer' AS TEXT) as OBJETO
UNION ALL
SELECT CAST ('pg_objtype = "UML - Class"' AS TEXT) as OBJETO
UNION ALL
SELECT CAST ('oType = dia.get_object_type (pg_objtype)' AS TEXT) as OBJETO
```

```

UNION ALL
SELECT CAST ('theObjects = [diagram, data, layer, display, oType]' AS TEXT) as
OBJETO
UNION ALL
SELECT CAST(relname || ', h1, h2 = oType.create (' || 20*((row_number() OVER
(PARTITION BY schemaname)) % 12) || ',' || 30*((row_number() OVER (PARTITION BY
schemaname)) % 5) || ')' as TEXT) as OBJETO FROM pg_catalog.pg_statio_sys_tables
WHERE schemaname = 'pg_catalog'
UNION ALL
SELECT CAST('theObjects.append(' || relname || ')' as TEXT) as OBJETO FROM
pg_catalog.pg_statio_sys_tables WHERE schemaname = 'pg_catalog'
UNION ALL
SELECT CAST(relname || '.properties["stereotype"] = "pg_catalog"' as TEXT) as
OBJETO FROM pg_catalog.pg_statio_sys_tables WHERE schemaname = 'pg_catalog'
UNION ALL
SELECT CAST(relname || '.properties["comment"] = "' || relname || ' Class'" as
TEXT) as OBJETO FROM pg_catalog.pg_statio_sys_tables WHERE schemaname =
'pg_catalog'
UNION ALL
SELECT CAST(relname || '.properties["name"] = "' || relname || '"' as TEXT) as
OBJETO FROM pg_catalog.pg_statio_sys_tables WHERE schemaname = 'pg_catalog'
UNION ALL
SELECT CAST('attributes_ ' || relname || ' = []' as TEXT) as OBJETO FROM
pg_catalog.pg_statio_sys_tables WHERE schemaname = 'pg_catalog'
UNION ALL
SELECT CAST('attributes_ ' || table_name || '.append(' || column_name || ''
','' || data_type || ''','Value_not_set','','' || column_name || '-' ||
data_type || '',1,0,0)) ' AS TEXT) as OBJETO from information_schema.columns
where table_schema = 'pg_catalog' AND table_name IN (SELECT relname as
table_name FROM pg_catalog.pg_statio_sys_tables WHERE schemaname = 'pg_catalog')
UNION ALL
SELECT CAST(relname || '.properties["attributes"] = attributes_ ' || relname as
TEXT) as OBJETO FROM pg_catalog.pg_statio_sys_tables WHERE schemaname =
'pg_catalog'
UNION ALL
SELECT CAST('layer.add_object (' || relname || ')' as TEXT) as OBJETO FROM
pg_catalog.pg_statio_sys_tables WHERE schemaname = 'pg_catalog'
UNION ALL
SELECT CAST('oType = dia.get_object_type ("UML - Activity")' as TEXT) as OBJETO
UNION ALL
SELECT CAST('oend, h1, h2 = oType.create (0,-1)' as TEXT) as OBJETO
UNION ALL
SELECT CAST('theObjects.append(oend)' as TEXT) as OBJETO
UNION ALL
SELECT CAST('oend.properties["text"] = "PG_CATALOG: Claudio Bezerra Leopoldino -
Meu Blog de PostgreSQL - http://postgresqlbr.blogspot.com.br/ -
claudiob_br@yahoo.com.br"' as TEXT) as OBJETO
UNION ALL
SELECT CAST('layer.add_object (oend)' as TEXT) as OBJETO
UNION ALL
SELECT CAST('diagram.save()' as TEXT) as OBJETO

```

```
UNION ALL
```

```
SELECT CAST('print "Generation FINISHED"' as TEXT) as OBJETO;
```

Após a execução deste script SQL, salve o resultado em arquivo texto com extensão .py e o edite, acrescentando, excluindo e alterando elementos.

3 - Execução do script no DIA

É feita através do python console, acessado pelo menu "Diálogos/Python Console" do DIA.

Para executar, basta digitar no console "execfile("nomedoarquivo")" e confirmar com enter, como na figura abaixo. Após a execução, o DIA criará o diagrama de acordo com o script python fornecido.

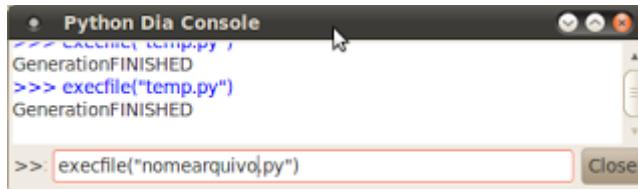


Figura 3 - Python Console

4 - Geração de Script Python - INFORMATION_SCHEMA

Abaixo coloco a consulta de geração do diagrama do esquema information_schema:

```
SELECT CAST ('diagram = dia.new("INFORMATION_SCHEMA.dia")' AS TEXT) as OBJETO
UNION ALL
SELECT CAST ('data = diagram.data' AS TEXT) as OBJETO
UNION ALL
SELECT CAST ('display = diagram.display()' AS TEXT) as OBJETO
UNION ALL
SELECT CAST ('layer = data.active_layer' AS TEXT) as OBJETO
UNION ALL
SELECT CAST ('pg_objtype = "UML - Class"' AS TEXT) as OBJETO
UNION ALL
SELECT CAST ('oType = dia.get_object_type (pg_objtype)' AS TEXT) as OBJETO
UNION ALL
SELECT CAST ('theObjects = [diagram, data, layer, display, oType]' AS TEXT) as
OBJETO
UNION ALL
SELECT CAST(relename || ', h1, h2 = oType.create (' || 20*(row_number() OVER
(PARTITION BY schemaname)) || ',1)' as TEXT) as OBJETO FROM
pg_catalog.pg_statio_sys_tables WHERE schemaname = 'information_schema'
UNION ALL
SELECT CAST('theObjects.append(' || relname || ')' as TEXT) as OBJETO FROM
pg_catalog.pg_statio_sys_tables WHERE schemaname = 'information_schema'
```

```

UNION ALL
SELECT CAST(relname || '.properties["stereotype"] = "information_schema"' as
TEXT) as OBJETO FROM pg_catalog.pg_statio_sys_tables WHERE schemaname =
'information_schema'
UNION ALL
SELECT CAST(relname || '.properties["comment"] = "' || relname || ' Class"' as
TEXT) as OBJETO FROM pg_catalog.pg_statio_sys_tables WHERE schemaname =
'information_schema'
UNION ALL
SELECT CAST(relname || '.properties["name"] = "' || relname || '"' as TEXT) as
OBJETO FROM pg_catalog.pg_statio_sys_tables WHERE schemaname =
'information_schema'
UNION ALL
SELECT CAST('attributes_' || relname || ' = []' as TEXT) as OBJETO FROM
pg_catalog.pg_statio_sys_tables WHERE schemaname = 'information_schema'
UNION ALL
SELECT CAST('attributes_' || table_name || '.append('' || column_name ||
''','' || data_type || ''',''Value_not_set'', '' || column_name || '-' ||
data_type || ''',1,0,0)) ' AS TEXT) as OBJETO from information_schema.columns
where table_schema = 'information_schema' AND table_name IN (SELECT relname as
table_name FROM pg_catalog.pg_statio_sys_tables WHERE schemaname =
'information_schema')
UNION ALL
SELECT CAST(relname || '.properties["attributes"] = attributes_' || relname as
TEXT) as OBJETO FROM pg_catalog.pg_statio_sys_tables WHERE schemaname =
'information_schema'
UNION ALL
SELECT CAST('layer.add_object (' || relname || ')' as TEXT) as OBJETO FROM
pg_catalog.pg_statio_sys_tables WHERE schemaname = 'information_schema'
UNION ALL
SELECT CAST('oType = dia.get_object_type ("UML - Activity")' as TEXT) as OBJETO
UNION ALL
SELECT CAST('oend, h1, h2 = oType.create (0,-1)' as TEXT) as OBJETO
UNION ALL
SELECT CAST('theObjects.append(oend)' as TEXT) as OBJETO
UNION ALL
SELECT CAST('oend.properties["text"] = "INFORMATION_SCHEMA: Claudio Bezerra
Leopoldino - Meu Blog de PostgreSQL - http://postgresqlbr.blogspot.com.br/
-claudiobr@yahoo.com.br"' as TEXT) as OBJETO
UNION ALL
SELECT CAST('layer.add_object (oend)' as TEXT) as OBJETO
UNION ALL
SELECT CAST('diagram.save()' as TEXT) as OBJETO
UNION ALL
SELECT CAST('print "Generation FINISHED"' as TEXT) as OBJETO;

```

5 - Script Python Produzido - INFORMATION_SCHEMA

Abaixo coloco a listagem do script python produzido e editado:

```
diagram=dia.new("INFORMATION_SCHEMA.dia")
data=diagram.data
display=diagram.display()
layer=data.active_layer
pg_objtype="UML - Class"
oType=dia.get_object_type(pg_objtype)
theObjects=[diagram,data,layer,display,oType]
sql_implementation_info,h1,h2=oType.create(20,1)
sql_sizing_profiles,h1,h2=oType.create(40,1)
sql_features,h1,h2=oType.create(60,1)
sql_languages,h1,h2=oType.create(80,1)
sql_packages,h1,h2=oType.create(100,1)
sql_parts,h1,h2=oType.create(120,1)
sql_sizing,h1,h2=oType.create(140,1)
theObjects.append(sql_implementation_info)
theObjects.append(sql_sizing_profiles)
theObjects.append(sql_features)
theObjects.append(sql_languages)
theObjects.append(sql_packages)
theObjects.append(sql_parts)
theObjects.append(sql_sizing)
sql_implementation_info.properties["stereotype"]="information_schema"
sql_sizing_profiles.properties["stereotype"]="information_schema"
sql_features.properties["stereotype"]="information_schema"
sql_languages.properties["stereotype"]="information_schema"
sql_packages.properties["stereotype"]="information_schema"
sql_parts.properties["stereotype"]="information_schema"
sql_sizing.properties["stereotype"]="information_schema"
sql_implementation_info.properties["comment"]="sql_implementation_infoClass"
sql_sizing_profiles.properties["comment"]="sql_sizing_profilesClass"
sql_features.properties["comment"]="sql_featuresClass"
sql_languages.properties["comment"]="sql_languagesClass"
sql_packages.properties["comment"]="sql_packagesClass"
sql_parts.properties["comment"]="sql_partsClass"
sql_sizing.properties["comment"]="sql_sizingClass"
sql_implementation_info.properties["name"]="sql_implementation_info"
sql_sizing_profiles.properties["name"]="sql_sizing_profiles"
sql_features.properties["name"]="sql_features"
sql_languages.properties["name"]="sql_languages"
sql_packages.properties["name"]="sql_packages"
sql_parts.properties["name"]="sql_parts"
sql_sizing.properties["name"]="sql_sizing"
attributes_sql_implementation_info=[]
attributes_sql_sizing_profiles=[]
attributes_sql_features=[]
attributes_sql_languages=[]
attributes_sql_packages=[]
attributes_sql_parts=[]
attributes_sql_sizing=[]
attributes_sql_implementation_info.append(('implementation_info_id','charactervarying','Value_not_set','implementation_info_id-charactervarying',1,0,0))
```

```
attributes_sql_implementation_info.append(('implementation_info_name','character varying','Value_not_set','implementation_info_name-charactervarying',1,0,0))
attributes_sql_implementation_info.append(('integer_value','integer','Value_not_set','integer_value-integer',1,0,0))
attributes_sql_implementation_info.append(('character_value','charactervarying','Value_not_set','character_value-charactervarying',1,0,0))
attributes_sql_implementation_info.append(('comments','charactervarying','Value_not_set','comments-charactervarying',1,0,0))
attributes_sql_languages.append(('sql_language_source','charactervarying','Value_not_set','sql_language_source-charactervarying',1,0,0))
attributes_sql_languages.append(('sql_language_year','charactervarying','Value_not_set','sql_language_year-charactervarying',1,0,0))
attributes_sql_languages.append(('sql_language_conformance','charactervarying','Value_not_set','sql_language_conformance-charactervarying',1,0,0))
attributes_sql_languages.append(('sql_language_integrity','charactervarying','Value_not_set','sql_language_integrity-charactervarying',1,0,0))
attributes_sql_languages.append(('sql_language_implementation','charactervarying','Value_not_set','sql_language_implementation-charactervarying',1,0,0))
attributes_sql_languages.append(('sql_language_binding_style','charactervarying','Value_not_set','sql_language_binding_style-charactervarying',1,0,0))
attributes_sql_languages.append(('sql_language_programming_language','charactervarying','Value_not_set','sql_language_programming_language-charactervarying',1,0,0))
attributes_sql_packages.append(('feature_id','charactervarying','Value_not_set','feature_id-charactervarying',1,0,0))
attributes_sql_packages.append(('feature_name','charactervarying','Value_not_set','feature_name-charactervarying',1,0,0))
attributes_sql_packages.append(('is_supported','charactervarying','Value_not_set','is_supported-charactervarying',1,0,0))
attributes_sql_packages.append(('is_verified_by','charactervarying','Value_not_set','is_verified_by-charactervarying',1,0,0))
attributes_sql_packages.append(('comments','charactervarying','Value_not_set','comments-charactervarying',1,0,0))
attributes_sql_parts.append(('feature_id','charactervarying','Value_not_set','feature_id-charactervarying',1,0,0))
attributes_sql_parts.append(('feature_name','charactervarying','Value_not_set','feature_name-charactervarying',1,0,0))
attributes_sql_parts.append(('is_supported','charactervarying','Value_not_set','is_supported-charactervarying',1,0,0))
attributes_sql_parts.append(('is_verified_by','charactervarying','Value_not_set','is_verified_by-charactervarying',1,0,0))
attributes_sql_parts.append(('comments','charactervarying','Value_not_set','comments-charactervarying',1,0,0))
attributes_sql_sizing_profiles.append(('sizing_id','integer','Value_not_set','sizing_id-integer',1,0,0))
attributes_sql_sizing_profiles.append(('sizing_name','charactervarying','Value_not_set','sizing_name-charactervarying',1,0,0))
attributes_sql_sizing_profiles.append(('profile_id','charactervarying','Value_not_set','profile_id-charactervarying',1,0,0))
attributes_sql_sizing_profiles.append(('required_value','integer','Value_not_set','required_value-integer',1,0,0))
```

```

attributes_sql_sizing_profiles.append(('comments','charactervarying','Value_not_set','comments-charactervarying',1,0,0))
attributes_sql_features.append(('feature_id','charactervarying','Value_not_set','feature_id-charactervarying',1,0,0))
attributes_sql_features.append(('feature_name','charactervarying','Value_not_set','feature_name-charactervarying',1,0,0))
attributes_sql_features.append(('sub_feature_id','charactervarying','Value_not_set','sub_feature_id-charactervarying',1,0,0))
attributes_sql_features.append(('sub_feature_name','charactervarying','Value_not_set','sub_feature_name-charactervarying',1,0,0))
attributes_sql_features.append(('is_supported','charactervarying','Value_not_set','is_supported-charactervarying',1,0,0))
attributes_sql_features.append(('is_verified_by','charactervarying','Value_not_set','is_verified_by-charactervarying',1,0,0))
attributes_sql_features.append(('comments','charactervarying','Value_not_set','comments-charactervarying',1,0,0))
attributes_sql_sizing.append(('sizing_id','integer','Value_not_set','sizing_id-integer',1,0,0))
attributes_sql_sizing.append(('sizing_name','charactervarying','Value_not_set','sizing_name-charactervarying',1,0,0))
attributes_sql_sizing.append(('supported_value','integer','Value_not_set','supported_value-integer',1,0,0))
attributes_sql_sizing.append(('comments','charactervarying','Value_not_set','comments-charactervarying',1,0,0))
sql_implementation_info.properties["attributes"]=attributes_sql_implementation_info
sql_sizing_profiles.properties["attributes"]=attributes_sql_sizing_profiles
sql_features.properties["attributes"]=attributes_sql_features
sql_languages.properties["attributes"]=attributes_sql_languages
sql_packages.properties["attributes"]=attributes_sql_packages
sql_parts.properties["attributes"]=attributes_sql_parts
sql_sizing.properties["attributes"]=attributes_sql_sizing
layer.add_object(sql_implementation_info)
layer.add_object(sql_sizing_profiles)
layer.add_object(sql_features)
layer.add_object(sql_languages)
layer.add_object(sql_packages)
layer.add_object(sql_parts)
layer.add_object(sql_sizing)
oType=dia.get_object_type("UML - Activity")
oend,h1,h2=oType.create(0,-1)
theObjects.append(oend)
oend.properties["text"]="INFORMATION_SCHEMA: Claudio Bezerra Leopoldino - Meu
Blog de PostgreSQL - http://postgresqlbr.blogspot.com.br/ -
claudiob_br@yahoo.com.br"
layer.add_object(oend)
diagram.save()
print"GenerationFINISHED"

```

6 - Limitações e Considerações Finais

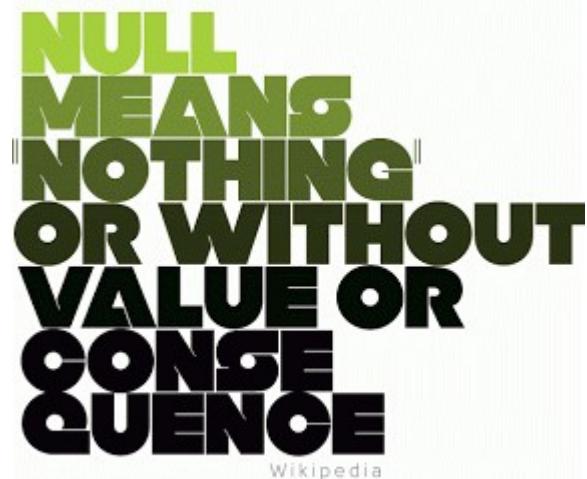
O Dia é mais uma ferramenta de modelagem de diagramas visuais que uma ferramenta de banco de dados, e esta não é uma limitação pequena, apesar da grande flexibilidade em se gerar bons resultados através de scripts. Não espere a geração de SQL, ou outras rotinas avançadas em uma ferramenta de diagramas!

O nosso script não faz o layout automático levando em conta os relacionamentos. Nos casos mostrados, não existem relacionamentos na forma foreign keys entre as tabelas de sistema, mas isso pode ser necessário para a sua necessidade. Implemente!

Você está convidado a indicar melhorias nos códigos aqui apresentados, criar seus diagramas, testar novos diagramas, e scripts, e a compartilhá-los!

Você REALMENTE Sabe Lidar com Valores Nulos no PostgreSQL?

O título desta postagem é uma pergunta que me fiz. Eu achava que sim, mas ao fazer uns testes e pesquisas, vi que o assunto é mais amplo do que eu pensava inicialmente. Convido o leitor a explorar todas as principais variações do tratamento de valores nulos no postgresql.



* **Primeiro ponto: Nulo não é zero, nem "", nem espaço.**

Nulo significa nulo, que o valor não foi preenchido. Vejamos o exemplo abaixo:

```
teste=# SELECT (null = '') IS TRUE AS COMPARA1, (null = ' ') IS
TRUE AS COMPARA2, (null = 0) IS TRUE AS COMPARA3;
compara1 | compara2 | compara3
-----+-----+-----
f | f | f
```

```
(1 registro)
```

*** Saiba testar se um valor é nulo ou não (IN NULL e IS NOT NULL)**

O exemplo abaixo utiliza IS NULL e IS NOT NULL:

```
teste=# SELECT (1 IS NULL) AS COMPARA1, (1 IS NOT NULL) AS
COMPARA2;
compara1 | compara2
-----+-----
f | t
(1 registro)
```

*** Saiba que valores nulos retornam nulo em comparações (IN, NOT IN, =, !=, etc.)**

Esta é uma decorrência do padrão SQL.

```
teste=# SELECT null NOT IN (1,2,3), null IN (1,2,3);
?column? | ?column?
-----+-----
| 
(1 registro)

teste=# SELECT 1 NOT IN (null), 1 IN (null);
?column? | ?column?
-----+-----
| 
(1 registro)

teste=# SELECT 1 != null, 1 = null;
?column? | ?column?
-----+-----
| 
(1 registro)
```

*** Criar tabelas que aceitem e rejeitem campos nulos é importante!**

```
teste=# CREATE TABLE TSTNULL (campo1 INTEGER NOT NULL, campo2
INTEGER NULL, campo3 INTEGER);
CREATE TABLE
```

*** Como manipular valores nulos e não nulos nas tabelas criadas?**

Use o valor null para deixar o campo desejado sem valor algum. Caso se utilize DEFAULT, o campo assume valor nulo se não há um valor default definido.

```
teste=# INSERT INTO TSTNULL VALUES (1, null, DEFAULT); --O DEFAULT
é NULO, CASO NÃO FORNECIDO
INSERT 0 1
teste=# INSERT INTO TSTNULL VALUES (null, 1, 1);
ERRO: valor nulo na coluna "campo1" viola a restrição não-nula

teste=
teste# SELECT * FROM TSTNULL;
campo1 | campo2 | campo3
-----+-----+-----
 1 | |
(1 registro)
```

* Entenda que campos UNIQUE aceitam vários valores nulos.

```
teste=# CREATE TABLE TSTNULL2 (campo1 INTEGER UNIQUE);
NOTA: CREATE TABLE / UNIQUE criará índice implícito
"tstnull2_campo1_key" na tabela "tstnull2"
CREATE TABLE
teste=
teste# INSERT INTO TSTNULL2 VALUES (1);
INSERT 0 1
teste# INSERT INTO TSTNULL2 VALUES (NULL);
INSERT 0 1
teste# INSERT INTO TSTNULL2 VALUES (NULL); --NAO GERA ERRO
INSERT 0 1
teste=
teste# SELECT * FROM TSTNULL2;
campo1
-----
 1

(3 registros)
```

* Campos PRIMARY KEY não aceitam nulos por definição. Você sabia?

```
teste# --CAMPOS PRIMARY KEY NÃO ACEITAM NULOS
teste# CREATE TABLE TSTNULL3 (campo1 INTEGER PRIMARY KEY);
```

```

NOTA: CREATE TABLE / PRIMARY KEY criará índice implícito
"tstnull3_pkey" na tabela "tstnull3"
CREATE TABLE
teste=# 
teste=# INSERT INTO TSTNULL3 VALUES (1);
INSERT 0 1
teste=# INSERT INTO TSTNULL3 VALUES (NULL); --ERRO
ERRO: valor nulo na coluna "campo1" viola a restrição não-nula
teste=#

```

* Campos PRIMARY KEY com mais de um campo também não aceitam nulos.

```

teste=# CREATE TABLE TSTNULL4 (campo1 INTEGER, campo2 INTEGER,
teste(# CONSTRAINT PK_TESTNULL4 PRIMARY KEY (campo1, campo2));
NOTA: CREATE TABLE / PRIMARY KEY criará índice implícito
"pk_testnull4" na tabela "tstnull4"
CREATE TABLE
teste=# 
teste=# INSERT INTO TSTNULL4 VALUES (1, 1);
INSERT 0 1
teste=# INSERT INTO TSTNULL4 VALUES (1, NULL); --ERRO
ERRO: valor nulo na coluna "campo2" viola a restrição não-nula
teste=#

```

* Consultas podem ordenar os valores nulos, colocando-os na frente ou atrás do resultado da consulta.

Para obter este efeito, empregue as cláusulas NULLS FIRST e NULLS LAST. O padrão é colocar os valores nulos por último no retorno da consulta.

```

teste=# SELECT * FROM TSTNULL ORDER BY CAMPO2 NULLS FIRST;
campo1 | campo2 | campo3
-----+-----+-----
1 | |
(1 registro)

```

```

teste=# SELECT * FROM TSTNULL ORDER BY CAMPO2 NULLS LAST;
campo1 | campo2 | campo3
-----+-----+-----
1 | |
(1 registro)

```

```

teste=# SELECT * FROM TSTNULL ORDER BY CAMPO1 NULLS FIRST, CAMPO2

```

```
NULLS LAST;
campo1 | campo2 | campo3
-----+-----+-----
 1 | |
(1 registro)
```

*** A indexação leva em conta campos nulos. Você sabia?**

Para influenciar este mecanismo, utilize as cláusulas NULLS FIRST e NULLS LAST.

```
teste=# CREATE INDEX idx_campo1_first ON TSTNULL (campo1 NULLS
FIRST);
CREATE INDEX
```

```
teste=# CREATE INDEX idx_campo1_last ON TSTNULL (campo1 NULLS
LAST);
CREATE INDEX
```

```
teste=# CREATE INDEX idx_campo1_2 ON TSTNULL (campo1 NULLS LAST,
campo2 NULLS FIRST);
CREATE INDEX
```

*** Troque os nulos por um valor padrão usando o comando COPY.**

Tanto no COPY FROM quanto no COPY TO, você pode substituir os nulos por um valor mais palatável, utilizando a cláusula NULL. Não deixe de indicar o valor que substituirá os nulos.

```
teste=# COPY TSTNULL TO 'tstnulos.txt' NULL 'NULO' CSV;
teste=# COPY TSTNULL FROM 'tstnulos.txt' NULL 'NULO' CSV;
```

*** É possível "Forçar" valores importados para valores not null, no comando COPY FROM de arquivos CSV.**

Basta utilizar a cláusula cláusula FORCE_NOT_NULL, a partir da versão 9.0 do postgresql. Esta cláusula transforma valores nulos em strings de tamanho zero.

```
teste=# COPY TSTNULL FROM 'tstnulos.txt' (format csv,
FORCE_NOT_NULL(campo1)); -- VALORES NULOS DE CAMPO1 SÃO LIDOS COMO
STRINGS DE TAMANHO ZERO, NÃO COMO NULOS
```

*** Você pode tratar parâmetros nulos nas funções com as cláusulas "CALLED ON**

"NULL INPUT", "RETURNS NULL ON NULL INPUT" ou "STRICT".

Quando desejar aceitar tratar parâmetros com valor nulo em uma função, utilize a cláusula "CALLED ON NULL INPUT". O postgresql não vai retornar erro ou fazer qualquer tratamento sobre o parâmetro, passando esta responsabilidade ao programador da função.

O uso de "RETURNS NULL ON NULL INPUT" ou "STRICT" faz com que o retorno de uma função com algum parâmetro nulo, seja **sempre null**.

O teste é bem simples. Abaixo, coloco alguns exemplos:

```
CREATE OR REPLACE FUNCTION soma_nulo_1(i integer, j integer)
RETURNS integer AS $$

BEGIN
    IF i IS NULL OR j IS NULL THEN
        RETURN 0;
    ELSE
        RETURN (i + j);
    END IF;
END;

$$ LANGUAGE plpgsql CALLED ON NULL INPUT;
SELECT soma_nulo_1(1,2);
SELECT soma_nulo_1(null,2);

CREATE OR REPLACE FUNCTION soma_nulo_2(i integer, j integer)
RETURNS integer AS $$

BEGIN
    IF i IS NULL OR j IS NULL THEN
        RETURN 0;
    ELSE
        RETURN (i + j);
    END IF;
END;

$$ LANGUAGE plpgsql RETURNS NULL ON NULL INPUT;
SELECT soma_nulo_2(1,2);
SELECT soma_nulo_2(null,2);

CREATE OR REPLACE FUNCTION soma_nulo_3(i integer, j integer)
RETURNS integer AS $$

BEGIN
    IF i IS NULL OR j IS NULL THEN
        RETURN 0;
```

```

ELSE
    RETURN (i + j);
END IF;
END;

$$ LANGUAGE plpgsql STRICT;
SELECT soma_nulo_3(1,2);
SELECT soma_nulo_3(null,2);

```

* Considerações Finais

Tratar valores nulos é de vital importância para evitar inconsistências. O postgresql oferece várias opções para lidar com este tipo de ocorrência.

Formatação de Data e Hora com as funções TO_CHAR e TO_DATE

Dados do tipo data e hora apresentam muitas variações de formatação. Para o usuário as datas se apresentam como campos textuais, mas para o SGBD, são tipos de dados específicos, com características próprias, representando valores temporais. Para trabalhar com valores deste tipo, os sistemas fazem frequentemente conversões entre esses dois tipos.

Este post apresenta o uso das funções TO_DATE e TO_CHAR para formatar e tratar datas e horas.

* TO_CHAR - Formatando campos do tipo data para texto.

- Exemplo 1: Formatação padrão

```

postgres=# SELECT TO_CHAR(current_date, 'DD-MM-YYYY');
      to_char
-----
17-06-2013
(1 registro)

```

- Exemplo 2: Formatação padrão utilizando o separador "/"

```

postgres=# SELECT TO_CHAR(current_date, 'DD/MM/YYYY');
      to_char
-----
17/06/2013
(1 registro)

```

- Exemplo 3: Alterando a ordem de visualização do mês, dia e ano apresentados.

```

postgres=# SELECT TO_CHAR(current_date, 'DD-MM-YYYY'),

```

```
TO_CHAR(current_date, 'MM-DD-YYYY') , TO_CHAR(current_date, 'YYYY-MM-YY') ;
```

```
to_char | to_char | to_char
```

```
-----+-----+-----
```

```
17-06-2013 | 06-17-2013 | 2013-06-13
```

```
(1 registro)
```

- Exemplo 4: Separando mês, dia e ano em campos distintos.

```
postgres=# SELECT TO_CHAR(current_date, 'DD') AS DIA,
TO_CHAR(current_date, 'MM') AS MES, TO_CHAR(current_date, 'YYYY') AS ANO;
```

```
dia | mes | ano
```

```
-----+-----+-----
```

```
17 | 06 | 2013
```

```
(1 registro)
```

- Exemplo 5: Apresentando o mês e o dia por extenso.

```
postgres=# SELECT TO_CHAR(current_date, 'DAY') AS DIA,
TO_CHAR(current_date, 'MON') AS MES;
```

```
dia | mes
```

```
-----+-----
```

```
MONDAY | JUN
```

```
(1 registro)
```

- Exemplo 6: Apresentando a hora do sistema formatada.

```
postgres=# SELECT TO_CHAR(current_timestamp, 'HH24:MI:SS');
```

```
to_char
```

```
-----
```

```
07:29:13
```

```
(1 registro)
```

- Exemplo 7: Separando hora, minuto e segundos em campos distintos.

```
postgres=# SELECT TO_CHAR(current_timestamp, 'HH24') ,
TO_CHAR(current_timestamp, 'MI') , TO_CHAR(current_timestamp, 'SS');
```

```
to_char | to_char | to_char
```

```
-----+-----+-----
```

```
07 | 29 | 13
```

```
(1 registro)
```

- Exemplo 8: Usando TO_CHAR para inserção e atualização de dados.

```
postgres=#
postgres=# CREATE TABLE char_date_tst (momento varchar(10));
```

```

CREATE TABLE
postgres=# INSERT INTO char_date_tst (momento) VALUES
(TO_CHAR(current_date, 'MM-DD-YYYY'));
INSERT 0 1
postgres=# UPDATE char_date_tst SET momento = '01-05-2010' WHERE
momento = TO_CHAR(current_date, 'MM-DD-YYYY');
UPDATE 1
postgres=# SELECT * FROM char_date_tst;
 momento
-----
01-05-2010
(1 registro)

```

*** TO_DATE - Formatando campos do tipo texto para data.**

- Exemplo 1: Usando TO_DATE para data em formato DD-MM-YYYY.

```

postgres=# SELECT TO_DATE('30-04-2010', 'DD-MM-YYYY');
 to_date
-----
2010-04-30
(1 registro)

```

- Exemplo 2: Usando TO_DATE para data em formato MM-DD-YYYY.

```

postgres=# SELECT TO_DATE('04-30-2010', 'MM-DD-YYYY');
 to_date
-----
2010-04-30
(1 registro)

```

- Exemplo 3: Usando TO_DATE para data com separador "/"

```

postgres=# SELECT TO_DATE('04/30/2010', 'MM/DD/YYYY');
 to_date
-----
2010-04-30
(1 registro)

```

- Exemplo 4: Teste de igualdade entre datas de formatos distintos usando TO_DATE.

```

postgres=# SELECT TO_DATE('04-30-2010', 'MM-DD-YYYY') =
TO_DATE('30-04-2010', 'DD-MM-YYYY') AS IGUALDADE;
 igualdade
-----
t
(1 registro)

```

- Exemplo 5: Usando TO_DATE para inserção e atualização de dados.

```
postgres=# CREATE TABLE date_tst (momento date);
CREATE TABLE
postgres=# INSERT INTO date_tst VALUES (TO_DATE('04-30-2010', 'MM-
DD-YYYY'));
INSERT 0 1
postgres=# UPDATE date_tst SET momento = momento + 1 WHERE momento
= TO_DATE('04-30-2010', 'MM-DD-YYYY');
UPDATE 1
postgres=# SELECT * FROM date_tst;
 momento
-----
2010-05-01
(1 registro)
* Considerações Finais
```

Os exemplos acima não cobrem todas as possibilidades, mas podem ajudar nas principais operações que lidem com a formatação de datas e horas no postgresql. A cláusula CAST é uma alternativa para conversão entre estes tipos de dados, mas não realiza formatações sobre os mesmos.

Para a tratar a formatação de timestamps e números, podem ser utilizadas as funções TO_TIMESTAMP e TO_NUMBER, que estão fora do escopo deste post.

DtSQL: Ferramenta Front-End para Banco de Dados

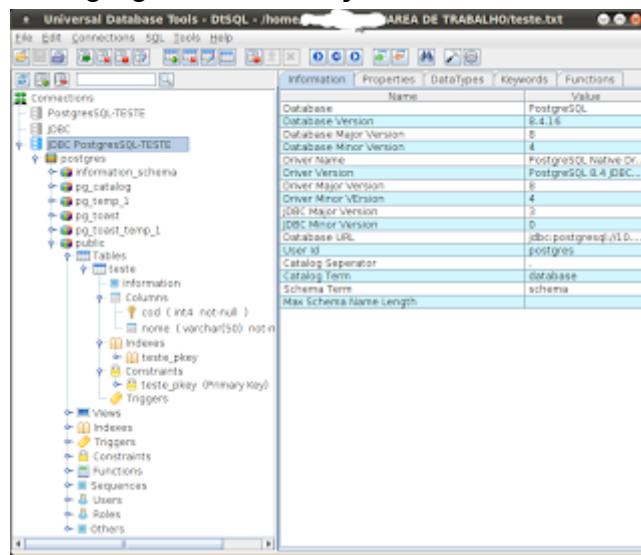
A procura por ferramentas que realmente aumentem a produtividade continua. O [DtSQL](#) apresenta um bom conjunto de funcionalidades e é compatível com o PostgreSQL e mais de 20 outros SGBDs. O DtSQL foi tornado livre em 2013, e tem sofrido atualizações recentemente, o que é um ponto positivo. Está disponível para Windows, Mac OS, Linux e UNIX.



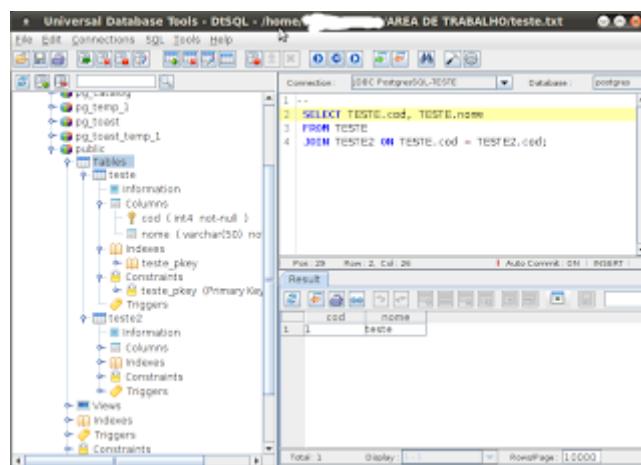
Abaixo, coloco algumas informações, sobre os recursos oferecidos, com base na versão de março deste ano:

* Interface Gráfica Simples

- A interface gráfica é simples, mas parece bastante com outras já bastante conhecidas, como a do PgAdmin e a do Squirrel. Ao mesmo que isso facilita a utilização, deixa a impressão de que poderia agregar mais inovação e valor à ferramenta.

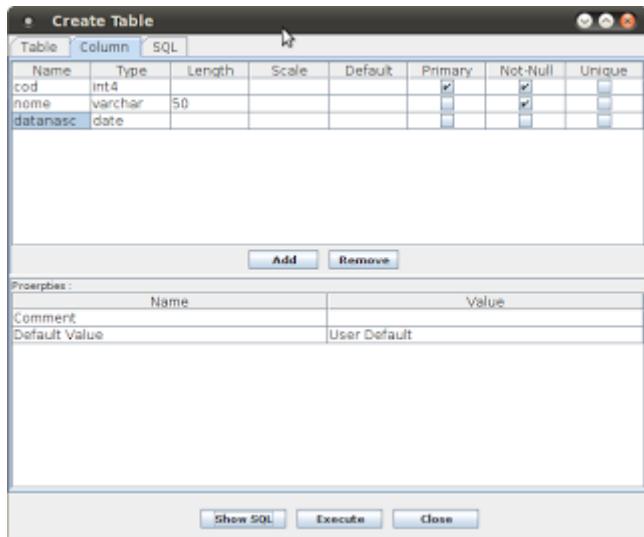


No entanto a interface não é perfeita. Para criar e executar comandos no editor de texto, deve-se clicar no banco de dados ou em um objeto (tabela ou visão). Demorei para descobrir este recurso, então creio que a interface não seja tão intuitiva quanto poderia.



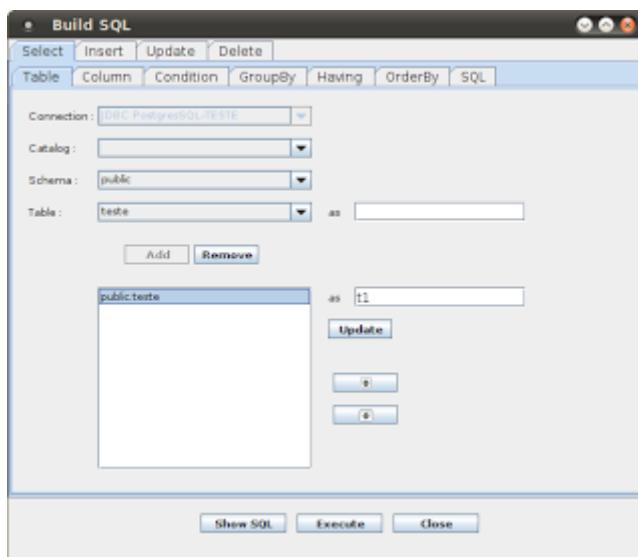
* Assistentes

- Tarefas como a criação de tabelas, visões, índices e outros objetos são automatizadas por maio de assistentes que facilitam o trabalho.



* Query Builder (Confuso)

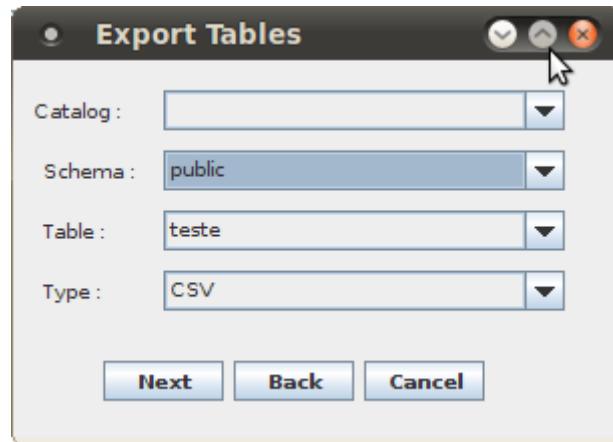
- Não posso dizer que gostei do Query Builder. Achei confuso e complicado. Mas recomendo que seja testado, pois pode ser exatamente o que você procura.



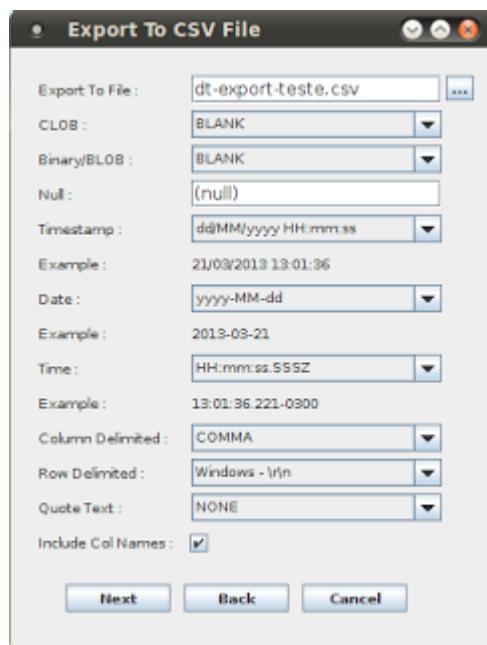
* Exportação/ Importação de Dados

- A exportação é a melhor feature da ferramenta. Assistentes permitem a exportação e importação em poucos cliques.

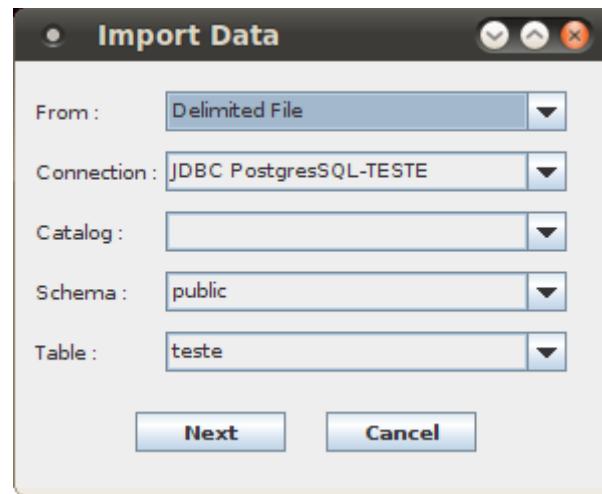
Não percebi bugs ou outros problemas no processo, mas recomendo testes para uso em bancos de dados de maiores proporções.



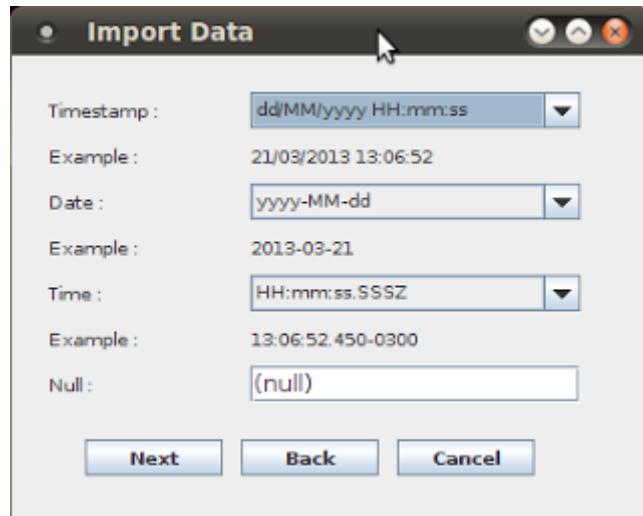
Exportação: Seleção de tabela a ser exportada.



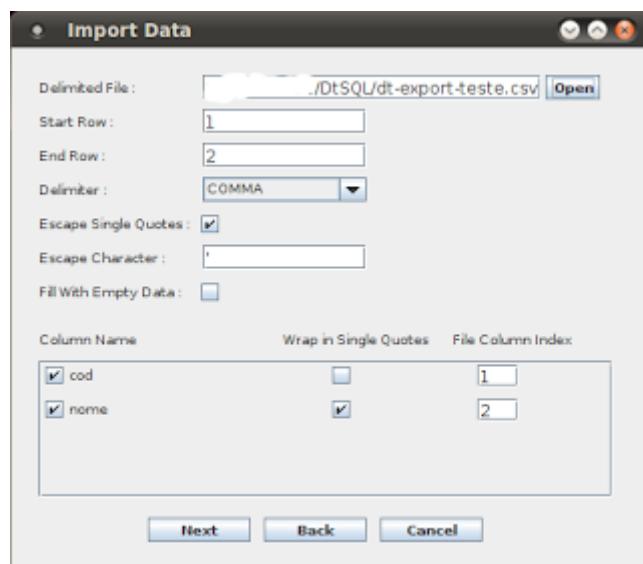
Exportação: Parâmetros



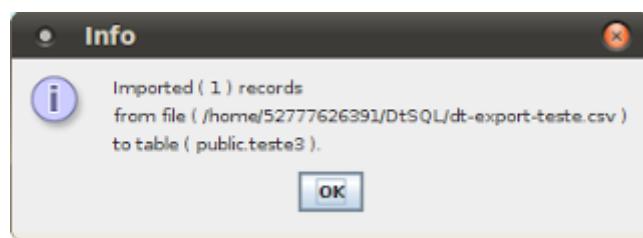
Importação: Definição de Tabela para receber os dados



Importação: Parâmetros



Importação: Arquivo com dados a importar



Importação: Sumário

* Considerações Finais

A primeira impressão foi positiva, embora eu prefira soluções como o Squirrel. Creio que pode ser utilizada como front-end. Destaco os pontos positivos:

- Assistentes
- Disponibilidade para vários sistemas operacionais
- Conectividade com Múltiplos Bancos de Dados
- Recursos para importar/ exportar dados. Esta funcionalidade foi a que achei mais promissora.
- Atualizações recentes, indicando que a ferramenta não está parada.

Pontos negativos:

- Query builder confuso.
- Falta de recursos mais avançados como navegação gráfica nos dados, engenharia reversa, diagramação, monitoramento, etc.
- O código da ferramenta aparentemente não foi aberto.
- É mantida por uma empresa, não por uma comunidade, gerando dúvidas sobre o futuro da ferramenta.

Teste e me dê sua opinião!

Edição de SQL e Funções no PSQL

O psql é a principal interface dos desenvolvedores com o PostgreSQL. No entanto, editar códigos no psql pode ser uma tarefa onerosa. As consultas e funções podem ser extensas e o trabalho se tornar cansativo e improdutivo. Existem algumas opções que podem ajudar a trabalhar melhor com os códigos, sem precisar sair do PSQL, que abordamos resumidamente aqui.

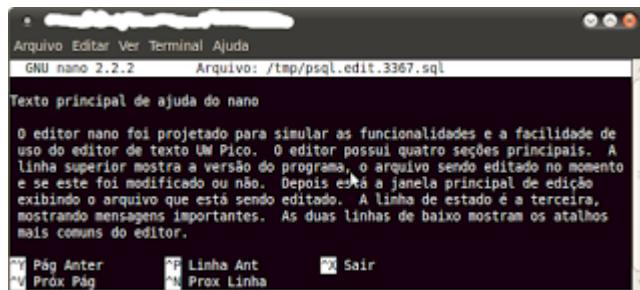
- Executando arquivos TXT salvos previamente.

A maneira que mais utilizo para trabalhar com funções e SQL é executar arquivos txt salvos previamente. Gosto de ter scripts para as necessidades básicas em seus respectivos lugares, que possam ser reutilizados, e tem sido bem útil trabalhar desta forma. Edite o seu SQL no editor que achar melhor e salve-o. Para executá-lo, utilize a sintaxe:

```
\i (nome do arquivo como código SQL)
```

- Editar código SQL usando o Editor Padrão

Neste caso, o sistema abre o editor padrão definido para o postgresql. Caso não exista um editor definido, o postgres perguntará, dentre os disponíveis, qual você deseja utilizar. No meu caso, utilizo o nano (<http://www.nano-editor.org/>). É um editor bem simples e fácil de usar, que apresenta boas teclas de atalho.

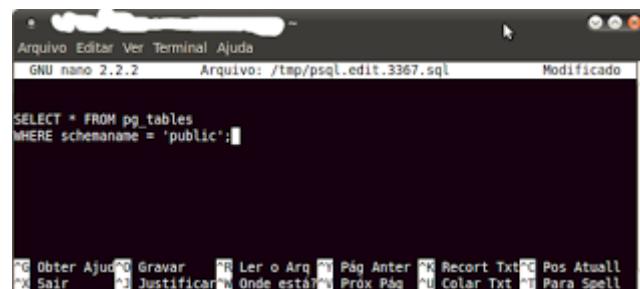


The screenshot shows the nano text editor's help screen. The title bar reads "Arquivo Editar Ver Terminal Ajuda" and "GNU nano 2.2.2 Arquivo: /tmp/pgsql.edit.3367.sql". The main text area displays the "Texto principal de ajuda do nano" (Main help text of the nano) which describes the editor's features and keyboard shortcuts. The status bar at the bottom shows various keyboard shortcuts.

Digite:

\e (ou \edit)

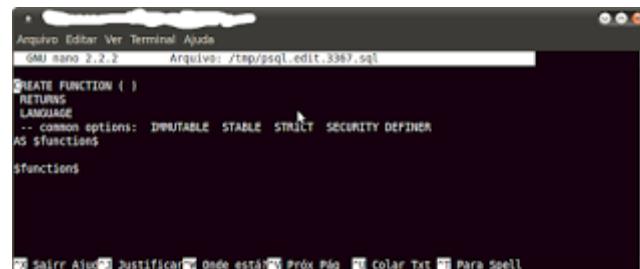
O sistema abre a tela do editor para inserir e editar seu texto, permitindo rolar as páginas e manter o SQL sem problemas. É possível salvar o script para reutilização. Para apenas executar, sem salvar, basta sair do editor. No caso do NANO, teclando CONTROL+X.



The screenshot shows the nano text editor with a SQL query in the buffer. The title bar reads "Arquivo Editar Ver Terminal Ajuda" and "GNU nano 2.2.2 Arquivo: /tmp/pgsql.edit.3367.sql Modificado". The main text area contains the following SQL code: "SELECT * FROM pg_tables WHERE schemaname = 'public';". The status bar at the bottom shows various keyboard shortcuts.

- Editar nova função no editor.

Ao se digitar **\ef**, o psql abre o Editor predeterminado, apresentando um "esqueleto de função" para edição. Basta sair teclando CONTROL+X para criar a função.



The screenshot shows the nano text editor with a function creation script in the buffer. The title bar reads "Arquivo Editar Ver Terminal Ajuda" and "GNU nano 2.2.2 Arquivo: /tmp/pgsql.edit.3367.sql". The main text area contains the following SQL code: "CREATE FUNCTION () RETURNS LANGUAGE -- common options: IMMUTABLE STABLE STRICT SECURITY DEFINER AS \$function\$functions\$". The status bar at the bottom shows various keyboard shortcuts.

- Editar função existente

Neste caso, utilize a sintaxe:

\ef (nome da função a editar)

```

Arquivo Editar Ver Terminal Ajuda
GNU nano 2.2.2 Arquivo: /tmp/psql.edit.5118.sql
CREATE OR REPLACE FUNCTION public.pegar_codigo()
RETURNS trigger
LANGUAGE plpgsql
AS $functions$
BEGIN
UPDATE lista_codigos SET flag_utilizado = true WHERE codigo = new.codigo;
RETURN new.codigo;
END;
$functions$
```

Se você esqueceu o nome das funções que deseja editar, pode adaptar a consulta abaixo para descobrir.

```
SELECT proname, pronamespace, proowner FROM PG_PROC;
```

Agora é editar seus scripts, consultas e funções de dentro do psql!

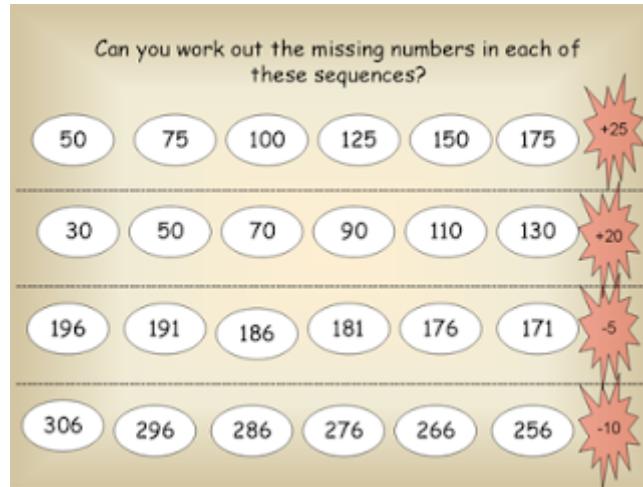
OOPS! Alterando o Editor Padrão!

Já ia me esquecendo! Altere o editor padrão utilizando a sintaxe abaixo:

```
\set PSQL_EDITOR (caminho do editor)
```

Produza Sequências Com a Função Generate_Series()!

A geração de séries numéricas e temporais tem diversas aplicações em bancos de dados. A produção de uma sequência de inteiros pode ser de grande valia para solucionar vários tipos de problemas, enquanto que uma lista de datas que seja produzida pode permitir o agendamento de tarefas, por exemplo. A função generate_series, implementada pelo postgres, permite a geração de diversas séries sem a necessidade de construção de programas ou funções iterativas, o que economiza esforço de programação.



Exemplos de sequências numéricas.

A função generate_series assume três grandes formas:

- **generate_series(valor inicial, valor final)** - Gera uma série numérica de valores, partindo do valor inicial ao final, utilizando como incremento o valor 1;
- **generate_series(valor inicial, valor final, incremento)** - Gera uma série de valores, partindo do valor inicial ao final, utilizando como incremento o valor parametrizado. Produz uma progressão aritmética;
- **generate_series(valor inicial, valor final, incremento do tipo intervalar)** - Gera uma série temporal de valores, partindo do valor inicial ao final, ambos do tipo timestamp, utilizando como incremento o valor parametrizado.

Abaixo comento algumas das possibilidades oferecidas por estas funções:

* Sequências Simples

Abaixo, sequências numéricas simples que utilizam o incremento 1.

- Sequência simples.

```
postgres=# SELECT generate_series(1,3);
generate_series
-----
1
2
3
(3 registros)
```

- Sequência simulando incremento de 3 unidades.

```
postgres=# SELECT generate_series(1,5)*3-2 AS TRIPLA;
tripla
-----
1
4
7
10
13
```

(5 registros) - Sequência com incremento fracionário.

```
postgres=# SELECT (generate_series(1,5)*1.0)/2 AS FRACIONARIO;
fracionario
```

```
-----
0.5000000000000000
1.0000000000000000
1.5000000000000000
2.0000000000000000
2.5000000000000000
```

(5 registros)

- Sequência com valores repetidos, utilizando o operador de resto da divisão.

```
postgres=# SELECT generate_series(1,10)%5 AS REPETIDO;
repetido
```

```
-----
1
2
3
4
0
1 (REPETIÇÕES)
2
3
4
0
```

(10 registros)

* Sequências Com Incremento Explícito.

- Incremento 1, fornecido.

```
postgres=# SELECT generate_series(1,5,1);
generate_series
```

```

1
2
3
4
5
(5 registros)

```

- Incremento 2, fornecido. Observe que se o valor máximo é atingido, a sequência é interrompida.

```
postgres=# SELECT generate_series(1,5,2);
```

```
generate_series
```

```
-----
1
3
5
(3 registros)
```

- Sequência com incremento decrescente.

```
postgres=# SELECT generate_series(5,1,-1);
```

```
generate_series
```

```
-----
5
4
3
2
1
(5 registros)
```

* Sequências Temporais.

Exigem um pouco mais de abstração por envolverem intervalos de tempo, mas não são necessariamente complexas. Abaixo elenco alguns exemplos elementares.

- Utilizando timestamps com a sintaxe mais básica.

```
postgres=# SELECT generate_series('2013-02-06 12:00'::timestamp,
```

```
postgres(# '2013-02-08 12:00'::timestamp,
```

```
postgres(# '1 day');
```

```
generate_series
```

```
-----
2013-02-06 12:00:00
2013-02-07 12:00:00
2013-02-08 12:00:00
```

(3 registros)

- Utilizando current_timestamp.

```
postgres=# SELECT generate_series(current_timestamp,
postgres(# current_timestamp + '5 days',
postgres(# '1 day');
generate_series
-----
2013-02-06 09:35:31.343344-03
2013-02-07 09:35:31.343344-03
2013-02-08 09:35:31.343344-03
2013-02-09 09:35:31.343344-03
2013-02-10 09:35:31.343344-03
2013-02-11 09:35:31.343344-03
```

(6 registros)

- Utilizando incremento de algumas horas.

```
postgres=# SELECT generate_series(current_timestamp,
postgres(# current_timestamp + '1 day',
postgres(# '8 hours');
generate_series
-----
2013-02-06 09:41:29.799331-03
2013-02-06 17:41:29.799331-03
2013-02-07 01:41:29.799331-03
2013-02-07 09:41:29.799331-03
```

(4 registros)

- Utilizando incremento decrescente.

```
postgres=# SELECT generate_series(current_timestamp + '5 days',
postgres(# current_timestamp,
postgres(# '-1 day');
generate_series
-----
2013-02-11 09:48:26.540137-03
2013-02-10 09:48:26.540137-03
2013-02-09 09:48:26.540137-03
2013-02-08 09:48:26.540137-03
2013-02-07 09:48:26.540137-03
2013-02-06 09:48:26.540137-03
```

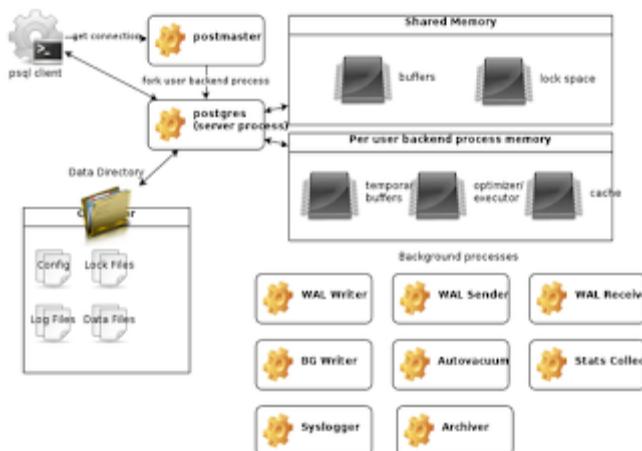
(6 registros)

* Considerações Finais

A função generate_series() permite a economia de tempo e flexibilidade na geração de séries numéricas e temporais, produzindo um resultado legível e de fácil utilização. Não são a única forma de se gerar estes dados no postgres, e afeta a portabilidade de banco de dados, mas é um recurso importante a ser considerado pelos desenvolvedores.

Visão Geral da Arquitetura do PostgreSQL!

Adorei esta imagem, com a visão geral da arquitetura do PostgreSQL. O sitio original é [este aqui](#).



PGTUNE: Otimize a configuração do PostgreSQL!

Tuning de banco de dados não é exatamente uma tarefa fácil. É necessário conhecer os rudimentos de bancos de dados, o SGBD utilizado em profundidade, linguagem SQL (ou outra, dependendo do SGBD), além de saber informações precisas sobre os bancos de dados em si e as aplicações que os utilizam.

São muitas variáveis em jogo e o processo de otimização raramente possibilita a quantidade de testes realmente necessária em virtude do tempo ser exíguo, e os sistemas não poderem parar. O [PGTUNE](#) é um script python, desenvolvido por Gregory Smith, que atua como ferramenta complementar para o tuning de bancos de dados postgresql, mais especificamente, sugerindo alterações no complexo arquivo postgresql.conf visando um melhor desempenho.

Neste post, vamos apresentar o funcionamento básico do pg_tune e elencar pontos fortes e limitações.

* Instalação

O PGTUNE demanda a [instalação do python](#), a qual é bastante simples para linux, windows e MAC (além de máquina virtual do java, .NET, etc.).

Assumindo que você já possui o postgres e o python instalados, basta acessar o sítio do [PGTUNE](#), baixar o arquivo compactado e descompactá-lo.

Para este post utilizamos:

- pgsql versão 0.9.3
- postgresql versão 9.2.1
- python versão 2.6.5

* Utilização

Abra o prompt de comando e entre na pasta descompactada do pgsql.

O comando abaixo aciona o pgsql, passando como parâmetros o arquivo postgresql.conf do postgres e o arquivo de saída (no caso, `pgsql.conf`), gerado com sugestões para o tuning do banco de dados.

```
python pgsql -i /etc/postgresql/9.2/main/postgresql.conf -o
pgsql.conf
```

O resultado da execução aparece na figura abaixo, com a listagem do arquivo de saída:

```
#default_with_oids = off
#escape_string_warning = on
#lo_compat_privileges = off
#quote_all_identifiers = off
#sql_inheritance = on
#standard_conforming_strings = on
#synchronize_segsync = on

# - Other Platforms and Clients -
#transform_null_equals = off

#-----
# ERROR HANDLING
#-----

#exit_on_error = off          # terminate session on any error?
#restart_after_crash = on     # reinitialize after backend crash?

#-----
# CUSTOMIZED OPTIONS
#-----


# Add settings for extensions here
default_statistics_target = 50 # pg_tune wizard 2012-11-09
maintenance_work_mem = 4896MB # pg_tune wizard 2012-11-09
constraint_exclusion = on # pg_tune wizard 2012-11-09
checkpoint_completion_target = 0.9 # pg_tune wizard 2012-11-09
effective_cache_size = 5632MB # pg_tune wizard 2012-11-09
work_mem = 48MB # pg_tune wizard 2012-11-09
wal_buffers = 8MB # pg_tune wizard 2012-11-09
checkpoint_segments = 36 # pg_tune wizard 2012-11-09
shared_buffers = 1920MB # pg_tune wizard 2012-11-09
max_connections = 80 # pg_tune wizard 2012-11-09
```

Caso deseje testar as alterações, altere o postgresql.conf e reinicie o servidor do banco.

Além dos arquivos de entrada e saída, o script apresenta outros recursos realmente interessantes, através de parâmetros (opcionais) de execução:

- * **-M ou –memory:** Utilizar este parâmetro para fornecer a memória do servidor. Caso não esteja especificado, o pgTune tentará detectar a memória da máquina de execução do script, assumindo que é a máquina servidora.
- * **-T ou –type:** Especifica o tipo de base de dados. As opções aceitas são:
 - DW - Grandes massas de dados com poucas alterações mas com consultas extremamente complexas
 - OLTP - Sistema tradicional de processamento de transações
 - Web - Sistema web, com grande número de acessos concorrentes
 - Mixed - Sistema com características intermediárias em relação aos demais tipos.
 - Desktop - Sistema monousuário
- * **-c or –connections:** Especifica o máximo de conexões desejado. Este é o parâmetro mais relevante em sistemas com grande número de usuários.

Exemplos adicionais com os parâmetros:

```
python pgTune -i /etc/postgresql/9.2/main/postgresql.conf -o
pgsql.conf -M 500000
```

```
python pgTune -i /etc/postgresql/9.2/main/postgresql.conf -o
pgsql.conf -c 1000
```

```
pgTune-0.9.3$ python pgTune -i
/etc/postgresql/9.2/main/postgresql.conf -o psql.conf -M 500000
-c1000
```

* Considerações finais

O pgTune mostrou que pode ser utilizado sem problemas **como ferramenta complementar** de tuning de banco de dados. Recomendo que antes de qualquer alteração no postgresql.conf, que se faça testes e backup do arquivo antigo.

Abaixo, listo pontos fortes e limitações:

Vantagens:

- Facilidade de instalação e utilização
- Ser multiplataforma, por utilizar python
- Regras de alteração de valores são acessíveis dentro do script

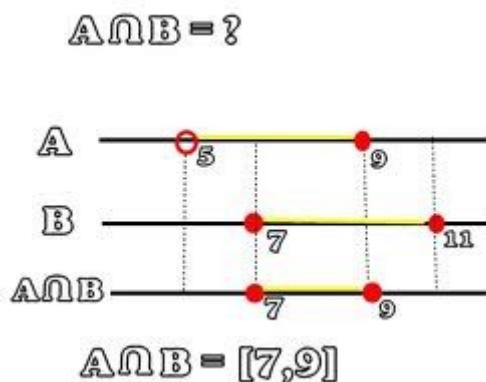
- Facilidade de interpretação de resultados
- Pode ser utilizado sem interromper o funcionamento do servidor de banco de dados
- Não altera o ambiente de produção, deixando para o DBA a opção de aceitar ou não as alterações sugeridas

Limitações:

- Não faz testes aprofundados no ambiente que indiquem a eficácia da otimização
- Funcionar através de cálculos sobre regras relacionadas à configuração, e não realiza testes de simulação comprobatórios
- Não considera outros fatores além da configuração do postgresql.conf
- Não existe uma forma clara de distinguir entre os diversos tipos de sistema, e um servidor pode ter mais de um banco de dados com tipos diferentes, o que invalidaria a configuração com o parâmetro **-T**.
- O parâmetro **-M** não aceita abreviações como 4gb, 5000mb.

Range Types: Novo recurso do Postgresql 9.2!

Início e fim, começo e encerramento. Armazenar intervalos de valores é uma tarefa importante que já estava disponível no Postgresql, porém de modo mais dispendioso em termos de programação. Era possível por exemplo criar dois campos indicando os extremos de um intervalo sem problemas, implementar uma função ou ainda criar um tipo intervalar com o comando CREATE TYPE.



Representação Matemática de Intersecção de Intervalos de Valores

A versão 9.2 apresenta o conceito de "Range Types", que engloba um tipo de dados específico para intervalos, além de recursos para cálculos e manipulações relacionadas a estes tipos peculiares de dados. Considero este um grande avanço, que pode reduzir o esforço de implementação e aumentar o desempenho em várias situações. Pode-se por exemplo, indexar um campo intervalar.

Neste post, embora não se busque esgotar o tema, vamos ilustrar as principais possibilidades desta nova feature com exemplos.

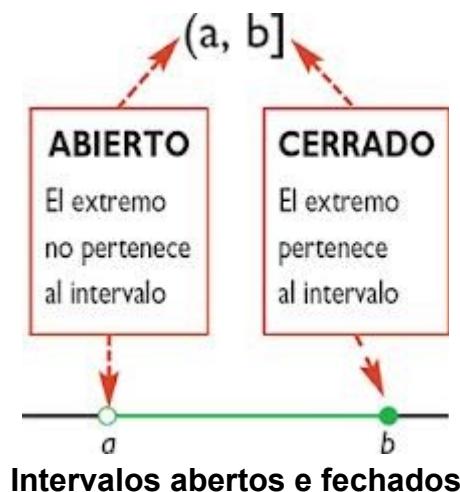
* Tipos de Intervalos

O postgresql apresenta seis tipos de intervalos padrão, sendo alguns discretos e outros contínuos, mas você pode criar outros utilizando CREATE TYPE:

- int4range — Intervalo de inteiros de 4 bytes.
- int8range — Intervalo de "bigint" (inteiros de 8 bytes)
- numrange — Intervalo de números reais
- tsrange — Intervalo de timestamps sem time zone
- tstzrange — Intervalo de timestamps com time zone
- daterange — Intervalo de datas

* Operações Básicas para Definir Dados Intervalares

Um intervalo consiste em um conjunto de valores delimitados por um valor inicial e um final. O postgres oferece opção de se trabalhar com intervalos total e parcialmente limitados e não limitados. Os delimitadores compreendem colchetes "[]" para os intervalos fechados e parênteses "(") para os intervalos abertos.



Abaixo segue uma sequência de exemplos ilustrando as principais operações feitas com intervalos e os resultados obtidos.

Exemplo 1: Intervalo fechado contendo os números 1 a 5.

```
postgres=# SELECT '[1,5] '::numrange;
numrange
-----
[1,5]
(1 registro)
```

Exemplo 2: Intervalo fechado em campo inteiro, contendo os números de 1 a 5.

```
postgres=# SELECT '[1,5]'::int4range;
int4range
-----
[1,6)
(1 registro)
```

Exemplo 3: Intervalo de 1 a 5, aberto no 5, isto é, não contendo o valor 5.

```
postgres=# SELECT '[1,5)'::numrange;
numrange
-----
[1,5)
(1 registro)
```

Exemplo 4: Intervalo sem limite máximo.

```
postgres=# SELECT '[1,'::numrange;
numrange
-----
[1,
(1 registro)
```

Exemplo 5: Intervalo sem limite mínimo.

```
postgres=# SELECT '(,1]'::numrange;
numrange
-----
(,1]
(1 registro)
```

Exemplo 6: Duas formas de expressar um intervalo sem quaisquer limites.

```
postgres=# SELECT '(,)'::numrange, numrange(null, null);
numrange | numrange
-----+-----
(, ) | (, )
(1 registro)
```

Exemplo 7: Intervalos de 1 a 5, abertos e fechados, criados a partir do construtor do tipo numrange.

```
postgres=# SELECT numrange(1,5,'[]'), 
numrange(1,5,'()' ), numrange(1,5,'[]' ), numrange(1,5,'()' );
numrange | numrange | numrange | numrange
-----+-----+-----+-----
[1,5) | (1,5] | [1,5] | (1,5)
(1 registro)
```

Exemplo 8: Duas formas de expressar um intervalo sem elementos.

```
postgres=# SELECT numrange(1,1,'[]'), '[1,1)::numrange;
numrange | numrange
-----+-----
empty | empty
(1 registro)
```

Exemplo 9: Intervalo em campo data.

```
postgres=# SELECT daterange('12/01/2012',current_date,'[]');
daterange
-----
[2012-01-12,2012-10-17)
(1 registro)
```

Exemplo 10: Intervalo em campo data, exemplo 2.

```
postgres=# SELECT daterange(current_date -10,current_date,'[]');
daterange
-----
[2012-10-07,2012-10-17)
(1 registro)
```

Exemplo 11: Teste de pertencimento de elemento a um intervalo.

```
postgres=# SELECT int4range(10, 20, '[]') @> 9, int4range(10,
20,'[]') @> 15, int4range(10, 20,'[]') @> 21;
?column? | ?column? | ?column?
-----+-----+-----
f | t | f
```

Exemplo 12: Recuperando Limites Superior e Inferior de um Intervalo

```
postgres=# SELECT lower(numrange(10, 100, '[]')), upper(numrange(10, 100, '[]'));
lower | upper
-----+-----
10 | 100
(1 registro)
```

Exemplo 13: Verificação de sobreposição de intervalos

É feita com o operador **&&**.

```
postgres=# SELECT numrange(10, 20) && numrange(25, 30),
daterange('01/01/2010', '31/12/2011') && daterange('01/01/2011',
```

```
'31/12/2012');
?column? | ?column?
-----+-----
f | t
(1 registro)
```

Exemplo 14: Intersecção de Intervalos

O teste é feito com o operador *, retornando 'empty' caso a intersecção não apresente elementos.

```
postgres=# SELECT int4range(10, 20,'[]') * int4range(15,
25,'[]'),daterange('2010-01-01','2012-06-30','[]') *
daterange('2011-01-01','2012-12-30','[]');
?column? | ?column?
-----+-----
[15,21) | [2011-01-01,2012-07-01)
(1 registro)
```

```
postgres=# SELECT int4range(10, 20) * int4range(25, 35);
?column?
-----+
empty
(1 registro)
```

Exemplo 15: Teste se intervalo é vazio (empty)

```
postgres=# SELECT isempty(numrange(10, 20)), isempty(numrange(10,
10));
isempty | isempty
-----+-----
f | t
(1 registro)
```

Exemplo 16: Criação de Tabelas e Visões e Índices com Intervalos

```
postgres=# CREATE TABLE teste_range (rang_data daterange,
rang_int4 int4range, rang_int8 int8range, rang_num numrange,
rang_timestamp tsrange);
CREATE TABLE
postgres=# CREATE TABLE teste_range_2 (rang_data daterange PRIMARY
KEY, rang_int4 int4range UNIQUE, rang_int8 int8range, rang_num
numrange, rang_timestamp tsrange);
NOTA: CREATE TABLE / PRIMARY KEY criará índice implícito
```

```
"teste_range_2_pkey" na tabela "teste_range_2"
NOTA: CREATE TABLE / UNIQUE criará índice implícito
"teste_range_2_rang_int4_key" na tabela "teste_range_2"
CREATE TABLE
postgres=# CREATE OR REPLACE VIEW view_teste_range AS SELECT *
FROM teste_range ORDER BY rang_data;
CREATE VIEW
postgres=# CREATE INDEX ind_teste_range ON teste_range (rang_data,
rang_int4);
CREATE INDEX
postgres=# INSERT INTO teste_range (rang_data, rang_int4) VALUES
(daterange('2010-01-01','2012-06-30','[]'), int4range(10,
20,'[]'));
INSERT 0 1
postgres=# select * from TESTE_RANGE;
 rang_data | rang_int4 | rang_int8 | rang_num | rang_timestamp
-----+-----+-----+-----+
[2010-01-01,2012-07-01) | [10,21) | | |
(1 registro)
```

* Pontos Fortes

- Programas, funções e consultas que utilizem intervalos ficam menores.
- Opções para manipulação de intervalos são confiáveis e não demandam plug-ins ou instalação de novos componentes.

* Pontos Fracos

- Perda de compatibilidade com outros SGBDs em todas as funcionalidades que utilizem intervalos.

Tratamento de Parâmetros de Funções com PL/pgsql

Existem várias formas de se processar erros em parâmetros fornecidos a funções. Existem casos em que valores diferentes do esperado e nulos são fornecidos, o que faz com que as entradas de parâmetros devam receber um tratamento meticoloso.

Neste post vamos apresentar alguns recursos simples que podem ser utilizados para tratar parâmetros em funções no Postgresql.

* Raise Notice

Utilize Raise Notice para disparar **avisos** ao usuário da função. Estes avisos podem funcionar como advertências, apresentar informações relevantes sobre os parâmetros fornecidos e sobre a execução da função em si.

Estes avisos não interrompem a execução da função nem são considerados erros pelos aplicativos.

Exemplo 1:

```
CREATE OR REPLACE FUNCTION teste_par_1(par_1 varchar(10)) RETURNS
varchar(10) AS
$$
BEGIN
IF char_length(par_1) < 2 THEN
RAISE NOTICE 'Valor não fornecido ou muito pequeno: %', $1;
RETURN 'AVISO';
END IF;
RETURN 'OK';
END;
$$ LANGUAGE PLPGSQL;
```

```
banco=# Select teste_par_1 ('T');
NOTA: Valor não fornecido ou muito pequeno: T
teste_par_1
-----
AVISO
(1 registro)
```

* Raise Exception

Utilize Raise Exception para disparar um erro ao usuário da função acompanhado de uma mensagem explicativa. A emissão de erro interrompe a execução da função.

Exemplo 2:

```
CREATE OR REPLACE FUNCTION teste_par_2(par_2 varchar(10)) RETURNS
varchar(10) AS
$$
BEGIN
```

```

IF char_length(par_2) < 2 THEN
RAISE EXCEPTION 'Formato inválido: %',$1;
RETURN 'ERRO';
END IF;
RETURN 'OK';
END;
$$ LANGUAGE PLPGSQL;

```

```

banco=# Select teste_par_2 ('T');
ERRO: Formato inválido: T

```

* RETURNS NULL ON NULL INPUT ou STRICT

O uso da cláusula STRICT ou "RETURNS NULL ON NULL INPUT" faz com que seja retornado valor nulo caso um dos parâmetros fornecidos seja nulo. É um recurso interessante e que pode poupar tempo de processamento em funções mais elaboradas. Para que a função aceite valores nulos, existe a cláusula "CALLED ON NULL INPUT", mas a mesma é pouco utilizada por ser o comportamento default para as funções no Postgresql.

Observe no exemplo abaixo que o valor nulo (null) é diferente da string sem elementos.

Exemplo 3:

```

CREATE OR REPLACE FUNCTION teste_par_3(par_3 varchar(10)) RETURNS
varchar(10) AS
$$
BEGIN
RETURN 'OK';
END;
$$ LANGUAGE PLPGSQL RETURNS NULL ON NULL INPUT;

```

```

banco=# Select teste_par_3 (null);
teste_par_3
-----

```

(1 registro)

```

banco=# Select teste_par_3 (' ');
teste_par_3

```

```
-----
OK
(1 registro)
banco=# Select teste_par_3 ('T');
teste_par_3
-----
```

```
OK
(1 registro)
```

A utilização de várias validações conjuntamente é a melhor forma de assegurar que a função receba valores processáveis. O exemplo abaixo é uma ilustração desta necessidade.

Exemplo 4:

```
CREATE OR REPLACE FUNCTION teste_par (par.todos varchar(10))
RETURNS varchar(10) AS
$$
BEGIN
IF char_length(par.todos) <=3 THEN
  RAISE EXCEPTION 'Valor muito pequeno não nulo: %',$1;
  RETURN 'ERRO';
ELSE
  IF char_length(par.todos) <=5 THEN
    RAISE NOTICE 'Valor muito pequeno: %',$1;
    RETURN 'AVISO';
  END IF;
END IF;
RETURN 'OK';
END;
$$ LANGUAGE PLPGSQL RETURNS NULL ON NULL INPUT;

pf=# Select teste_par (null);
teste_par
-----

(1 registro)

pf=# Select teste_par ('T');
ERRO: Valor muito pequeno não nulo: T

pf=# Select teste_par ('Test');
NOTA: Valor muito pequeno: Test
teste_par
```

AVISO**(1 registro)**

Atualmente existem além de NOTICE e EXCEPTION vários outros qualificadores das mensagens: DEBUG, LOG, INFO, NOTICE, WARNING, e EXCEPTION, sendo que este último é o valor padrão.

Utilize-os nas suas validações, tentando sempre manter o código o mais simples possível!

Desenvolva suas Aplicações de Bancos Postgres com Wavemaker



Tela 1: Servidor do Wavemaker Online

Cansado de ter de programar as interfaces em Java e PHP? Ferramentas de desenvolvimento são importantes para adquirir maior produtividade e para se explorar os vários recursos dos bancos de dados. O Wavemaker é uma ferramenta de desenvolvimento que oferece bons recursos para criar e gerir aplicações web, minimizando o esforço de programação, e que apresenta plena compatibilidade com bancos de dados PostgreSQL!

É uma ferramenta livre com código aberto através de licença Apache. Neste post a preocupação não é mostrar em profundidade os recursos da ferramenta, nem criar um tutorial, mas sim apresentar as funcionalidades básicas.



Tela 2: Interface do Wavemaker

A instalação é relativamente simples, e o programa pode ser baixado em <http://www.wavemaker.com>. O wavemaker é compatível com windows, linux e macintosh.



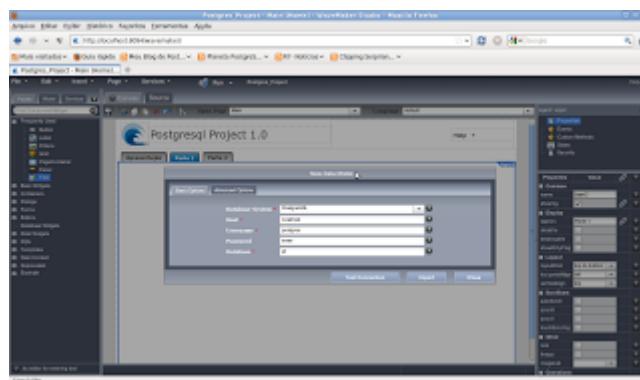
Tela 3: Criação de Projeto no Wavemaker

O Wavemaker apresenta uma interface bastante simplificada e ao mesmo tempo prática, e as operações são todas feitas dentro do navegador web. Basta se selecionar um objeto para suas propriedades estarem disponibilizadas para edição à direita da tela. A interface de programação é WYSIWYG. É uma ferramenta cliente-servidor, o que exige os devidos cuidados com a segurança em rede.

Abaixo, alguns recursos associados ao PostgreSQL:

* Importar Database

Por meio do menu "Services/ Import database" é possível recuperar todas as informações em um banco já existente. A interface é intuitiva para quem tem alguma experiência de desenvolvimento.



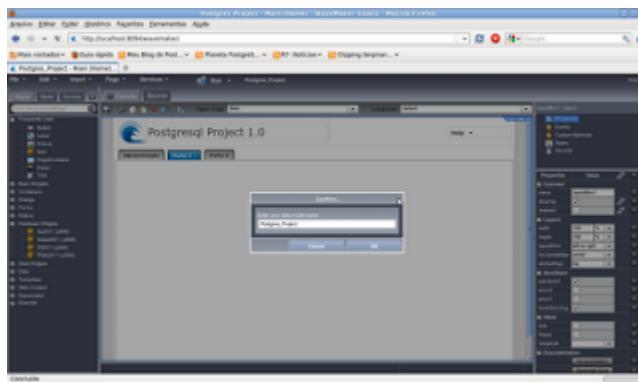
Tela 4: Importar Database

Entre com os dados do banco de dados, teste a conexão utilizando a opção "Test connection" e acione a importação do banco de dados com o botão "Import".

As tabelas importadas aparecem à esquerda da tela, na pasta "Database Widgets". É possível utilizar estas tabelas para criar formulários CRUD, consultas e relatórios, entre outras possibilidades.

* Projetar Database

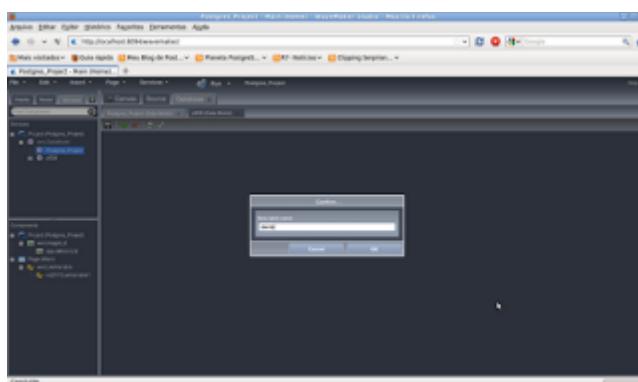
Acione a opção "Services/ Design Database" para criar suas bases de dados, tabelas e para estabelecer os relacionamentos entre as mesmas.



Tela 5: Projetar Database

Ao disparar esta opção, você define o nome do banco a ser criado e confirma. O banco aparecerá no menu à esquerda da tela.

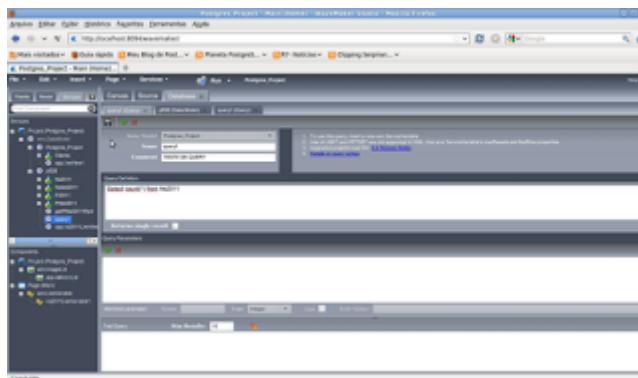
Selecione o banco e na parte central da tela aparecerão as opções de criação das tabelas do seu banco. A interface realmente é bem agradável. Clique no ícone do disquete para salvar as tabelas que for desenvolvendo.



Tela 6: Criação de Tabela

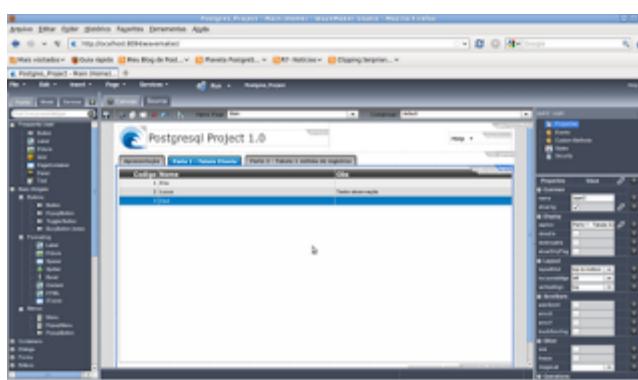
* Consultar

O menu "Services/ Query" permite que se realize e salve consultas às tabelas.



Tela 7: Construção de Consultas

A ferramenta apresenta ainda grids, treeviews, charts para apresentação dos dados, entre outras funcionalidades. É possível definir o dataset de uma grid e indicar as colunas a serem mostradas, o que facilita muito o desenvolvimento.



Tela 8: Dados de Uma Tabela

*** Pontos fortes:**

- Boa interface
- Visual WYSIWYG
- Facilidade de instalação (segui o tutorial e não houve qualquer incidente)
- Código aberto com licença Apache
- Tutoriais no sítio da ferramenta
- A desenvolvedora foi adquirida recentemente pela VMWare, o que pode garantir mais recursos para a evolução desta ferramenta

*** Pontos fracos**

- Compatibilidade boa com Postgresql, mas não excepcional. Recursos específicos como herança de tabelas e indexação avançada não são abordados na ferramenta e tem de ser codificados manualmente no banco.
- A desenvolvedora foi adquirida recentemente pela VMWare, e o impacto desta mudança no desenvolvimento da ferramenta não pode ser previsto de antemão

* Avaliação Pessoal

A primeira impressão que me causou foi bastante positiva, mas não recomendo a utilização em ambientes de produção sem vários testes com prototipação e simulações de carga.

Unlogged Tables: Funcionalidade para Aumento de Desempenho!

Todos sempre buscamos melhorar o desempenho das operações de banco de dados. E um dos recursos de performance ainda pouco utilizados da versão 9.1 do postgres são as chamadas *unlogged tables*.

O que são Unlogged Tables?

Unlogged Tables são tabelas que não apresentam suporte a recuperação pós-falha. Não apresentam portanto log de transações (*write-ahead-log* - WAL). Essa característica possibilita um grande ganho de desempenho em todas as operações realizadas. O ganho de desempenho obtido se deve ao sacrifício da possibilidade de recuperar os dados em caso de falha de sistema.

Uma *unlogged table* tem seus dados automaticamente perdidos em caso de falha, pois é truncada automaticamente, o que gera um ganho no tempo de recuperação do banco de dados.

Os dados de uma *unlogged table* não sofrem replicação dentro do postgresql.

Em *unlogged tables* não há necessidade de se manter o *log* e sincronizá-lo com o banco de dados, fator importante para o de ganho de desempenho.

Em que situações é recomendado utilizar este tipo de tabela?

Em situações em que a durabilidade dos dados não seja realmente importante:

- Para parâmetros de aplicações web;
- Cache de dados em geral;
- Tabelas de status de aplicações, entre outras possibilidades.

Acredito que apenas uma pequena parte de sistemas de bancos de dados possa ser armazenada em tabelas *unlogged*.

As operações de inserção, alteração, alteração e consulta a dados de uma "tabela sem log" são diferentes de uma tabela "normal"?

A forma de fazer e os comandos utilizados permanecem os mesmos. No entanto, internamente, não há *write-ahead-log* (WAL), o que faz com que os dados da tabela seja perdidos em caso de quedas de sistema. A velocidade das operações tende a ser bem maior.

Como criar *Unlogged Tables*?

A criação de tabelas sem *log* é bastante simples. Basta colocar a cláusula "UNLOGGED" no comando de criação da tabela.

```
teste=> CREATE TABLE LOGADA (cod integer, descricao varchar(50));
CREATE TABLE
teste=> CREATE UNLOGGED TABLE NAO_LOGADA (cod integer, descricao
varchar(50));
CREATE TABLE
teste=>
É permitido indexar este tipo de tabela?
```

Não existem restrições à indexação, exceto para índices GIST em que este recurso não está implementado. É possível inclusive reindexar, se for o caso! Os índices de uma *unlogged table* também são "unlogged", isto é, são truncados em caso de falha do sistema.

```
teste=> CREATE INDEX UNLOGT ON NAO_LOGADA(cod);
CREATE INDEX
teste=>
teste=> insert into NAO_LOGADA values (1, 'Teste 1');
INSERT 0 1
teste=> insert into NAO_LOGADA values (2, 'Teste 2');
INSERT 0 1
teste=> insert into NAO_LOGADA values (3, 'Teste 3');
INSERT 0 1
teste=> REINDEX TABLE NAO_LOGADA;
REINDEX
```

De quanto é o ganho esperado em desempenho?

DEPENDE da operações realizada. Veja o link abaixo e depois faça seus próprios testes:

<http://pgsnaga.blogspot.com/2011/10/pgbench-on-unlogged-tables.html>

Considerações Finais

Unlogged Tables são um recurso válido para ganho de performance em certos casos específicos. No entanto, a definição de que tabelas devem ser *unlogged* pode gerar erros graves e impossibilitar a recuperação de dados relevantes. Esta decisão deve ser sempre bastante embasada e levar em conta as necessidades de todos os usuários do banco.

Como Localizar e Deletar registros duplicados

1. Select para localizar duplicados

```
select campo,campo1,count(*)
from tabela having count(*) > 1
group by campo,campo1
```

2. Como deletar duplicados

```
delete from tab p1
where rowid < (select max(rowid)
                 from tab1 p2
                 where p1.primary_key = p2.primary_key);
```

Encontrar registros duplicados

Quando a chave é o emp_cnpj

```
SELECT
    emp_cnpj,
    count(*)
FROM empresa
WHERE
    emp_cnpj <> ""
GROUP BY emp_cnpj
HAVING COUNT(*) > 1
```

Se não importar qual das duplicatas irá permanecer você pode usar:

```
CREATE TABLE tab_temp AS SELECT DISTINCT * FROM sua_tabela;
DROP sua_tabela;
ALTER TABLE TAB_TEMP RENAME TO SUA_TABELA;
```

Se não for qualquer linha duplicata que possa ser excluída verifique se as cláusulas ORDER BY, GROUP BY e DISTINCT ON podem ajudar a especificar

qual das duplicatas deverá permanecer.

SELECT DISTINCT cep FROM cep_tabela

WHERE cep IN (SELECT cep FROM cep_tabela AS Tmp GROUP BY cep, tipo, logradouro, bairro, municipio, uf HAVING Count(*) > 1) ORDER BY cep;

(Adaptação de consulta gerada pelo assistente Encontrar duplicadas do Access).

Ou:

select count(*) as quantos, cep from cep_tabela group by cep having count(*) > 1;

REMOVER DUPLICADOS

Para tabelas criadas WITH OIDS:

DELETE FROM cep_tabela2 WHERE oid NOT IN

(SELECT min(oid) FROM cep_tabela2 GROUP BY cep, tipo, logradouro, bairro, municipio, uf);

Do exemplo 8.10 do manual em português do Brasil.

Ou:

Criando uma segunda tabela que conterá somente os registros exclusivos e ainda guarda uma cópia da tabela original:

CREATE TABLE cep_tabela2 AS SELECT cep, tipo, logradouro, bairro, municipio, uf FROM cep_tabela GROUP BY cep, tipo, logradouro, bairro, municipio, uf ORDER BY cep;

Caso não importe qual das duplicatas irá permanecer:

CREATE TABLE tab_temp AS SELECT DISTINCT * FROM tabela;

DROP tabela;

ALTER TABLE tab_temp RENAME TO tabela;

(Dica de Osvaldo Rosario Kussama na lista de PostgreSQL Brasil)

Registros duplicados (distinct)

Con la cláusula "distinct" se especifica que los registros con ciertos datos duplicados sean obviadas en el resultado. Por ejemplo, queremos conocer todos los autores de los cuales tenemos libros, si utilizamos esta sentencia:

```
select autor from libros;
```

Aparecen repetidos. Para obtener la lista de autores sin repetición usamos:

```
select distinct autor from libros;
```

También podemos tipar:

```
select autor from libros
group by autor;
```

Note que en los tres casos anteriores aparece "null" como un valor para "autor". Si sólo queremos la lista de autores conocidos, es decir, no queremos incluir "null" en la lista, podemos utilizar la sentencia siguiente:

```
select distinct autor from libros
where autor is not null;
```

Para contar los distintos autores, sin considerar el valor "null" usamos:

```
select count(distinct autor)
from libros;
```

Note que si contamos los autores sin "distinct", no incluirá los valores "null" pero si los repetidos:

```
select count(autor)
from libros;
```

Esta sentencia cuenta los registros que tienen autor.

Podemos combinarla con "where". Por ejemplo, queremos conocer los distintos autores de la editorial "Planeta":

```
select distinct autor from libros
where editorial='Planeta';
```

También puede utilizarse con "group by" para contar los diferentes autores por editorial:

```
select editorial, count(distinct autor)
from libros
group by editorial;
```

La cláusula "distinct" afecta a todos los campos presentados. Para mostrar los títulos y editoriales de los libros sin repetir títulos ni editoriales, usamos:

```
select distinct titulo,editorial
from libros
order by titulo;
```

Note que los registros no están duplicados, aparecen títulos iguales pero con editorial diferente, cada registro es diferente.

Entonces, "distinct" elimina registros duplicados.

```
--  
SELECT pp, COUNT (pp) FROM table GROUP BY pp ORDER BY COUNT DESC;
```

Encontrar registros duplicados

Quando a chave é o emp_cnpj

```
SELECT
    emp_cnpj,
    count(*)
FROM empresa
WHERE
    emp_cnpj <> ""
GROUP BY emp_cnpj
HAVING COUNT(*) > 1
```

Se não importar qual das duplicatas irá permanecer você pode usar:

```
CREATE TABLE tab_temp AS SELECT DISTINCT * FROM sua_tabela;
DROP sua_tabela;
ALTER TABLE TAB_TEMP RENAME TO SUA_TABELA;
```

Se não for qualquer linha duplicata que possa ser excluída verifique se as cláusulas ORDER BY, GROUP BY e DISTINCT ON podem ajudar a especificar qual das duplicatas deverá permanecer.

```
SELECT DISTINCT cep FROM cep_tabela
```

```
WHERE cep IN (SELECT cep FROM cep_tabela AS Tmp GROUP BY cep,tipo,logradouro,
bairro, municipio,uf HAVING Count(*) >1 ) ORDER BY cep;
```

(Adaptação de consulta gerada pelo assistente Encontrar duplicadas do Access).

Ou:

```
select count(*) as quantos, cep from cep_tabela group by cep having count(*) > 1;
```

REMOVER DUPLICADOS

Para tabelas criadas WITH OIDS:

```
DELETE FROM cep_tabela2 WHERE oid NOT IN
```

```
(SELECT min(oid) FROM cep_tabela2 GROUP BY cep, tipo, logradouro, bairro, municipio, uf);
```

Do exemplo 8.10 do manual em português do Brasil.

Ou:

Criando uma segunda tabela que conterá somente os registros exclusivos e ainda guarda uma cópia da tabela original:

```
CREATE TABLE cep_tabela2 AS SELECT cep, tipo, logradouro, bairro, municipio, uf
FROM cep_tabela GROUP BY cep, tipo, logradouro, bairro, municipio, uf ORDER BY cep;
```

Caso não importe qual das duplicatas irá permanecer:

```
CREATE TABLE tab_temp AS SELECT DISTINCT * FROM tabela;
```

```
DROP tabela;
```

```
ALTER TABLE tab_temp RENAME TO tabela;
```

(Dica de Osvaldo Rosario Kussama na lista de PostgreSQL Brasil)

Registros duplicados (distinct)

Con la cláusula "distinct" se especifica que los registros con ciertos datos duplicados sean obviadas en el resultado. Por ejemplo, queremos conocer todos los autores de los cuales tenemos libros, si utilizamos esta sentencia:

```
select autor from libros;
```

Aparecen repetidos. Para obtener la lista de autores sin repetición usamos:

```
select distinct autor from libros;
```

También podemos tipar:

```
select autor from libros
group by autor;
```

Note que en los tres casos anteriores aparece "null" como un valor para "autor". Si sólo queremos la lista de autores conocidos, es decir, no queremos incluir "null" en la lista, podemos utilizar la sentencia siguiente:

```
select distinct autor from libros
where autor is not null;
```

Para contar los distintos autores, sin considerar el valor "null" usamos:

```
select count(distinct autor)
from libros;
```

Note que si contamos los autores sin "distinct", no incluirá los valores "null" pero si los repetidos:

```
select count(autor)
from libros;
```

Esta sentencia cuenta los registros que tienen autor.

Podemos combinarla con "where". Por ejemplo, queremos conocer los distintos autores de la editorial "Planeta":

```
select distinct autor from libros
where editorial='Planeta';
```

También puede utilizarse con "group by" para contar los diferentes autores por editorial:

```
select editorial, count(distinct autor)
from libros
group by editorial;
```

La cláusula "distinct" afecta a todos los campos presentados. Para mostrar los títulos y editoriales de los libros sin repetir títulos ni editoriales, usamos:

```
select distinct titulo,editorial
from libros
order by titulo;
```

Note que los registros no están duplicados, aparecen títulos iguales pero con editorial diferente, cada registro es diferente.

Entonces, "distinct" elimina registros duplicados.

--
 SELECT pp, COUNT (pp) FROM table GROUP BY pp ORDER BY COUNT DESC;

25 - PostgreSQL Cheat Sheet

CREATE DATABASE

```
CREATE DATABASE dbName;
```

CREATE TABLE (with auto numbering integer id)

```
CREATE TABLE tableName (
  id serial PRIMARY KEY,
  name varchar(50) UNIQUE NOT NULL,
  dateCreated timestamp DEFAULT current_timestamp
);
```

Add a primary key

```
ALTER TABLE tableName ADD PRIMARY KEY (id);
```

Create an INDEX

```
CREATE UNIQUE INDEX indexName ON tableName (columnNames);
```

Backup a database (command line)

```
pg_dump dbName > dbName.sql
```

Backup all databases (command line)

```
pg_dumpall > pgbackup.sql
```

Run a SQL script (command line)

```
psql -f script.sql databaseName
```

Search using a regular expression

```
SELECT column FROM table WHERE column ~ 'foo.*';
```

The first N records

```
SELECT columns FROM table LIMIT 10;
```

Pagination

```
SELECT cols FROM table LIMIT 10 OFFSET 30;
```

26 - Prepared Statements

```
PREPARE preparedInsert (int, varchar) AS
```

```
INSERT INTO tableName (intColumn, charColumn) VALUES ($1, $2);
```

```
EXECUTE preparedInsert (1,'a');
```

```
EXECUTE preparedInsert (2,'b');
```

```
DEALLOCATE preparedInsert;
```

Create a Function

```
CREATE OR REPLACE FUNCTION month (timestamp) RETURNS integer
AS 'SELECT date_part("month", $1)::integer;'
LANGUAGE 'sql';
```

Table Maintenance

```
VACUUM ANALYZE table;
```

Reindex a database, table or index

```
REINDEX DATABASE dbName;
```

Show query plan

```
EXPLAIN SELECT * FROM table;
```

Import from a file

```
COPY destTable FROM '/tmp/somefile';
```

Show all runtime parameters

```
SHOW ALL;
```

Grant all permissions to a user

```
GRANT ALL PRIVILEGES ON table TO username;
```

Perform a transaction

```
BEGIN TRANSACTION
```

```
UPDATE accounts SET balance += 50 WHERE id = 1;
```

```
COMMIT;
```

Basic SQL

Get all columns and rows from a table

```
SELECT * FROM table;
```

Add a new row

```
INSERT INTO table (column1,column2)  
VALUES (1, 'one');
```

Update a row

```
UPDATE table SET foo = 'bar' WHERE id = 1;
```

Delete a row

```
DELETE FROM table WHERE id = 1;
```

*Prints in under two pages. Questions, comments, criticism, or requests can be directed
[Here](#)*

[More Cheat Sheets Here.](#)

Copyright © 2005 [Foundeo Inc.](http://foundeo.com/) (<http://foundeo.com/>) / [Peter Freitag](http://www.petefreitag.com/) (<http://www.petefreitag.com/>), All Rights Reserved.

This document may be printed freely as long as this notice stays intact.

27 - Exercícios

Entendendo e Trabalhando com Clusters

Criando Novos Clusters no PostgreSQL para Windows

Para criar um novo cluster

Crie um diretório para abrigar o novo cluster (lembre que o usuário postgres deve ter permissão de escrita nele).

Ex.: data2 no diretório bin.

Criar o novo cluster:

- C:\Program Files\PostgreSQL\8.3\bin>initdb -U postgres -D data2

Editar o data2\postgresql.conf e alterar a porta para 5444

Iniciar o servidor do novo cluster

- pg_ctl -D data2 start

Acessar a console do novo cluster

- psql -p 5444 -U postgres

Listar os bancos

- \l -- Observe que somente existem os bancos de templates. Temos um novo cluster.

Obs.: No Windows não consegui dar suporte a latin1 em novos clusters nem o original suporta.

Isso só foi conseguido em novos clusters no Linux.

Tecle Ctrl+Alt+Del no Windows ou 'ps ax|grep post' no Linux e veja que agora temos dois postmaster ou pg_ctl na memória.

Obs.: O utilitário 'cpau' de <http://www.joeware.net/freetools/tools/cpau/index.htm>

Lembre que o usuário do postgresql não têm privilégios de login no Windows.

Com este utilitário podemos estar "logado" no Windows como usuário do PostgreSQL.

Criação de Novos Clusters no PostgreSQL 8.3 for Linux (Ubuntu 7.10):

Criando os clusters

cluster em latin1

Criação do diretório para o cluster, data_latin1, tornando o usuário postgres seu dono:
mkdir data_latin1

su - postgres

export LANG=pt_BR.ISO-8859-1

bin/initdb --encoding latin1 -D data_latin1

Editar o script data_latin1/postgresql.conf e alterar a porta para 5433

Conectando no cluster em latin1

bin/pg_ctl -D data_latin1 start

bin/psql -U postgres postgres -p 5433

cluster em utf-8

su - postgres

bin/initdb -D data_utf8

Como utf8 o default no Ubuntu, não preciso passar parâmetro.

Editar o script data_utf8/postgresql.conf e alterar a porta para 5434

Conectando no cluster utf-8

```
bin/pg_ctl -D data_utf8 start
bin/psql -U postgres postgres -p 5434
```

Esquemas

- Esquemas ajudam a organizar os bancos e também economizam conexões.
- Para ver os esquemas em funcionamento iremos criar um banco contendo dois esquemas: pessoal e comercial
- O esquema pessoal conterá as tabelas: funcionários e departamentos
- O esquema comercial conterá as tabelas: clientes, produtos e pedidos

```
create database dba_projeto3;
alter database dba_projeto3 rename to dba_projeto2;
\c dba_projeto2
```

\dn -- Observe os esquemas

```
create schema pessoal authorization dba1;
```

```
create schema comercial;
alter schema comercial owner to dba1;
```

\dn -- Observe os esquemas

\dn+

-- Vamos aproveitar as tabelas do banco dba_projeto e importar para o esquema comercial.

-- Lembrando que para importar no psql temos que, antes, colocar o esquema comercial no path com:

```
set search_path to public, comercial;
```

-- Após importar todas as tabelas, realizar as alterações e importar os registros, execute:

```
\c dba_projeto2 dba1
```

\d -- Observe que não aparecerá nenhuma tabela. Então execute:

```
set search_path to comercial;
```

-- Então execute novamente

```
\d -- Agora aparecerão as tabelas do esquema comercial
```

```
select count(*) from clientes; -- Não terá permissão
```

```
\c dba_projeto2 postgres
```

```
set search_path to comercial;
```

```
alter table clientes owner to dba1;
alter table produtos owner to dba1;
alter table pedidos owner to dba1;
```

```
\c dba_projeto2 dba1
```

```
set search_path to comercial;
```

```
select count(*) from clientes;
select count(*) from produtos;
select count(*) from pedidos;
```

-- Agora realize uma consulta na tabela clientes, mas sem o esquema comercial estar no path:

```
reset search_path;
```

```
select * from clientes; -- Tabela não existe. Isso porque estamos no esquema public
```

```
select * from comercial.clientes; -- Agora sim, passando o nome do esquema e ponto antes do nome da tabela
```

Agora criar as duas tabelas do esquema pessoal e incluir alguns registros

```
set search_path to pessoal;
```

```
create table empregados(
    cpf char(11) primary key,
    nome char(45) not null,
    depto int,
    telefone char(10),
    celular char(10),
    email char(45),
    data_nasc date
);
```

```
create table deptos(
    codigo int primary key,
    descricao char(50)
);
```

```
insert into deptos (codigo, descricao) values (1, 'Pessoal');
insert into deptos (codigo, descricao) values (2, 'Administração');
insert into deptos (codigo, descricao) values (3, 'Financeiro');
insert into deptos (codigo, descricao) values (4, 'Técnico');
insert into deptos (codigo, descricao) values (5, 'TI');
```

```
insert into empregados (cpf, nome, depto, telefone, celular, email, data_nasc)
values
('111111111110', 'Antônio Brito Cunha', 1,'2343-3454', '9933-3450', 'joao@joao.com.br',
'25/05/1975'),
('111111111111', 'Roberto Brito Cunha', 1,'2343-3453', '9933-3451', 'roberto@joao.com.br',
'25/05/1976'),
('111111111112', 'Pedro Brito Cunha', 2,'2343-3452', '9933-3452', 'pedro@joao.com.br',
'25/05/1977'),
('111111111113', 'Airton Brito Cunha', 2,'2343-3451', '9933-3453', 'airton@joao.com.br',
'25/05/1978'),
('111111111114', 'Francisco Brito Cunha',3, '2343-3450', '9933-3454',
'francisco@joao.com.br', '25/05/1979'),
('111111111115', 'Paulo Brito Cunha', 4,'2343-3459', '9933-3455', 'paulo@joao.com.br',
'25/05/1970'),
('111111111116', 'Neide Brito Cunha', 4,'2343-3458', '9933-3456', 'neide@joao.com.br',
'25/05/1971'),
('111111111117', 'Ivete Brito Cunha', 5,'2343-3457', '9933-3457', 'ivete@joao.com.br',
'25/05/1972'),
('111111111118', 'Maria Brito Cunha', 5,'2343-3456', '9933-3458', 'maria@joao.com.br',
'25/05/1973'),
('111111111119', 'Rogério Brito Cunha', 5,'2343-3455', '9933-3459', 'rogerio@joao.com.br',
'25/05/1974');
```

Otimizador de Consultas

Fazer o download do banco de dados de teste pagina de:

<http://pgfoundry.org/frs/download.php/1719/pagila-0.10.1.zip>

Ou de:

<http://www.postgresql.org/ftp/projects/pgFoundry/dbsamples/>

Vamos criar um banco chamado pagila e outro pagila2

No pagina rodaremos o ANYLYZE e no pagina2 não rodaremos.

Faça o download, descompacte e leia o README para algumas informações.

Agora vamos criar os dois bancos:

```
c:\Arquivos de Programas\PostgreSQL\8.2\bin>createdb -U postgres pagila
```

```
c:\Arquivos de Programas\PostgreSQL\8.2\bin>createdb -U postgres pagila2
```

Vamos importar a estrutura do banco pagila, que está no script "pagila-schema.sql":

```
c:\Arquivos de Programas\PostgreSQL\8.2\bin>psql -U postgres pagila < c:\pagila-schema.sql
```

Agora importando os dados:

```
c:\Arquivos de Programas\PostgreSQL\8.2\bin>psql -U postgres pagila < c:\pagila-data.sql
```

Vamos fazer o mesmo com o banco pagila2:

```
c:\Arquivos de Programas\PostgreSQL\8.2\bin>psql -U postgres pagila2 < c:\pagila-schema.sql
```

```
c:\Arquivos de Programas\PostgreSQL\8.2\bin>psql -U postgres pagila2 < c:\pagila-data.sql
```

Agora vamos rodar o vacuum com analyze no banco pagila:

```
c:\Arquivos de Programas\PostgreSQL\8.2\bin>vacuumdb -z -U postgres pagila
```

Iremos abrir duas seções da console com o psql. Numa com o pagila e na outra com o pagila2, para ficar mais prático. Deixe uma janela ao lado da outra.

Vamos acessar o banco pagila na primeira console:

```
c:\Arquivos de Programas\PostgreSQL\8.2\bin>psql -U postgres pagila
```

Vamos acessar o banco pagila2 na segunda console:

```
c:\Arquivos de Programas\PostgreSQL\8.2\bin>psql -U postgres pagila2
```

Agora faça os testes sugeridos na aula 16.

ALERTA: o material da aula mostra exemplos usando uma versão anterior do pagila e do PostgreSQL, portanto alguns ajustes serão necessários em alguns pontos.

Um deles é nesta consulta:

```
select
    film.title AS title,
    array_to_string(array_accum(actor.first_name || ' ' || actor.last_name), ',') AS actors
from
    film
    inner join film_actor on film.film_id = film_actor.film_id
    inner join actor on film_actor.actor_id = actor.actor_id
GROUP BY film.title
ORDER BY film.title
```

Veja que tanto nomes de tabelas quanto campos foram alterados.

Para fazer essa consulta funcionar fiz as adaptações abaixo:

```
select
    film_actor.film_id      AS title,
    (actor.first_name || ' ' || actor.last_name) AS actors
from
    film_actor
    inner join actor on film_actor.actor_id = actor.actor_id
GROUP BY film_actor.film_id, actor.first_name, actor.last_name
ORDER BY film_actor.film_id;
```

Restaurando um Template1 Corrompido usando o Template0

Por padrão não podemos acessar o template0

```
\c template0
FATAL: Database "template0" is not currently accepting connections
Previous connection kept
```

O segredo está na tabela de sistema pg_database. Esta tabela existe em todos os bancos e guarda algumas propriedades de cada um deles. Duas destas propriedades são mais interessantes agora:

datistemplate - que diz quem é nosso banco de template e datallowconn que diz para qual banco usuários registrados podem conectar.

```
select * from pg_database;
```

Mostrará o status destas duas colunas e o que precisamos fazer para alterar.

Para conectar ao template0 precisamos alterar o flag:

```
UPDATE pg_database SET datallowconn = TRUE WHERE datname = 'template0';
```

Agora podemos conectar ao template0:

```
\c template0
```

Para poder apagar o template1 e recriar como cópia do template0:

```
UPDATE pg_database SET datistemplate = FALSE WHERE datname = 'template1';
```

```
template0=# drop database template1;
```

```
template0=# create database template1 with template = template0;
```

Restaurando as propriedades originais:

```
template0=# UPDATE pg_database SET datistemplate = TRUE
template0-# WHERE datname = 'template1';
```

```
template0=# \c template1
```

```
template1=# UPDATE pg_database SET datallowconn = FALSE
template1-# WHERE datname = 'template0';
```

Para garantir a maior eficiência possível:

```
VACUUM FULL FREEZE;
```

Fonte: http://wiki.postgresql.org/wiki/Adventures_in_PostgreSQL,_Episode_1

28 - Exemplo de Modelagem de Banco de Dados

Controle de Estoque Simples

```
create table clientes
(
    cliente int primary key,
    cpf char(11),
    nome char(45) not null,
    credito_liberado char(1) not null,
    data_nasc date,
    email varchar(50)
);

create table funcionarios
(
    funcionario int primary key,
    cpf char(11),
    nome char(45) not null,
    senha char(32) not null,
    email varchar(50),
    data_nasc date not null
);

create table produtos
(
    produto int primary key,
    descricao varchar(100) not null,
    unidade char(4) not null,
    data_cadastro timestamp not null
);

create table pedidos
(
    pedido int primary key,
    cliente int not null,
    funcionario int not null,
    data_pedido date not null,
    data_confirmacao date not null,
    FOREIGN KEY (cliente) REFERENCES clientes(cliente) ON DELETE RESTRICT,
    FOREIGN KEY (funcionario) REFERENCES funcionarios(funcionario) ON DELETE RESTRICT
);

create table pedido_itens
(
    item int primary key,
    estoque int not null,
    quantidade int not null,
    preco_venda numeric(12,2) not null,
    pedido int not null,
```

```
FOREIGN KEY (pedido) REFERENCES pedidos(pedido) ON DELETE RESTRICT
);
```

29 - Normalização de uma tabela de CEP sem mesmo chave primária.

Como é uma tabelona contendo diversos campos que deveriam vir de outras tabelas relacionadas, então iremos fazer este trabalho de normalizar isolando as informações:

Tenho um arquivo CSV de ceps do tempo que os Correios distribuíam gratuitamente em seu site, contendo 633.401 registros.

Agora vou usá-lo como exercício de normalização e tentar reaproveitar seus dados.

Esta tabela, ou melhor, após a normalização, serão algumas tabelas que poderão ser utilizadas num cadastro de pessoas.

su - postgres

psql

create database cep encoding 'latin1';

Latin1 é para compatibilizar com conteúdo da tabela ceps.

Pois o recomendado atualmente é a codificação UNICODE. Em um banco com codificação

latin1 (iso-8859-1) tente representar por exemplo, o símbolo do euro (€).

Não consegue, pois é outra codificação, portanto sempre que possível devemos usar UTF-8.

Para comprovar crie essa tabela, num banco em latin1:

create table codificacao(c char(1))

Tente inserir este registro:

insert into codificacao values ('€')

E receberá a mensagem:

ERRO: caractere 0xe282ac da codificação "UTF8" não tem equivalente em "LATIN1"

Tabela de CEPs original:

```
create table ceps
(
    cep char(8),
    tipo char(72),
    logradouro char(70),
    bairro char(72),
    municipio char(60),
    uf char(2)
```

);

Importar dados (script em: <http://pg.ribafs.net/down/scripts//cep.sql.zip>)
\copy ceps from /home/ribafs/cep_brasil.csv

Adicionar PK

```
alter table ceps add constraint cep_pk primary key(cep);
```

Tabela municipios

```
create sequence municipio_seq;
create table municipios as select distinct(municipio), uf from ceps order by uf;
alter table municipios rename column municipio to descricao;
alter table municipios add column municipio int;
update municipios set municipio=nextval('municipio_seq');
alter table municipios add constraint municipio_pk primary key(municipio);
alter table municipios add constraint municipio_unk unique(descricao);

municipios(descricao, uf, municipio)
```

Tabela bairros

```
create sequence bairro_seq;
create table bairros as select distinct(bairro) from ceps order by bairro;
alter table bairros rename column bairro to descricao;
alter table bairros add column bairro int;
update bairros set bairro=nextval('bairro_seq');
alter table bairros add constraint bairro_pk primary key(bairro);
alter table bairros add constraint bairro_unk unique(descricao);

bairro(descricao, bairro)
```

Tabela logradouros

```
create sequence logradouro_seq;
create table logradouros as select distinct(logradouro) from ceps order by logradouro;
alter table logradouros rename column logradouro to descricao;
alter table logradouros add column logradouro int;
update logradouros set logradouro=nextval('logradouro_seq');
alter table logradouros add constraint logradouro_pk primary key(logradouro);
alter table logradouros add constraint logradouro_unk unique(descricao);

logradouro(descricao, logradouro)
```

Tabela tipos

```

create sequence tipo_seq;
create table tipos as select distinct(tipo) from ceps order by tipo;
alter table tipos rename column tipo to descricao;
alter table tipos add column tipo int;
update tipos set tipo=nextval('tipo_seq');
alter table tipos add constraint tipo_pk primary key(tipo);
alter table tipos add constraint tipo_unk unique(descricao);

tipos(descricao, tipo)

```

Tabela ceps normalizada

```

create table cepsn
(
    cep char(8) not null,
    tipo int,
    logradouro int,
    bairro int,
    municipio int,
    primary key(cep, logradouro),
    constraint tipo_fk foreign key (tipo) references tipos(tipo),
    constraint logradouro_fk foreign key (logradouro) references logradouros(logradouro),
    constraint bairro_fk foreign key (bairro) references bairros(bairro),
    constraint municipio_fk foreign key (municipio) references municipios(municipio)
);

```

-- Como atualmente podem existir mais de um CEP por logradouro, então CEP não pode ser a PK,
-- portanto teremos uma chave natural formada pelo CEP e pelo logradouro

Criar assim:

```

create table cepsn as select distinct(cep) from ceps
alter table cepsn add column tipo int;
alter table cepsn add column logradouro int;
alter table cepsn add column bairro int;
alter table cepsn add column municipio int;
alter table cepsn add constraint tipo_fk foreign key (tipo) references tipos(tipo);
alter table cepsn add constraint logradouro_fk foreign key (logradouro) references logradouros(logradouro);
alter table cepsn add constraint bairro_fk foreign key (bairro) references bairros(bairro);
alter table cepsn add constraint municipio_fk foreign key (municipio) references municipios(municipio);

```

Agora vamos popular a tabela cepsn com os registros da tabela ceps.
Veja que não é somente importar todos os tipos da tabela tipos para o campo tipo de cepsn.

Temos que trazer os tipos corretos de todos os 644 mil registros. Cada um com seu tipo correspondente.

Postanto não será uma tarefa simples nem direta. Exigirá um pouco de conhecimento da linguagem SQL.

Quando não temos certeza se a nossa consulta é coerente e que poderá demorar muito, então ajuda muito consultar o PostgreSQL como ele faria essa consulta.

Execute a consulta com o EXPLAIN que ele vai dar uma dica, em especial os valores do custo final e rows.

CEPs - 633401

Tipos - 189

Logradouros - 316499

Bairros - 16766

Municípios - 346

Atualizar tipo em cepsn com o valor tipo de tipos, mas correspondentes aos de ceps

Agora para receber os valores do campo tipo tenho que pensar assim:

tomar de cepsn o cep e testar se igual ao cep de ceps

Ainda pegar neste cep o valor do tipo em ceps e testar se é igual ao valor da descrição em tipos.

Então trazer o resultado. A consulta abaixo faz isso:

```
update cepsn cn set tipo = (select t.tipo from ceps c,tipos t where cn.cep = c.cep and c.tipo = t.descricao);
```

```
update cepsn cn set logradouro = (select l.logradouro from ceps c,logradouros l where cn.cep = c.cep and c.logradouro = l.descricao);
```

Esta demora exageradamente.

Usei apenas:

```
update cepsn cn set logradouro = (select l.logradouro from ceps c,logradouros l where cn.cep = c.cep
```

```
and c.logradouro = l.descricao and l.logradouro >= 305372 and l.logradouro <=305375);
```

Custo total: 10.609.291,16 Tempo: 41.419 ms

```
update cepsn cn set bairro = (select b.bairro from ceps c,bairros b where cn.cep = c.cep and c.bairro = b.descricao);
```

Usei somente:

```
update cepsn cn set bairro = (select b.bairro from ceps c,bairros b where cn.cep = c.cep and c.bairro = b.descricao
and b.bairro < 9110 and b.bairro >9101);
```

```
update cepsn cn set municipio = (select m.municipio from ceps c,municipios m where cn.cep = c.cep and c.municipio = m.descricao);
```

Usei somente:

```
update cepsn cn set municipio = (select m.municipio from ceps c,municipios m where
cn.cep = c.cep
and c.municipio = m.descricao and m.uf='CE');
```

Obs.: só para ter uma idéia, o tempo desta consulta foi de 133.330ms e o curto total acusado pelo Explain era de 14.417.399.32 e rows 633401 (total)

Agora, finalmente uma consulta de CEP na tabela normalizada

```
select cep, t.descricao, l.descricao, b.descricao, m.descricao, m.uf from cepsn c, tipos t,
logradouros l, bairros b, municipios m where c.cep='60420440' and t.tipo=c.tipo and
l.logradouro=c.logradouro
and b.bairro=c.bairro and m.municipio=c.municipio;
```

Alterei a tabela cepsn adicionando índice único nos campos: tipo, bairro, logradouro e municipio:

```
CREATE UNIQUE INDEX unq_tipo ON pepsn (tipo);
```

Consultas úteis:

```
select * from municipios group by uf,municipio,descricao having count(municipio) = 1 order
by uf,descricao;
select count(municipio) from municipios group by uf having count(municipio)>1 and
uf='CE';
```

Cuidado com operadores boolean:

```
select * from municipios where uf='SP' and uf='DF' order by descricao; -- Nada retorna
select * from municipios where uf='SP' or uf='DF' order by descricao; -- esta retorna
```

Modelagem do Banco de Dados Pessoa

Neste banco serão utilizados vários recursos importantes

```
/*
Banco: pessoa
```

Cenário:

Modelagem de pessoas (clientes, funcionários, fornecedores, etc) para empresas, órgãos públicos ou outras instituições brasileiras.

Usando o Modelo Relacional e Normalizado (no SGBD PostgreSQL).
*/

-- Função de validação do 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)
-- ****
-- 
/*
-- Número com dígitos verificadores e sem nenhuma máscara (bpchar)
```

```
-- Válidos
SELECT f_cnpjcpf( 0, '46376021000107' );-- CNPJ
SELECT f_cnpjcpf( 1, '48533316461' );      -- CPF
SELECT f_cnpjcpf( 2, 'isento' );        -- Livre
```

```
-- Inválidos
SELECT f_cnpjcpf( 0, '46376044444107' );-- CNPJ
SELECT f_cnpjcpf( 1, '40003316461' );      -- CPF
SELECT f_cnpjcpf( 2, 'isento' );        -- Livre
```

Retornos possíveis:

```
-- -1: Tipo de Documento invalido.
-- -2: Caracter inválido no numero do documento.
-- -3: Numero do Documento invalido.
-- 1: Documento Validado Corretamente - OK (smallint)
*/
```

create language plpgsql;

```
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;
    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';

```

-- Esta função acima é do Juliano Ignácio num dos seus artigos do iMasters:
-- http://imasters.uol.com.br/artigo/1308/stored_procedures_triggers_functions

-- Validação de inscrição estadual do Ceará, tendo como fonte o algoritmo em:
http://www.sintegra.gov.br/Cad_Estados/cad_CE.html
-- No Ceará a IE tem 8 dígitos válidos mais o dígito verificador

create or replace function f_ie_ce(ie_ce text) returns text as

```

$$
declare
    n1 integer;
    n2 integer;
    n3 integer;
    n4 integer;
    n5 integer;
    n6 integer;
    n7 integer;
    n8 integer;
    n9 integer;
    nt integer;
    s integer;
    m integer;
    dv integer;
begin
    n1 = substring(ie_ce from 1 for 1)::integer;
    n2 = substring(ie_ce from 2 for 1)::integer;
    n3 = substring(ie_ce from 3 for 1)::integer;
    n4 = substring(ie_ce from 4 for 1)::integer;
    n5 = substring(ie_ce from 5 for 1)::integer;
    n6 = substring(ie_ce from 6 for 1)::integer;
    n7 = substring(ie_ce from 7 for 1)::integer;
    n8 = substring(ie_ce from 8 for 1)::integer;
    n9 = substring(ie_ce from 9 for 1)::integer;
    s = 9*n1 + 8*n2 + 7*n3 + 6*n4 + 5*n5 + 4*n6 + 3*n7 + 2*n8;
    m = s%11;
    dv = 11 - m;
    -- raise exception 'm vale(%), dv vale(%), n6(%)',m,dv, n6;
    if (dv>=10) then
        dv = 0;
    end if;
    if (dv = n9) then
        return 1;
    else
        return 0;
    end if;
end;
$$
language 'plpgsql';

-- select ie_ce('060000014')

-- Função para gerar uma inscrição estadual (do Ceará) válida para testes

```

```

create or replace function ie_ce_gerador() returns text as
$$
declare
    ie_ce text;
    n1 integer;

```

```

n2 integer;
n3 integer;
n4 integer;
n5 integer;
n6 integer;
n7 integer;
n8 integer;
n9 integer;
nt integer;
s integer;
m integer;
dv integer;
d boolean;
begin
  d = FALSE;
  while (d = FALSE) loop
    ie_ce = floor(random()*1000000000+1)::text;

    -- if (char_length(ie_ce)<9) then continue; end if;

    n1 = substring(ie_ce from 1 for 1)::integer;
    n2 = substring(ie_ce from 2 for 1)::integer;
    n3 = substring(ie_ce from 3 for 1)::integer;
    n4 = substring(ie_ce from 4 for 1)::integer;
    n5 = substring(ie_ce from 5 for 1)::integer;
    n6 = substring(ie_ce from 6 for 1)::integer;
    n7 = substring(ie_ce from 7 for 1)::integer;
    n8 = substring(ie_ce from 8 for 1)::integer;
    n9 = substring(ie_ce from 9 for 1)::integer;

    s = 9*n1 + 8*n2 + 7*n3 + 6*n4 + 5*n5 + 4*n6 + 3*n7 + 2*n8;
    m = s%11;
    dv = 11 - m;
    -- raise exception 'm vale(%), dv vale(%), n6(%)',m,dv, n6;

    if (dv>=10) then
      dv = 0;
    end if;
    if (dv = n9) then
      d = TRUE;
      return ie_ce;
      exit;
    else
      d = FALSE;
      continue;
    end if;
    if (d = TRUE) then exit; end if;
  end loop;
return 0;
end;

```

```
$$
language 'plpgsql';
-- testar com select ie_ce_gerador()
```

-- Criação de domínios para melhorar as restrições dos tipos e agilizar criação de novas tabelas

```
CREATE DOMAIN dom_cnpj AS text
CONSTRAINT chk_cnpj CHECK (f_cnpjcpf(0, VALUE) = 1) NOT NULL;
```

```
CREATE DOMAIN dom_cpf AS text
CONSTRAINT chk_cpf CHECK (f_cnpjcpf(1, VALUE)=1 OR VALUE ~ '^informal$') NOT NULL;
-- Aceitar 'informal' ou 11 dígitos do CPF
```

```
CREATE DOMAIN dom_ie_ce AS text -- inscrição estadual para o Ceará, pois muda de um estado para outro
CONSTRAINT chk_ie_ce CHECK (f_ie_ce(VALUE)=1 OR VALUE ~ '^isento$') NOT NULL;
```

```
CREATE DOMAIN dom_cep AS text
CONSTRAINT chk_cep CHECK (VALUE ~ '^[0-9]{8}$') NOT NULL;
-- Os abaixo devem permitir nulo
```

```
CREATE DOMAIN dom_email AS text
CONSTRAINT chk_email CHECK (VALUE ~ '^[a-zA-Z][[:alnum:]._-]*@[a-zA-Z][[:alnum:]._-]*[.][a-zA-Z]+$');
```

```
CREATE DOMAIN dom_url AS text
CONSTRAINT chk_url CHECK (VALUE ~ '^((https|http):\/\/)?(([a-z][a-z0-9_-]*\.)+)(aero|arpa|biz|com|coop|edu|gov|info|int|jobs|mil|museum|name|nato|net|org|pro|travel|br|[a-z]{2})(/[a-z0-9_-~]+)*([/[a-z0-9_-\.]*(\?|[a-z0-9+_~-\.V%=&]*))?)$');
```

```
CREATE DOMAIN dom_telefone AS text
CONSTRAINT chk_telefone CHECK (VALUE ~ '^[3-9]{1}[0-9]{7}$'); -- Válidos somente iniciados com 3 e superiores
```

```
create table tipos
(
    tipo int primary key,
    descricao varchar(50) not null
);
```

```
-- Tabelas
create table logradouros
(
    logradouro int primary key,
    descricao varchar(50) not null
```

```
);

create table bairros
(
    bairro int primary key,
    descricao varchar(50) not null
);

create table ufs
(
    uf int primary key,
    descricao varchar(2) not null
);

create table municipios
(
    municipio varchar(50) primary key,
    uf int not null,
    constraint uf_fk foreign key (uf) references ufs(uf)
);

create table ceps
(
    cep dom_cep,
    tipo int,
    logradouro int,
    bairro int,
    municipio int,
    primary key(cep, logradouro),
    constraint tipo_fk foreign key (tipo) references tipos(tipo),
    constraint logradouro_fk foreign key (logradouro) references logradouros(logradouro),
    constraint bairro_fk foreign key (bairro) references bairros(bairro),
    constraint municipio_fk foreign key (municipio) references municipios(municipio)
);

create table enderecos
(
    cep int not null,
    logradouro int not null,
    numero varchar(8) not null,
    primary key(cep, numero),
    constraint cep_fk foreign key (cep,logradouro) references ceps(cep,logradouro)
);

-- Lembrando que aqui ainda faltam alguns atributos para o endereço: bloco, andar, apartamento e talvez ainda outros
```

```

create table telefones
(
    telefone int not null primary key,
    ddd varchar(4) not null,
    numero dom_telefone
);

create table fisicas
(
    fisica int primary key,
    cpf dom_cpf
);

create table juridicas
(
    cnpj dom_cnpj primary key,
    inscricao_estadual dom_ie_ce,
    site dom_url
);

create table pessoas
(
    pessoa int not null primary key,
    nome varchar(45) not null,
    cep int not null,
    tipo int not null,
    numero varchar(8) null,
    telefone int, -- Permitindo NULL, para o caso de alguém não ter telefone
    email dom_email,
    constraint telefone_fk foreign key (telefone) references telefones(telefone),
    constraint endereco_fk foreign key (cep,numero) references enderecos(cep,numero),
    constraint fisica_fk foreign key(tipo) references fisicas(fisica),
    constraint juridica_fk foreign key(tipo) references juridicas(cnpj)
);

-- Criação de índices parciais, que permitirão a criação de campos com CPF únicos, mas
-- somente para os que existirem

create unique index idx_cpf on fisicas (cpf)
    WHERE NOT (cpf = 'informal');

create unique index idx_ie on juridicas (inscricao_estadual)
    WHERE NOT (inscricao_estadual = 'isento');

-- INSERINDO ALGUNS REGISTROS PARA TESTE

insert into fisicas values(1, '22366437803');
insert into fisicas values(2, '47720595203');
insert into fisicas values(3, '33557245640');
insert into fisicas values(4, '56484636427');

```

```
--insert into fisicas values(5, '56484636426'); -- inválido
insert into fisicas values(6, 'informal');
insert into fisicas values(7, '90807363685');
--insert into fisicas values(8, 'informal_erro');
insert into fisicas values(8, 'informal'); -- aceita repetir informal
insert into fisicas values(9, 'informal');
insert into fisicas values(10, 'informal');
```

30 - Segurança

Segurança de Dados

Incêndio

- edifício que abriga o computador foi construído com material retardante e resistente ao fogo?
- Materiais combustíveis, como papéis e outros suprimentos, são armazenados fora da sala do computador?
- As fitas e os discos são armazenados fora da sala do computador?
- Existem cofres a prova de fogo para armazenar arquivos de segurança?
- Exercícios contra incêndios são realizados periodicamente?
- Existem empregados treinados para executar tarefas específicas no caso de ocorrência de sinistros?
- Existem telefones internos de emergência para comunicação de sinistros?
- computador e a fitoteca são protegidos do fogo, existindo mangueiras e extintores com dióxido de carbono?
- Existem detetores de fumaça:
 - a) Sob o piso falso?
 - b) No teto?
 - Os detetores de fumaça são mantidos e testados de forma programada?
- Existem quadros de controle para detectar e rapidamente localizar fogo e fumaça?
- As placas do piso falso são facilmente removíveis, para permitir verificação de fogo e fumaça?
- Existem marcações no piso para facilitar a localização dos detetores?
- Existem saca-placas na área do computador para cases de emergências?
- Os extintores estão distribuídos estrategicamente em locais visíveis e destacados?
- Os empregados foram instruídos no uso aos extintores.
- É proibido fumar na sala do computador e fitoteca?
- O mobiliário é feito de material não-combustível na área do computador?
- As chaves de emergência desligam o sistema de ar condicionado?
- Existe iluminação de emergência na instalação:
 - a) Em pontos estratégicos?
 - b) Nas rotas de evacuação?
- Alarme de incêndio toca na vigilância?
- Os vigilantes são treinados a combater incêndios que possam ocorrer fora do expediente normal?
- Os funcionários são treinados a combater incêndios que possam ocorrer na área do computador?
- Existem válvulas (portal) corta-fogo nos dutos de ar condicionado?
- Existe um sistema de alarme para fazer a evacuação dos prédios?
- Existem extintores de água pressurizada ou hidrantes nas áreas de almoxarifado?
- Inflamáveis usados na limpeza de fitas são guardados em local seguro e contém indicação do nome e conteúdo?
- É mantido um relacionamento formal com a guarnição do Corpo de Bombeiros que atende a região?

- Existem hidrantes instalados em locais estratégicos nos andares dos prédios?
- Esses hidrantes e seus equipamentos auxiliares são testados regularmente?
- É proibido o uso de coletor de lixo de madeira, papelão ou plástico na sala do computador?
- Os prédios são dotados de saída de emergência para auxiliar nos momentos de evacuação do pessoal?
- Essas saídas são sinalizadas?
- Havendo necessidade, os carros do Corpo de Bombeiros podem ter acesso fácil a qualquer lado do prédio?
- O sistema de hidrantes está eficientemente instalado?
- Os hidrantes são marcados e de fácil visualização?
- Existe um sistema de alarme, como sirene, que sinaliza nos momentos de evacuação de pessoal?
- Existe um sistema de áudio para auxiliar nos momentos de evacuação de pessoal?
- Se existe esse sistema de áudio, ele é separado por áreas ou andares? Existem procedimentos sobre evacuação de pessoal?
- Existem equipamentos portáteis, disponíveis em pontos estratégicos para auxiliar a evacuação de pessoal?
- A sala do computador é separada das áreas adjacentes por divisórias e portas de material retardante?
- As placas do piso falso são de material retardante?
- Existe reserva técnica de água para hidrantes?
- Os extintores e hidrantes da instalação são de fácil acesso?
- As saídas de emergência são mantidas desempedidas?
- Existem plantas atualizadas da rede hidráulica?
- Existem brigadas de incêndio organizadas?
- Existem plantas de localização dos extintores e detetores?

Inundação

- Os encanamentos, exceto os necessários, foram retirados do piso falso e áreas sobre o computador?
- Os conduítes são a prova d'água?
- A vedação é adequada contra infiltração de água:
 - a) Nas portas externas?
 - b) Nas janelas externas?
- Existem escoamento de água e drenagem adequados para impedir inundação na sala do computador?
- O armazenamento de papéis é feito em áreas protegidas contra umidade?
- As torres de resfriamento de água estão construídas em local fora do prédio que abriga o computador?
- Há bombas de água instaladas para permitir drenar áreas localizadas em terrenos baixos?
- A sala do computador se encontra em nível acima da rua?
- A sala do computador possui impermeabilização adequada do teto, impedindo infiltração de água?

Energia Elétrica

- Existe sistema de geração própria de energia de back-up?
- A entrada de força é controlada para evitar transientes elétricas?
- No caso de falta de luz, existe iluminação de emergência para a remoção do pessoal?
- Os alarmes contra fogo podem ser alimentados por baterias, no caso de falha no fornecimento de energia?
- O sistema de geração própria de energia é testado periodicamente?
- Existem pára-raios?
- A rede possui barramento próprio para os diversos equipamentos (computadores, ar condicionado etc.)?
- Existe regulador de tensão na alimentação dos computadores?
- Os cabos das instalações sob o piso elevado estão acomodados em calhas para sua proteção?
- As tomadas fêmeas para os equipamentos possuem ligação de aterramento? Existem plantas atualizadas de toda a rede elétrica?
- Existe esquema de plantão no serviço de eletricidade?
- O quadro de força é protegido, mas de fácil acesso para as situações de emergência?
- Existe um sistema de desarme automático para proteção a rede contra curto-circuito?
- Existe um controle das perdas de horas de máquina que são devidas aos problemas de eletricidade?
- Existem avisos nos locais onde haja "alta tensão" ou "voltagem perigosa"?
- Existe um plano de manutenção para a rede elétrica?
- A subestação de força se encontra na parte mais baixa da edificação?

Ar Condicionado

- Existe sistema de ar condicionado dedicado à área do computador?
- Equipamentos de ar condicionado estão instalados em compartimentos fechados (acesso a pessoal autorizado)?
- As tomadas de ar são protegidas de contaminação?
- Existe back-up de ar condicionado pela existência de um segundo compressor e torre de arrefecimento?
- Existe manutenção preventiva para o compressor e outros equipamentos do sistema de ar condicionado?
- A temperatura e a umidade são registradas e controladas na área do computador?
- Existem alarmes nos sistemas de ar condicionado?
- Os autos do ar condicionado são de material retardante?
- Instrumentos de comando do sistema de ar condicionado estão protegidos evitando manutenção não-autorizada?
- Existe uma limpeza programada para os filtros de ar condicionado?
- Existem plantas com especificações de toda a rede de ar condicionado?

Acesso

- A área do computador fica em local não visível da rua?
- Se a área do computador é visível para o público em geral, são as janelas de material inquebrável?
- Existe um serviço de vigilância de 24 horas, inclusive nos fins de semana e feriados?

- Existem métodos os para verificar o acesso indevido a área do computador?
- número de acessos para a área do computador e mantida em um mínimo?
- estado das saídas de emergência e verificado periodicamente?
- As portas para a área do computador são mantidas fechadas?
- Os empregados demitidos são imediatamente removidos do ambiente de trabalho e a vigilância e prevenida?
- Existem alarmes para informar a vigilância a violação de portas e acessos a áreas de segurança?
- pessoal da vigilância e informado sobre empregados que irão trabalhar fora do período normal?
- Ha empregados designados a acompanhar o pessoal de fornecedores fora do expediente normal?
- Os visitantes são identificados com crachás?
- O acesso a área do computador e da fitoteca é restrito aos funcionários autorizados?
- É controlado o ingresso na empresa de pessoas estranhas com registro em formulários especiais?
- É feito um rodízio periódico entre os recepcionistas?
- É proibida a visita a funcionário, que não seja por assuntos de serviços?
- É proibida a entrada de vendedores, cobradores, demonstradores etc.?
- Existe algum tipo de treinamento específico para os vigilantes?
- A rede de iluminação, está bem distribuída, com luzes indispensáveis acesas em horas fora do expediente?
- Periodicamente são verificados os antecedentes criminais dos vigilantes?
- Existe um programa de inspeções incertas para verificar o funcionamento da vigilância?
- Em casos de terrenos cercados, existem guaritas para o serviço de vigilância?
- Estão os vigilantes devidamente instruídos para procedimentos de emergência?
- corpo de vigilantes possui um manual com procedimentos de emergência?
- Existe um sistema de clavículário no Corpo de Vigilantes?
- Pessoas estranhas aos quadros da empresa são escoltadas quando estão trabalhando em área de segurança?
- pessoal de áreas de segurança e instruído a questionar pessoas sem identificação?
- A vigilância é proibida de fazer a inspeção dentro da sala do computador?
- Ha um controle rigoroso das chaves das portas?
- Os funcionários são obrigados a usar identificação apropriada?
- pessoal de serviço é obrigado a usar identificação visível?
- pessoal de limpeza e manutenção e instruído quanto a medidas de segurança adotadas?

Sala do Computador

- piso(sob o piso falso) é conservado limpo?
- As cestas de lixos são constantemente esvaziadas para prevenir que o excesso de lixo transborde?
- É proibido fumar na sala do computador?
- É proibido comer e beber na sala do computador?
- Os equipamentos são conservados limpos por dentro e por fora?

- Periodicamente é verificada a necessidade de efetuar dedetização e desratização?
- A Gerência e a Supervisão inspecionam as áreas para mantê-las em ordem?
- As cestas de lixo são de metal com tampa com objetivo de abafar princípios de incêndio?
- Existe uma programação para remover caixas vazias e formulários não aproveitados?
- E proibida a execução de trabalho que gerem poeira na área dos equipamentos?

Segurança Física

Hardware

- Os disc-packs são testados e mantidos limpos?
- A limpeza das unidades de fita é feita de forma programada?
- Os erros de fita são registrados, para permitir avaliar as condições gerais da fitoteca?
- São usados protetores para impedir que as fitas desenrolem quando armazenadas? Todas as fitas são registradas e controladas quando retiradas da fitoteca?
- É feito um inventário físico periódico para garantir que todas as fitas e discos estão sob controle?
- Acesso a fitoteca é restrito às pessoas autorizadas?
- Existe fitoteca de segurança distante (fora do prédio) da fitoteca de produção para fitas e discos?
- A armazenagem fora da fitoteca tem segurança no mínimo igual à da fitoteca?
- Existe uma verificação por pessoas especializadas para constatar o uso indevido de espaço?
- Existe back-up para seu equipamento na empresa?
- Ele absorve a carga de trabalho da instalação?
- Existe contrato de back-up que permita o acesso a outro equipamento?
- A instalação de back-up possui segurança compatível com as necessidades de sua instalação?
- Ela absorve sua carga de trabalho?
- Existe um plano de transferência que permite o deslocamento dos serviços para a instalação de back-up?
- O plano de transferência é testado e revisto periodicamente?
- Existe um esquema regular para sua manutenção?
- O fornecedor de seus equipamentos possui almoxarifado de peças de reposição na localidade?
- A manutenção preventiva das instalações é realizada com base em programa acordado com a produção?
- Existe algum documento que autorize a saída de fitas, disco e listagens pela portaria principal?
- Existe algum controle das falhas dos equipamentos?

Software

Atualização da distribuição (automática para os pacotes de segurança)

- Firewall, fail2ban e denyhosts
 - Usuários, grupos e privilégios
 - Enxugar usuários
 - Enxugar pacotes, processos e serviços
 - Monitorar serviços, recursos, processos, logs, arquivos, etc
 - Backup frequente do SO
 - Backup frequente e confiável dos bancos de dados
 - Política de senhas fortes

 - controle revê a documentação dos sistemas para verificar sua concordância com os padrões operacionais?
 - A documentação de back-up e revista periodicamente para se certificar de que ela está atualizada?
 - É feito anualmente o inventário:
 - Do arquivo de programas?
 - Do arquivo da documentação de programas?
- são utilizadas senhas para identificar um terminal ou seu usuário?
- A capacidade de incluir, excluir ou alterar arquivos é limitada pelo sistema?
 - acesso as senhas é restrito?
 - Quando um arquivo de dados é utilizado, gera-se back-up automaticamente, como garantia?
 - Qualquer alteração em sistema, ou documentação somente é feita após autorização do analista responsável?
 - Os operadores de terminais são identificados a cada tarefa executada, a fim de estabelecer responsabilidade?
 - Os arquivos de segurança:
 - são identificados diferenciadamente na fitoteca?
 - São copiados e arquivados na fitoteca de segurança?
 - Os dados de segurança são codificados, de tal maneira que tome impossível o acesso por estranhos?
 - Qualquer sistema alterado tem sua documentação simultaneamente alterada?
 - Os documentos de entrada dos sistemas enviados pelos clientes são controlados?
 - Os relatórios inservíveis impressos, contendo dados sensíveis, são destruídos?
 - A documentação do software está localizada em ambiente seguro?
 - acesso aos arquivos protegidos e controlado:
 - a)Por rotinas automáticas?
 - b) Pela operação?
 - Existe back-up para:
 - Biblioteca de programas?
 - Documentação do software?
 - Existem vários níveis de controle para os arquivos de dados on-line?
 - Existe proteção aos dados através de palavras-chaves?
 - As palavras-chaves são alteradas com freqüência e com periodicidade apropriada a segurança?

- As violações ou tentativas de violações de arquivos protegidos são registradas?
- O acesso aos arquivos protegidos é controlado:
- Por rotinas automáticas?
- Pela operação?
- Essa proteção é testada periodicamente para garantir que os procedimentos são adequados?
- São utilizadas técnicas especiais de proteção durante a transmissão de dados?
- As alterações de programa são:
 - a) Controladas?
 - b) Documentadas?

Arquivo de Dados

- Cópia dos arquivos de segurança são mantidas em outro prédio ?
- Essas cópias são atualizadas?
- Os back-up são verificados periodicamente?
- Os arquivos são discriminados por níveis de segurança?
- Existe armazenamento especial (cofre) para “arquivos de segurança”?
- Os manuais de procedimento de operação têm back-up?
- O acesso aos arquivos é controlado pela fitoteca?
- O acesso à área de fitoteca é restrito a pessoas autorizadas?
- É proibido aos analistas e programadores permanecerem de posse de fitas (fora da fitoteca)?
- São efetuadas palestras para conscientizar os empregados da importância da segurança com discos e fitas?
- Há idéias de custos (R\$) e dos discos envolvidos na perda de arquivo de dados e programas?
- A sistemática de controle de fitas e discos permite:
- Conhecer a freqüência de uso?
- Conhecer a freqüência de limpeza?
- As rotinas de retenção-liberação de arquivos de dados são previstas para todos os sistemas?

Controles Internos

- Existe back-up para programas?
- Existe back-up para arquivos?
- Os analistas e programadores são alertados para não deixarem expostos relatórios reservados?
- Existe auditoria no desenvolvimento de sistemas?
- Os auditores têm suficiente conhecimento de programação para garantir que a codificação não é fraudulenta?
- As alterações de programa são controladas?
- Existem controles para verificar impressão de relatórios?
- O accounting da máquina está disponível para controle interno?
- Existe controle de qualidade das saídas?

- Os formulários sensíveis (cheque, pagamento etc.) são armazenados em local seguro?
- manuseio de formulários é feito de forma a prevenir furtos?
- É proibido aos operadores fazer programas ou modificações de processamento por intermédio de console?
- Existe controle do uso do computador?
- Os auditores participam do desenvolvimento dos planos de segurança?
- Existe controle de qualidade dos suprimentos (formulários, fitas etc.) utilizados?
- São feitos inventários periódicos de todo material existente?
- Os analistas e programadores são impedidos de operar o computador?
- Existe esforço de auditoria para determinar se os procedimentos de segurança estão sendo cumpridos?
- No planejamento (modernização ou aumento) de sua instalação, são consideradas necessidades de segurança?
- Foram apropriadamente identificados e anotados os recursos necessários a segurança física da instalação?
- Os recursos necessários a segurança foram considerados no orçamento?
- A organização é estruturada para que tarefas sejam bem divididas, minimizando oportunidades de fraude?
- Os assessores de segurança participam de cursos específicos a sua função?
- Os incidentes de segurança são investigados para apuração da causa e tomada de ação corretiva?

Correspondência e Comunicação

- Há um sistema especial de segurança para a entrega de relatórios?
- Os arquivos de correspondência de cada setor são fechados após o expediente normal?
- Nesses arquivos, há separação entre a correspondência normal e confidencial?
- É checado o material que deixa a recepção com destinos diversos?
- Os quadros de conexões telefônicas são trancados e de acesso somente permitido a pessoal autorizado?
- Na sala de telex somente e permitido o ingresso de pessoas autorizadas?
- É evitado que documentos confidenciais sejam vendidos como papel velho?
- Quando discos ou fitas são transportados, são mantidas duplicatas deles como garantia?
- Documentos tais como correspondências, relatórios e outros são classificados quanto ao grau de sigilo?

1) Segurança Física

1.1) Segurança do Hardware

1.2) Segurança no Acesso Físico ao Servidor

2) Segurança Lógica do SGBD

2.1) Segurança relativa aos grupos, usuários e privilégios (Capítulo 4)

2.2) Segurança do projeto dos bancos dados e seus objetos e consultas e código SQL

2.3) Segurança dos aplicativos (senhas, usuários, etc)

1) Segurança Física

1.1) Segurança do Hardware

Atualmente, segurança para servidores de bancos de dados não é uma opção mas um requisito, especialmente se o servidor estiver numa rede. Conhecer como permitir aos clientes chegarem ao SGBD e como evitar que outros não cheguem é uma tarefa importante para o DBA.

É importante que o hardware onde encontra-se o servidor do PostgreSQL, seja um hardware além de robusto fisicamente, seguro. Para isso diversos cuidados devem ser tomados:

- Fontes redundantes
- RAID com HDs

Muitos outros cuidados semelhantes, em especial que a origem do hardware seja confiável e tenha uma boa garantia.

Importante uma leitura do **PostgreSQL Hardware Performance Tuning**:

http://www.postgresql.org/files/documentation/books/aw_pgsql/hw_performance/

Traduzido: <http://www.vivaolinux.com.br/artigos/impressora.php?codigo=5930>

1.2) Segurança no Acesso Físico ao Servidor

Sem garantir segurança física ao servidor de nada vai adiantar os cuidados com a administração do mesmo, com usuários, senhas, etc. Primeiro garantir boa segurança no hardware, depois garantir que somente pessoal de confiança tenha acesso ao servidor. Finalmente a segurança lógica.

A segurança física, de acesso ao servidor, deve ser levada em conta e receber a devida atenção para que as informações possam de fato ser protegidas.

Por padrão, o PostgreSQL é instalado no Windows em: C:\Program Files\PostgreSQL\8.3 e no Linux, quando através dos fontes em: /usr/local/pgsql

Para saber o nome dos arquivos dos respectivos bancos de dados podemos usar a consulta:

```
select oid, datname from pg_database;
```

Uma forma de esconder um pouco alguns bancos é criando um Tablespace e os armazenando em outro diretório. Vide capítulo 2 do módulo 2.

Em termos de Sistema Operacional, tudo indica que um Linux seja mais seguro para um servidor do SGBD PostgreSQL, devido ao seu sistema de permissões e controle de usuários, que é mais exigente que o de um Windows. Sem contar que existem indícios de ser um ambiente para maior desempenho.

A Segurança da Informação Pessoal e Corporativa

O conceito de segurança da informação não está ligado somente à computadores e seus sistemas, este é um termo muito mais amplo utilizado para dar a idéia de segurança de dados pessoais ou corporativos.

O termo sempre é associado à segurança de informações digitais, mas devemos nos lembrar que a informação pode estar em qualquer mídia, um cd-rom, um pendrive, uma folha de papel, um bloco de notas, uma agenda...

Imagine que você tem uma agenda, dessas de papel (lembra que isso existe?), e nesta agenda você tem todos os seus dados pessoais, compromissos, telefones de amigos e familiares, contatos profissionais, informações bancárias e etc. Agora imagine que você perde essa agenda... O que fazer? Toda a sua vida está nela, será muito fácil alguém se passar por você de posse de todas essas informações.

Da mesma forma funcionam seus dados digitais, praticamente todos que usam computadores mantém algum tipo de dado pessoal armazenado nele. E a segurança disso? A diferença da a boa e velha agenda de papel é que os computadores podem ser acessados à distância, sem você saber.

Porque é mais fácil conseguir informações no meio digital? Ora, encontrar uma agenda na rua não é algo tão comum, mas encontrar sistemas de informação sem proteção ou com proteção fraca é.

Vamos pegar como exemplo uma rede corporativa com, por exemplo, um sistema acessado por meio de autenticação com senha. Esse sistema foi projetado e concebido com a idéia de ser seguro, ele usa conexão criptografada e ainda pede para a pessoa digitar algo aleatório exibido em uma imagem (Captcha) para evitar ataques de força bruta. Certo, até aqui temos um bom sistema, um sistema seguro e com uma bela muralha na frente. Qual seria o principal ponto fraco desse sistema supondo que não há bugs ou falhas exploráveis na autenticação dos usuários? Quer uma dica? Começa com 'Usu' e termina com 'ários'. É isso mesmo, de que adianta o alto investimento em um sistema seguro se o usuário anota a senha em um bloco de notas e o deixa em cima da

mesa à vista de qualquer um? O que acontece quando o usuário usa como senha o próprio nome? Ou palavras simples? Ou mesmo seqüências de teclas no teclado como 'qwerty' e 'asdfg'? Ou mesmo o famoso '123'?

Pronto a segurança foi para o espaço, pouco adiantou investir alto no desenvolvimento de algo seguro.

Todas as empresas, usando computadores ou não, deveriam ter um mínimo de preocupação com a segurança de informações. Estabelecer regras de comportamento para os funcionários, treiná-los para seguir essas regras, monitorar constantemente se estão seguindo tudo de conforme definido.

A falta de regras e treinamento pode levar os usuários a ter um 'comportamento de risco'.

Políticas de boa utilização de dados e de segurança devem fazer parte da vida de todos, não apenas de grandes corporações, as micro e pequenas empresas estão sujeitas a ataques também, aliás os usuários domésticos correm riscos igualmente!

É claro que o pensamento 'quem vai querer me invadir?' existe. Afinal de contas quem vai querer invadir o meu pc? Eu não tenho informações de grande valor mesmo. É, mas com várias pequenas informações e muitos 'bots' se pode fazer muita coisa ilegal. Imagine a Polícia Federal chegando em sua casa com a acusação de que centenas de e-mails SPAM partem da sua conexão diariamente. Isso sim é uma situação desagradável, como provar que não é você o responsável se o programa que envia as mensagens está instalado no seu computador e disparando o tempo todo? Pois é meu amigo, ou você acha que para esse tipo de coisa os cybercriminosos invadem apenas os servidores da Nasa?

Como você pode proteger suas informações e seu computador de possíveis ataques?

- Mantenha um bom firewall ativo

Ele bloqueia conexões indesejadas.

- Não clique em links recebidos por e-mail

Mesmo que você tenha, supostamente, ganhado \$ 1mi.

- Não divulgue seu e-mail de forma irresponsável

Bots de rede varem sites em busca de e-mails

- Tente não seguir as malditas correntes de e-mail

Você sabia que seu IP vai no e-mail enviado?

- Nunca responda e-mails de desconhecidos

- Não aceite programas que lhe enviam por e-mail ou softwares de mensagens instantâneas

Para isso vá ao site do desenvolvedor e baixe o software ou compre

- Não utilize software pirata

Eles vem com cracks que, além de quebrar a segurança do programa, contém vírus, spyware e malware em geral.

- Não utilize senhas simples!

Entrar no e-mail de alguém que deixa a senha 'qwerty' não pode nem ser considerado crime, já que essa senha e senha nenhuma nenhuma senha são praticamente equivalentes.

- Mantenha um bom anti-vírus

Sim, esse é um dos fatores principais (Ainda mais se você usa o sistema número um em quantidade de vírus). Preciso dizer que anti-vírus 'crackeado' é inútil?

- Proteja seu computador com senha

O atacante pode ser uma visita que liga o pc e pega as informações.

- Não acesse o site do banco em lan-houses!

Nestes lugares podem estar ativos os chamados Keyloggers que guardam todas as teclas digitadas.

Ou seja, a grande sacada é agir com bom senso e evitar armadilhas.

Nunca dê seus dados à pessoas que ligam oferecendo serviços e produtos ou mesmo dizendo ser do banco, da Receita Federal, etc... Sempre que precisa ou desconfiar de algo vá ao banco ou ligue, vá ao posto de atendimento da Receita, mas não acredite em ligações!

A segurança da informação é algo que é praticamente impossível de ser 100% efetiva, sempre existem falhas humanas, falhas nos softwares, falhas nos cadeados, agendas se perdem...

Mas agindo com bom senso, guardando bem os dados pessoais, protegendo as informações nos computadores e identificando possíveis golpes é possível tornar seu dia-a-dia bem melhor.

Não pense que ninguém quer suas informações porque elas não valem nada, elas valem sim, mas eu tenho certeza que para você e sua empresa elas não têm preço.

<http://infog.casoft.info/?p=32>

2) Segurança Lógica do SGBD

Jamais aconselharia deixar dessa maneira para um banco em produção. Com isso o usuário postgres poderia ter acesso (mesmo que com senha) de qualquer máquina da rede. Aconselho a permitir conexões com o usuário postgres apenas para os endereços 127.0.0.1/32 e o ip do servidor/32. E não use o usuário postgres como usuário de conexão em nenhum dos casos. Tenha um usuário apenas com os GRANTs necessários para não deixar a segurança da sua base de dados vulnerável.

Fernando Brombatti na lista pgbr-geral.

Cada aplicação deve ter um usuário com acesso apenas ao necessário para usar a aplicação.

As configurações default do postgresql rejeitam toda conexão de outros computadores e usam

a autenticação do tipo ident para gerenciar o acesso de usuários com mesmo nome no sistema operacional (isso em Linux/UNIX/BSD, não no Windows).

As versões atuais também suportam autenticação tipo LDAP.

Segurança no Sistema de Arquivos

Por default quando instalamos o PostgreSQL no Linux através dos fontes ele é instalado em:

/usr/local/pgsql

No Windows uma instalação com o instalador ele fica em:

C:\Arquivos de Programas\PostgreSQL

Os programas executáveis, como o pg_dump, o psql e outros ficam num subdiretório bin desse diretório de instalação.

Os bancos, logs e outros arquivos ficam no subdiretório data. Dentro desse sibdiretório data ficam alguns diretórios importantes como o base (que abriga os bacos de dados), o global (que guarda informações de todos os bancos) e o pg_xlog (guarda os logs de transações).

Os bancos que ficam no diretório data/base são gravados com números dos OIDs. Para saber que banco corresponde aos OIDs podemos usar uma consulta à tabela de sistema pg_database:

```
dnocs=# select oid, datname from pg_database;
```

oid		datname
10819		postgres
16400		template_postgis
16816		iniciante
16821		dnocs
1		template1
10818		template0
16835		biblioteca

Uma boa prática é monitorar as informações detalhadas sobre os arquivos e diretórios do PostgreSQL: permissões, data de criação e modificação, dono, etc.

Lembrando que no Linux a permissão de execução em diretórios permite listar o diretório e o acesso ao diretório.

Coneções Remotas

Também devemos tomar precauções ao conectar remotamente ao servidor, usando conexões seguras como SSH.

Veja outras recomendações sobre Conexões TCP/IP seguras por túneis SSH em:

<http://pgdocptbr.sourceforge.net/pg80/ssh-tunnels.html>

E Conexões TCP/IP seguras com SSL em:

<http://pgdocptbr.sourceforge.net/pg80/ssl-tcp.html>

2.1) Segurança relativa aos usuários

Em relação aos usuários, grupos e privilégios, já vimos na aula 4: 4) Administração de Grupos, Usuários e Privilégios.

Esta segurança é interna do SGBD. Mas antes que chegue ao SGBD deve passar por uma barreira de segurança chamada Firewall, depois de passar pelo Firewall deve ainda passar pelos scripts postgresql.conf, pelo pg_hba.conf e. pg_ident.conf (caso esteja num Linux).

Firewall – esta é a mais básica barreira de segurança de um servidor que encontra-se numa rede. Ele nos permite filtrar que clientes irão passar pelo firewall para que aplicações. Enquanto podemos ter um único firewall protegendo toda uma rede, também podemos ter um firewall protegendo um único servidor. Quando habilitamos um firewall, por default, está tudo bloqueado. A partir daí precisamos desbloquear cada cliente que deve ter acesso.

postgresql.conf - Este é o primeiro script que o cliente remoto encontra quando pretende conectar com o servidor do PostgreSQL. O acesso remoto e muitas configurações importantes começam neste script. Para que clientes remotos tenham acesso precisamos alterar a configuração padrão de localhost para *:

```
listen_addresses = '*'
```

pg_hba.conf - O próximo passo é a configuração do pg_hba.conf, que define que usuários podem conectar a que bancos, usando que IP ou rede com respectivas máscaras e com que sistema de autenticação. Cada linha define uma individual regra de acesso. Somente terá acesso se satisfazer a todas as colunas.

O pg_hba.conf permite linhas de três tipos: comentários, em branco e registros (linhas válidas ao final). Os registros podem ser separados por tabulação ou espaços. Espaços no início e ao final são ignorados. Um registro obrigatoriamente deve ser contido em uma única linha.

As colunas de cada linha do script são:

conxão-tipo	banco	usuário	end_rede	método-login	opções
-------------	-------	---------	----------	--------------	--------

Exemplo:

```
hostssl      all      all 10.0.1.0/28      password
host        teste    joao   192.168.120.5    md5
host        teste2   joao   192.168.120.5    md5
host        teste3   all    10.0.2.0/28      md5
```

O tipo de conexão pode ser:

local – conexão local ao SGBD,

host – conexão permitida somente quando existe o suporte à conexões tipo TCP/IP.

Hostssl – deve estar habilitado o suporte a SSL.

Para testar uma conexão remota podemos usar um cliente como o psql:

```
psql -h 10.0.1.33 teste joao
```

Habilitando SSL no PostgreSQL

A instalação for Windows já suporta SSL por default. No Linux devemos verificar ou habilitar.

Habilitar no postgresql.conf:

```
ssl = on
```

Após restartar o servidor ele estará escutando por conexões normais (TCP) e conexões SSL (SSL TCP) na mesma porta. Logo que conecta com suporte a SSL o PostgreSQL procura a chave de criptografia e os arquivos do certificado no diretório 'data' e não iniciará até que encontre os mesmos. Vide capítulo 3 do Livro "PostgreSQL 8 for Windows".

Obs.: Nunca use o tipo de autenticação **trust** em redes que não ofereçam uma boa segurança.

O tipo de autenticação **ident** deixa a cargo do cliente a segurança.

Antes de efetuar alterações no tipo de autenticação do pg_hba.conf, por exemplo, para md5, conceda senha ou as altere para todos os usuários, como a seguir:

```
CREATE ROLE usuario WITH ENCRYPTED PASSWORD 'senha';
```

```
ALTER ROLE usuario WITH ENCRYPTED PASSWORD 'senha';
```

Obs.: Se a senha for em texto claro, do tipo **password**, não use a palavra ENCRYPTED

Após isso altere o pg_hba.conf e restart o servidor.

Rejeitando Conexões

Um dos métodos de autenticação é o **reject**, que pode ser aplicado em caso de suspeita ou certeza de conexão não desejada.

Exemplo:

```
host all      192.168.0.15      255.255.255.255 reject
```

Monitorando Usuários

Através do PGAdmin podemos monitorar muito bem todos os usuários e suas ações.

Após conectar clique em Ferramentas – Status do Servidor.

Aí podemos monitorar o PID, usuário, banco, IP, início da conexão, consultas, etc.

Como também os bloqueios, as transações e o arquivo de Log. Até rotacionar o arquivo de Log, caso necessário e caso o usuário conectado tenha este privilégio.

Referência: Capítulo 10 do Livro PostgreSQL 8 for Windows.

2.2) Segurança do projeto dos bancos dados e seus objetos e consultas e código SQL

Esta parte diz respeito ao código SQL das consultas, que deve contemplar pelo menos as 3 formas normais, sempre usar chaves primárias nas tabelas, índices em campos muito consultados na cláusula WHERE e boas constraints que venham a melhorar a segurança das consultas.

Também vale lembrar que é útil o uso de VIEWS, de funções e procedures.

Usuários autorizados podem se aproveitar da estrutura dos objetos do SGBD para ter acesso ao que não devia.

2.3) Segurança dos aplicativos (senhas, usuários, etc)

Este tópico deve fazer parte da política de segurança da empresa/instituição para suas informações.

Aplicativos devem usar senhas fortes e renovar as mesmas com certa periodicidade.

Veja alguns bons artigos sobre segurança com senhas:

<http://www.microsoft.com/brasil/technet/Colunas/DiogoHenrique/SenhasAltaSeguranca2.mspx>

Recomendações e Procedimentos de Segurança :

<http://www2.iq.usp.br/sti/index.dhtml?pagina=754&chave=89N>

Recomendações para Política de Senhas:

http://www.brasilacademicocom/maxpt/article_read.asp?id=190&ARTICLE_TITLE=Recomenda%E7%F5es+para+Pol%C3%ADtica+de+Senhas&CATEG_Title=Dicas%3A+Seguran%E7a

Capítulo do Livro sobre PostgreSQL a ser escrito pelo Fábio Telles:

Segurança no PostgreSQL - Parte I: "A Saga"

Já faz um tempo que ando investigando a parte de segurança em Bancos de Dados. Passei um tempão estudando a parte de segurança no Oracle e andei investigando

algumas questões sobre o assunto [aqui](#) e [acolá](#) neste blog e na [palestra sobre melhores práticas](#) que fiz no [FISL 9.0](#) e no [PGCon Brasil 2007](#). Calhou de ontem eu estar conversando com o [Dickson Guedes](#) no IRC e resolvemos escrever um pouco sobre segurança no PostgreSQL. A idéia é para variar um pouco um tanto ambiciosa: fazer uma lista de possíveis melhorias que seriam interessante implementar no PostgreSQL para melhorar a segurança. Uma segunda etapa seria colocar a mão na massa e tentar implementar algum patch no PostgreSQL. Particularmente eu tenho muito receio de abrir o código fonte e sair mexendo. Manjo pouco de C e não tenho tanto domínio assim do funcionamento interno de SGDBs para fazer isso. Mas por outro lado, se eu não fizer o [fike](#) nunca vai me deixar em paz... e ao fim e ao cabo, a gente só aprende a fazer fazendo!!! Por fim, existe uma segunda intenção, que é começar a escrever coisas mais detalhadas sobre este assunto, iniciando assim alguns capítulos do [livro sobre PostgreSQL](#) que começa a sair do papel e entrar no ar. Particularmente, eu acho que só a parte de segurança já é suficiente escrever mais de 100 páginas. Bom, de toda forma, todos aqueles que lerem este texto e sentir que há algum equívoco ou que falta alguma coisa está convidadíssimo a colaborar, seja nos comentários, seja enviando um e-mail ou mesmo no IRC.

Caramba... mas segurança é um tema tão complexo assim? Bom, depende de como você define o que é segurança em SGDB para você. Vamos começar pensando em duas ou três formas de se enarar o problema:

- Segurança em SGDB não se refere apenas ao seu banco de dados, diz respeito a toda a cadeia tecnologia associada a sua aplicação, uma vez que um único elo fraco da sua solução de TI põe em risco o conjunto como um todo. Então não basta pensar no SGDB isoladamente. Temos que pensar no equipamento onde o SGDB se encontra, no SO, na instalação do SGDB, nos dados contidos no banco de dados, na aplicação acessando o banco de dados e finalmente no usuário que precisa destas informações;
- Todos estão carecas de saber, mas não custa repetir: os bancos de dados guardam um patrimônio valioso: informação. Quando você tem um problema de segurança num SGDB, são os seus dados que estão ameaçados. A pergunta que sempre se faz é “quanto vale a informação guardada nos bancos de dados?”. Melhor pergunta seria: “quanto custa a perda destes dados?”.
- Segurança tem haver com coisas com o potencial de tirar o sono/emprego de toda uma equipe de TI. O desempenho é importante. Algumas vezes é muito importante. Mas em geral, a segurança vem em primeiro lugar. Esta importância hoje tem nome, endereço, telefone, e-mail e tudo o mais. Depois da onda gerada pelo [SOx](#) nos Estados Unidos, a segurança passou a ser mais importante ainda. O [COBIT](#) e o [ITIL](#) e suas normas e regulamentações associadas representam mais uma pressão na busca por mais segurança em torno das informações.

Muito bonito tudo isso... mas enfim, quando falamos em segurança qual é a pauta em questão? Ah sim... muitas coisas, vejamos algumas:

- **Integridade:** tem muito haver com ACID. Seus dados não podem se corromper, independente do que venha a acontecer. Mesmo que seus dados se tornem indisponíveis durante um certo período (devido a uma queda de energia, falha na rede ou num disco por exemplo) você tem que garantir que uma vez que o problema de indisponibilidade esteja resolvido, todos os dados tem que reaparecer intactos. Integridade também tem muito haver com o uso de restrições de integridade, para que um erro do usuário ou da aplicação não permita que os dados sejam corrompidos. O uso das restrições de integridade devem proteger seus dados contra alterações que não estão de acordo com as suas regras de negócio;
- **Perda de dados:** a preocupação número 1 de todo administrador de banco de dados é evitar ao máximo a perda de dados. Aí entra em cena a mais tediosa das tarefas: o backup. Ocorre que em bancos de dados, não existe uma única forma de se fazer um backup, existem várias estratégias. De toda forma, é preciso definir antes de mais nada: qual é o volume máximo de dados que admissível perder? Os dados relativos a última semana de operação? Talvez o último dia? A última hora? Nem um segundo sequer? Bom, é claro que ninguém quer perder nada, mas você sabe realmente qual é o grau de proteção que a sua atual política de backup lhe oferece?
- **Disponibilidade:** capacidade de manter o acesso às informações de forma contínua. Falhas humanas, falhas de software e falhas de hardware podem gerar indisponibilidade a qualquer momento. A sua tolerância a indisponibilidade vai variar conforme a importância e ritmo de operação das suas aplicações. Algumas não podem parar nunca e trabalham em regime 24/7. Outras só precisam estar ativas em horário comercial, mas não admitem um minuto sequer de indisponibilidade neste período. Seja como for, é importantíssimo definir qual é o SLA agregado aos serviços prestados pelo banco de dados para que se possa traçar uma estratégia adequada para se atingir estes objetivos. Sua estratégia deve garantir que mesmo ocorrendo uma falhas, mesmo graves, os dados devem estar disponíveis no prazo determinado pelo seu contrato de SLA.
- **Controle de Acesso:** capacidade de que as informações disponíveis no banco de dados estejam disponíveis para as pessoas corretas, que terão permissão de acesso apenas as operações permitidas para o seu perfil. Geralmente o controle de acesso se dá a partir de objetos do banco de dados, mas pode se referir ao acesso de apenas um parte dos dados de um objeto, como apenas algumas linhas de uma tabela. O controle de acesso pode limitar a quantidade de recursos que você pode utilizar no banco de dados. Os recursos podem ser o número de conexões ativas, memória, processador, volume de dados acessados em disco, etc.
- **Auditória:** registro de operações realizadas no banco de dados. O registro pode conter informações sobre quem, realizou que tipo de operação, sobre qual objeto e em qual momento. O registro também pode conter quais foram as alterações

realizadas, permitindo reconstruir os dados num estado anterior se necessário. A auditoria também pode significar monitorar outras condições do banco de dados, como picos de utilização, espaço em disco e outros detalhes que se deseje registrar.

Como se pode ver, ao falar em segurança, temos inúmeros desafios pela frente. Temas consagrados como “Alta Disponibilidade”, “Backup”, “Política de Mínimo Privilégio”, “Trilhas de auditoria” entre outros, fazem parte do dia-a-dia (ou seria noite-a-noite?) dos que se preocupam com segurança. Em grandes equipes de TI, há pessoas especializadas em segurança. Muitas vezes, encontramos Administradores de Bancos de Dados especializados em um ou dois aspectos da segurança, como o controle de acesso e auditoria. Administradores de Dados são especialistas em avaliar problemas de restrição de integridade enquanto os Administradores de Sistemas costumam se preocupar com backup e alta disponibilidade.

Como se pode ver, temos muito assunto pela frente. Não tenho nenhum ímpeto de escrever seguindo uma ordem específica, provavelmente eu deverei escrever numa ordem caótica e tentar juntar as peças aos poucos.

Comentário:

Fonte: Blog do Fábio Telles:

<http://www.midstorm.org/~telles/2008/05/02/seguranca-no-postgresql-parte-i-a-saga/>

Desafios da Segurança de Informação

Autor: Anderson de Sousa Pereira <link.twister at gmail.com>

Desafios

O maior desafio para as empresas é manter a segurança de seus dados, e quando falamos em segurança de dados não devemos ter o luxo de ignorar qualquer tipo de risco que possa comprometer a integridade dos mesmos. E esses riscos envolvem corrompimentos de dados, perda, roubo, entre outros fatores... Isso pode ser ocasionado devido a fenômenos naturais como, furacões, inundações, terremotos... E também podem ocorrer devido á uma má administração.

O maior perigo para uma empresa é ter a falsa sensação de segurança. Afinal, pior do que não se preocupar com invasões, ataques, vírus, tragédias entre outras ameaças é confiar cegamente em uma estrutura que não esteja preparada para impedir o surgimento de problemas.

Por isso, não devemos confiar apenas em software, ou hardware. Quando falamos em segurança da informação, é necessário considerar quatro obstáculos que têm a mesma importância: a natureza, as pessoas, a tecnologia e os processos.

Não adianta investir apenas em um deles e deixar os outros de lado, eles devem ter a mesma atenção. E os quatro devem estar intimamente ligados, afinal um necessita do outro. Quanto a tecnologia, natureza e processos até que não é um desafio tão gritante, o maior desafio mesmo são as pessoas.

Quanto a natureza, nós podemos nos prevenir com um estudo do local onde a empresa será implantada e mantendo copias dos dados em outros locais, afinal até alguns anos atrás aqui no Brasil eu não me preocuparia com furacões e terremotos, mas ultimamente com o aquecimento global tudo é possível, só do ano de 2007 até hoje já tiveram vários tremores em algumas cidades do nosso país, claro que isso é só a ponta do iceberg...

Quanto à tecnologia e processos, desde que sejam bem administrados e usados corretamente se tornam ferramentas essenciais para o crescimento e evolução do negócio. Parece fácil, mas embora a teoria seja lógica, fazer com que sejam bem implantados é uma difícil e árdua missão. Afinal, é aí que entram as pessoas, cada um usando a tecnologia a seu favor e muitas vezes para fazer um mal uso. E quando eu falo de pessoas eu estou me referindo aos usuários (meros mortais) que dedicam suas vidas à atormentar o CSO (Chefe...xD). Grande parte dos incidentes de segurança contam com o apoio, intencional ou não, do inimigo interno. As pessoas estão em toda a parte da empresa.

Os cuidados básicos com as atitudes das pessoas, muitas vezes são esquecidos ou ignorados. Encontrar senhas escritas e coladas no monitor, bem como o repasse de informações sigilosas em ambientes não seguros, como parada de ônibus, reuniões informais são situações muito comuns. Contar com a colaboração das pessoas é simplesmente fundamental para a empresa. Mas tudo deve ser auditado pelo CSO. Mas o ponto principal da preocupação com as pessoas é que os fraudadores irão em busca delas para perpetuar seus crimes.

Mas nem sempre as pessoas tem 100% de culpa, afinal, no mundo cibernético, tem muito lixo e armadilhas que foram projetadas justamente para enganar pessoas, ai fica muitas vezes difícil de controlar os processos que são confiáveis e não confiáveis, então cabe ao CSO orientar e auditar toda e qualquer mudança que possa possa colocar em risco os dados da corporação. Antivírus, anti-spywares, um bom proxy e um firewall são essenciais, mas nada melhor do que orientar e auxiliar os usuários no uso da tecnologia a favor da corporação..

Postado no vivaolinux - <http://www.vivaolinux.com.br/artigos/impressora.php?codigo=8133>

Recomendo a leitura do post:

- <http://marcelokalib.blogspot.com/2008/04/procura-se-segurana-particular.html>

Referências

Livros:

- PostgreSQL – Korry Douglas, Susan Douglas, SAMS, Capítulo 21
- PostgreSQL Developer's Handbook de Ewald Geschwinde, Hans-Jürgen Schönig , SAMS, Capítulo 6
- Boas fontes de informações: Capítulo 23 do livro PostgreSQL The Comprehensive Guide de Korry Douglas e Susan Douglas.
- e Security and Access Restrictions no capítulo 6 do livro PostgreSQL Developer's Handbook de Ewald Geschwinde, Hans-Jürgen Schönig.

Riscos Envolvendo Informações

- Concentração das informações
- Permitir acesso indiscriminado
- Obscuridade
- Concentração de funções
- Falta de controle
- Retenção duradoura
- Relacionamento e Combinação de Informações
- Introdução de erros
- Lealdade
- Acesso não autorizado
- Perda da integridade

Principais Fatores de Segurança da Informação de Clare Less, Telecommunications, fev. 1989

Infra-estrutura

Separação de ambientes

Energia e ar-condicionado
Radiação eletro-magnética

Proteção física

Recursos Técnicos - Integridade de dados - Confiabilidade - Integridade de programas - Gerenciam.de Recursos de PD	Segurança da Informação	Recursos Humanos: - Controle de acesso - Autorização de acesso - Identificação de usuários
	Manutenção - Proteção da privacidade dos dados - Salvaguardas legais - Organização	

Políticas de Segurança

Definir os Objetivos

Que deve constar?

- objetivos
- destino
- propriedade dos recursos
- responsabilidades
- requisitos de acesso
- responsabilização
- generalizações

Procedimentos

Softwares de segurança: firewalls, anti-virus, antispy, etc.

Backups, mídias, local de armazenamento e acesso

Política de distribuição de e-mails

Logins e senha de usuários locais e de e-mails

Referência: Livro Segurança em Informática e de Informações, de Carso & Steffen

O objetivo de tornar seguro o ambiente é o de reduzir os efeitos caso aconteça uma invasão. Para isso uma boa administração, de forma flexível e uma boa política de backup.

Para que o DBA obtenha o máximo do PostgreSQL ele precisa ter um bom conhecimento:

- Teoria relacional de bancos de dados e familiarizado com o SQL ANSI;
- Conhecer como ler código fonte, especialmente em C e compilar código fonte em Linux;
- Poder administrar servidores Linux e se sentir confortável com ele;
- Poder manter, se preciso for, o hardware da rede;
- Conhecer a camada TCP OS, dividir uma rede em subredes e otimizar um firewall

É raro um DBA que conhece bem todas estas disciplinas.

Criando usuários

Existem três formas de criar usuários no PostgreSQL.

- Com SQL

CREATE USER

CREATE ROLE

- Pela linha de comando

createuser

Estas três formas de criar usuário resultam em privilégios distintos para o usuário criado.

Para uma role ordinária, um usuário típico pode:

- Acessar qualquer banco de dados, se o agrupamento usa a política de autenticação default, como descrita no pg_hba.conf

- Criar objetos no esquema public em qualquer banco de dados que o usuário pode acessar
- Criar objetos de sessão (temporários)
- Alterar parâmetros de runtima
- Criar funções definidas pelo usuário
- Executar funções definidas pelo usuário criadas por outros usuários no esquema public

Importante também saber o que usuários típicos não podem fazer

- Não podem criar bancos de dados nem esquemas
- Não podem criar outros usuários
- Acessar objetos criados por outros usuários
- Não pode efetuar login (somente para CREATE ROLE)

Direitos e Privilégios do Superusuário

Usuários comuns não podem executar direitos e privilégios que foram criados para super usuários.

Lembre que uma conta de usuário ordinário pode fazer qualquer coisa que deseja para os objetos que ele é dono.

Não deixe usuários típicos criarem nada.

Testando:

- logar como postgres
- SET SESSION AUTHORIZATION postgres;
- CREATE ROLE usuario1 WITH LOGIN PASSWORD '123';
- CREATE SCHEMA usuario1 CREATE TABLE tabela1(i int);
- INSERT INTO usuario1.tabela1 VALUES(1);
- grant usage on schema usuario1 to usuario1;
- SELECT i FROM usuario1.tabela1;

Logar como usuario1

- SET SESSION AUTHORIZATION usuario1;
- select * from usuario1.tabela1;

Permissão negada, pois este usuário somente pode acessar o esquema (USAGE). Não pode executar mais nada.

- SET SESSION AUTHORIZATION postgres;
- GRANT SELECT ON usuario1.tabela1 to usuario1;
- SET SESSION AUTHORIZATION usuario1;
- select * from usuario1.tabela1;

Agora funcionou.

Removendo Privilégios do usuário1 no esquema public:

```
REVOKE ALL PRIVILEGES ON SCHEMA PUBLIC FROM usuario1;
```

Comandos do psql

```
\d - lista tabelas
\dt? - listar tabelas iniciadas com t
\du - listar roles
```

\z - visualizar permissões

Listar dados, senhas e privilégios dos usuários

```
select * from pg_user;
```

Extraindo definições globais do cluster

```
#!/bin/bash
psql mydatabase << _eof_
set search_path=public,information_schema,pg_catalog,pg_toast;
\t
\o list.txt
SELECT n.nspname||'.'||c.relname as "Table Name"
FROM pg_catalog.pg_class c
JOIN pg_catalog.pg_roles r ON r.oid = c.relnowner
LEFT JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace
WHERE c.relkind IN ('r','')
ORDER BY 1;
\q
_eof_

for i in $( cat list.txt ); do
  psql -c "\d $i"
done
```

Linguagens procedurais confiáveis versus não confiáveis

```
postgres=# select lanname as language, lanpltrusted as trusted from pg_language;
language | trusted
-----+-----
internal | f
c        | f
sql      | t
plperlu  | f
plperl   | t
```

Usuários podem acessar e consultar as funções do catálogo mesmo que não tenhamos dado privilégios para isso.

Para evitar isso precisamos revogar este privilégio.

```
REVOKE ALL ON FUNCTION f2() FROM user1, GROUP PUBLIC;
```

Obtendo o código de origem da função

```
postgres=> SET SESSION AUTHORIZATION user1;
```

```
postgres=> select prosrc as "function f3()" from pg_proc where proname='f3';
```

Invadindo a senha do PostgreSQL

A administração de senha efetiva é fundamental para a segurança em um DBMS. É função do DBA impor uma política de senha aprovada. Uma senha deve consistir em caracteres alfanuméricos escolhidos aleatoriamente que não tenham um padrão discernível. A prática comum dita que as senhas possuem pelo menos seis caracteres e são mudadas frequentemente.

Contas e senhas de usuário do PostgreSQL

A política de segurança da conta de usuário do PostgreSQL é centrada nos comandos SQL que criam e administram a conta do usuário:

- CREATE ROLE
- ALTER ROLE

Obtendo a senha de um usuário do catálogo

```
postgres=# select username as useraccount,passwd as "password" from pg_shadow
where length(passwd)>1 order by username;
```

Gerando uma senha MD5

```
postgres=# select 'md5'||md5('123user1') as "my own generated hash", passwd as
"stored hash for user1" from pg_shadow where username='user1';
```

Existem alguns poucos mecanismos dentro do PostgreSQL que podem impor uma política de senha blindada.

As possíveis limitações de segurança incluem:

- O superusuário não pode impor um número mínimo de caracteres a ser usado para a senha.
- Apesar de haver um parâmetro padrão nas definições da configuração para como a senha deva ser armazenada (descriptografada ou criptografada com um hash MD5), o usuário não pode ser forçado usar um método de armazenamento particular pelo superusuário.
- Não existe nenhum mecanismo que imponha um tempo de vida à conta do usuário.
- O mecanismo que controla o tempo de vida efetivo da senha da conta do usuário se torna irrelevante quando o método de conexão não é PASSWORD ou MD5 no arquivo de configuração de autenticação de cliente do cluster, pg_hba.conf.
- Os parâmetros de tempo de execução do usuário que são alterados pela instrução ALTER ROLE e que foram configurados pelo superusuário ou pelas definições de configuração estabelecidas por padrão no arquivo postgresql.conf, podem ser alterados pelo proprietário da conta do usuário à vontade.
- Renomear uma conta de usuário limpa sua senha se ela foi criptografada.
- Não é possível controlar quem fez as mudanças nas contas de usuário ou quando essas mudanças ocorreram.

pg_hba.conf

Controla o acesso através de registros que são definidos em uma única linha.

É importante usar túneis SSH e criptografia SSL.

Criar Usuários

```
create role teste1 with login password 'senha';
create role teste1 with nologin password 'senha';
alter role teste1 with nologin;
```

31 - Linguagens do Lado do Servidor

Funções no PostgreSQL:

As internas e as definidas pelo usuário.

Internas

- Funções internas (round(), now(), max(), count(), etc).

As definidas pelo usuário são (UDF – User Defined Function):

- Funções escritas em SQL
- Funções em linguagens de procedimento (PL/pgSQL, PL/Tcl, PL/php, PL/Java, etc)
- Funções na linguagem C

Sintaxe de Criação

```
CREATE [ OR REPLACE ] FUNCTION
  name ( [ [ argmode ] [ argname ] argtype [, ...] ] )
  [ RETURNS rettype ]
{ LANGUAGE langname
  | IMMUTABLE | STABLE | VOLATILE
  | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
  | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
  | AS 'definition'
  | AS 'obj_file', 'link_symbol' }
} ...
[ WITH ( attribute [, ...] ) ]
```

Segurança

Para reforçar a segurança é interessante usar o parâmetro SECURITY DEFINER, que especifica que a função será executada somente com os privilégios do usuário que a criou.

SECURITY INVOKER indica que a função deve ser executada com os privilégios do usuário que a chamou (padrão).

Diversas Linguagens

Uma grande força do PostgreSQL é ele permitir a criação de funções pelo usuário em diversas linguagens: SQL, Plpgsql, TCL, Perl, Phyton, Ruby.

Exemplos

Para ter exemplos a disposição vamos instalar os do diretório "tutorial" dos fontes do PostgreSQL:

Acessar /usr/local/src/postgresql-8.3.1/src/tutorial e executar:

`make install`

Feito isso teremos 5 arquivos .sql.

O syscat.sql traz consultas sobre o catálogo de sistema, o que se chama de metadados (metadata).

O basic.sql e o advanced.sql são consultas SQL.

O complex.sql trata da criação de um tipo de dados pelo usuário e seu uso.

O func.sql traz algumas funções em SQL e outras em C.

O que outros SGBDs chamam de stored procedures (procedimentos armazenadas) o PostgreSQL chama de funções, que podem ser em diversas linguagens.

Exemplos simples

```
CREATE OR REPLACE FUNCTION olamundo() RETURNS int4
AS 'SELECT 1' LANGUAGE 'sql';
```

Executando

```
SELECT olamundo();
```

Passando Parâmetros

```
CREATE OR REPLACE FUNCTION add_numeros(nr1 int4, nr2 int4) RETURNS int4
AS 'SELECT $1 + $2' LANGUAGE 'sql';
```

Observe que para passar o primeiro parâmetro usa-se "\$1", o segundo "\$2" e assim por diante.

Executando

```
SELECT add_numeros(300, 700) AS resposta ;
```

Podemos passar como parâmetro o nome de uma tabela:

```
CREATE TEMP TABLE empregados (
    nome text,
    salario numeric,
    idade integer,
    baia point
);
```

```
INSERT INTO empregados VALUES('João',2200,21,point('(1,1)'));
INSERT INTO empregados VALUES('José',4200,30,point('(2,1)'));
```

```
CREATE FUNCTION dobrar_salario(empregados) RETURNS numeric AS $$ 
    SELECT $1.salario * 2 AS salario;
```

```
$$ LANGUAGE SQL;

SELECT nome, dobrar_salario(emp.*) AS sonho
  FROM empregados
 WHERE empregados.baia ~= point '(2,1)';
```

Algumas vezes é prático gerar o valor do argumento composto em tempo de execução.
Isto pode ser feito através da construção ROW.

```
SELECT nome, dobrar_salario(ROW(nome, salario*1.1, idade, baia)) AS sonho
  FROM empregados;
```

Função que retorna um tipo composto. Função que retorna uma única linha da tabela empregados:

```
CREATE FUNCTION novo_empregado() RETURNS empregados AS $$
  SELECT text 'Nenhum' AS nome,
         1000.0 AS salario,
          25 AS idade,
    point '(2,2)' AS baia;
$$ LANGUAGE SQL;
```

Ou

```
CREATE OR REPLACE FUNCTION novo_empregado() RETURNS empregados AS $$
  SELECT ROW('Nenhum', 1000.0, 25, '(2,2)')::empregados;
$$ LANGUAGE SQL;
```

Chamar assim:

```
SELECT novo_empregado();
```

ou

```
SELECT * FROM novo_empregado();
```

Funções SQL como fontes de tabelas

```
CREATE TEMP TABLE teste (testeid int, testesubid int, testename text);
INSERT INTO teste VALUES (1, 1, 'João');
INSERT INTO teste VALUES (1, 2, 'José');
INSERT INTO teste VALUES (2, 1, 'Maria');
```

```
CREATE FUNCTION getteste(int) RETURNS teste AS $$%
  SELECT * FROM teste WHERE testeid = $1;
$$ LANGUAGE SQL;
```

```
SELECT *, upper(testename) FROM getteste(1) AS t1;
```

Tabelas Temporárias - criar tabelas temporárias (TEMP), faz com que o servidor se encarregue de removê-la (o que faz logo que a conexão seja encerrada).

```
CREATE TEMP TABLE nometabela (campo tipo);
```

Funções SQL retornando conjunto

```
CREATE FUNCTION getteste(int) RETURNS SETOF teste AS $$  
    SELECT * FROM teste WHERE testeid = $1;  
$$ LANGUAGE SQL;
```

```
SELECT * FROM getteste(1) AS t1;
```

Funções SQL polimórficas

As funções SQL podem ser declaradas como recebendo e retornando os tipos polimórficos anyelement e anyarray.

```
CREATE FUNCTION constroi_matriz(anyelement, anyelement) RETURNS anyarray AS $  
$  
    SELECT ARRAY[$1, $2];  
$$ LANGUAGE SQL;
```

```
SELECT constroi_matriz(1, 2) AS intarray, constroi_matriz('a'::text, 'b') AS textarray;
```

```
CREATE FUNCTION eh_maior(anyelement, anyelement) RETURNS boolean AS $$  
    SELECT $1 > $2;  
$$ LANGUAGE SQL;  
SELECT eh_maior(1, 2);
```

<https://www.postgresql.org/docs/9.6/static/typeconv-func.html>

<https://www.postgresql.org/docs/9.6/static/functions.html>

Funções em Plpgsql

As funções em linguagens procedurais no PostgreSQL, como a Plpgsql são correspondentes ao que se chama comumente de Stored Procedures.

Por default o PostgreSQL só traz suporte às funções na linguagem SQL. Para dar suporte à funções em outras linguagens temos que efetuar procedimentos como a seguir.
Para que o banco postgres tenha suporte à linguagem de procedimento Plpgsql
executamos na linha de comando como super usuário do PostgreSQL:

createlang plpgsql –U nomeuser nomebanco

A Plpgsql é a linguagem de procedimentos armazenados mais utilizada no PostgreSQL, devido ser a mais madura e com mais recursos.

```
CREATE OR REPLACE FUNCTION func_escopo() RETURNS integer AS $$
```

```

DECLARE
    quantidade integer := 30;
BEGIN
    RAISE NOTICE 'Aqui a quantidade é %', quantidade; -- A quantidade aqui é 30
    quantidade := 50;
    --
    -- Criar um sub-bloco
    --
    DECLARE
        quantidade integer := 80;
        BEGIN
            RAISE NOTICE 'Aqui a quantidade é %', quantidade; -- A quantidade aqui é 80
        END;
        RAISE NOTICE 'Aqui a quantidade é %', quantidade; -- A quantidade aqui é 50
        RETURN quantidade;
    END;
$$ LANGUAGE plpgsql;

```

=> SELECT func_escopo();

```

CREATE FUNCTION instr(varchar, integer) RETURNS integer AS $$
DECLARE
    v_string ALIAS FOR $1;
    index ALIAS FOR $2;
BEGIN
    -- algum processamento neste ponto
END;
$$ LANGUAGE plpgsql;

```

```

CREATE FUNCTION concatenar_campos_selecionados(in_t nome_da_tabela)
RETURNS text AS $$ 
BEGIN
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE plpgsql;

```

```

CREATE FUNCTION somar_tres_valores(v1 anyelement, v2 anyelement, v3 anyelement)
RETURNS anyelement AS $$ 
DECLARE
    resultado ALIAS FOR $0;
BEGIN
    resultado := v1 + v2 + v3;
    RETURN resultado;
END;
$$ LANGUAGE plpgsql;

```

SELECT somar_tres_valores(10,20,30);

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 ... ;
```

Temos uma tabela (datas) com dois campos (data e hora) e queremos usar uma função para manipular os dados desta tabela:

```

CREATE or REPLACE FUNCTION data_ctl(opcao char, fdata date, fhora time) RETURNS
char(10) AS '
DECLARE
    opcao ALIAS FOR $1;
    vdata ALIAS FOR $2;
    vhora ALIAS FOR $3;
    retorno char(10);
BEGIN
    IF opcao = "I" THEN
        insert into datas (data, hora) values (vdata, vhora);
        retorno := "INSERT";
    END IF;
    IF opcao = "U" THEN
        update datas set data = vdata, hora = vhora where data="1995-11-01";
        retorno := "UPDATE";
    END IF;
    IF opcao = "D" THEN
        delete from datas where data = vdata;
        retorno := "DELETE";
    ELSE
        retorno := "NENHUMA";
    END IF;
    RETURN retorno;
END;
' LANGUAGE plpgsql;
//select data_ctl('I','1996-11-01', '08:15');
select data_ctl('U','1997-11-01','06:36');
select data_ctl('U','1997-11-01','06:36');
Mais Detalhes no capítulo 35 do manual oficial.

```

Funções que Retornam Conjuntos de Registros (SETS)

```
CREATE OR REPLACE FUNCTION codigo_empregado (codigo INTEGER)
```

```
RETURNS SETOF INTEGER AS '
```

```
DECLARE
```

```
    registro RECORD;
```

```
    retval INTEGER;
```

```
BEGIN
```

```

FOR registro IN SELECT * FROM empregados WHERE salario >= $1 LOOP
    RETURN NEXT registro.departamento_cod;
END LOOP;
RETURN;
END;
' language 'plpgsql';

select * from codigo_empregado (0);
select count (*), g from codigo_empregado (5000) g group by g;

```

Funções que retornam Registro

Para criar funções em plpgsql que retornem um registro, antes precisamos criar uma variável composta do tipo ROWTYPE, descrevendo o registro (tupla) de saída da função.

```

CREATE TABLE empregados(
    nome_emp text,
    salario int4,
    codigo int4 NOT NULL,
    departamento_cod int4,
    CONSTRAINT empregados_pkey PRIMARY KEY (codigo),
    CONSTRAINT empregados_departamento_cod_fkey FOREIGN KEY
(departamento_cod)
    REFERENCES departamentos (codigo) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION
)

```

```
CREATE TABLE departamentos (codigo INT primary key, nome varchar);
```

```
CREATE TYPE dept_media AS (minsal INT, maxsal INT, medsal INT);
```

```
create or replace function media_dept() returns dept_media as
```

```

'
declare
    r dept_media%rowtype;
    dept record;
    bucket int8;
    counter int;
begin
    bucket := 0;
    counter := 0;
    r.maxsal :=0;
    r.minsal :=0;
    for dept in select sum(salario) as salario, d.codigo as departamento
        from empregados e, departamentos d where e.departamento_cod = d.codigo
        group by departamento loop
        counter := counter + 1;
        bucket := bucket + dept.salario;
        if r.maxsal <= dept.salario or r.maxsal = 0 then
            r.maxsal := dept.salario;
        end if;
    end loop;
    r.medsal := bucket / counter;
    return r;
end;

```

```

end if;
if r.minsal <= dept.salario or r.minsal = 0 then
    r.minsal := dept.salario;
end if;
end loop;

r.medsal := bucket/counter;

return r;
end
' language 'plpgsql';

```

Funções que Retoram Conjunto de Registros (SETOF, Result Set)
Também requerem a criação de uma variável (tipo definidopelo user)

```
CREATE TYPE media_sal AS
(deptcod int, minsal int, maxsal int, medsal int);
```

```
CREATE OR REPLACE FUNCTION medsal() RETURNS SETOF media_sal AS
```

```

'
DECLARE
    s media_sal%ROWTYPE;
    salrec RECORD;
    bucket int;
    counter int;
BEGIN
    bucket :=0;
    counter :=0;
    s.maxsal :=0;
    s.minsal :=0;
    s.deptcod :=0;
    FOR salrec IN SELECT salario AS salario, d.codigo AS departamento
        FROM empregados e, departamentos d WHERE e.departamento_cod = d.codigo
    ORDER BY d.codigo LOOP
        IF s.deptcod = 0 THEN
            s.deptcod := salrec.departamento;
            s.minsal := salrec.salario;
            s.maxsal := salrec.salario;
            counter := counter + 1;
            bucket := bucket + salrec.salario;
        ELSE
            IF s.deptcod = salrec.departamento THEN
                IF s.maxsal <= salrec.salario THEN
                    s.maxsal := salrec.salario;
                END IF;
                IF s.minsal >= salrec.salario THEN
                    s.minsal := salrec.salario;
                END IF;
                counter := counter +1;
            ELSE

```

```

s.medsal := bucket/counter;
RETURN NEXT s;
s.deptcod := salrec.departamento;
s.minsal := salrec.salario;
s.maxsal := salrec.salario;
counter := 1;
bucket := salrec.salario;
END IF;
END IF;
END LOOP;
s.medsal := bucket/counter;
RETURN NEXT s;
RETURN;
END '
LANGUAGE 'plpgsql';

select * from medsal()

```

Relacionando:

```

select d.nome, a.minsal, a.maxsal, a.medsal
from medsal() a, departamentos d
where d.codigo = a.deptcod

```

PLPGSQL STORED PROCEDURES TUTORIAL

Delimitadores

O código de uma função plpgsql é especificado em CREATE FUNCTION como uma string literal delimitado por aspas.

1 Apóstrofo (corpo da função)

```

CREATE FUNCTION olamundo() RETURNS integer AS '
...
' LANGUAGE plpgsql;

```

Inicia após AS e termina antes de LANGUAGE

2 Apóstrofos (mascar string literal no corpo da função)

```

a_output := "Blah";
SELECT * FROM users WHERE f_name="foobar";

```

4 Apóstrofos (string constante no corpo da função)

```

a_output := a_output || " AND name LIKE ""foobar"" AND xyz"

```

Mais detalhes na documentação oficial.

Declaração de Variáveis

```
user_id integer;
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

Sintaxe geral da declaração de variáveis:

```
nome [ CONSTANT ] tipo [ NOT NULL ] [ { DEFAULT | := } expressão ];
```

Exemplos:

```
quantity integer DEFAULT 32;
url varchar := 'http://mysite.com';
user_id CONSTANT integer := 10;
```

Declarado Funções

```
CREATE FUNCTION func_escopo() RETURNS integer AS $$  
DECLARE  
    quantidade integer := 30;  
  
BEGIN  
    RAISE NOTICE 'Aqui a quantidade é %', quantidade; -- A quantidade aqui é 30  
    quantidade := 50;  
    --  
    -- Criar um sub-bloco  
    --  
    DECLARE  
        quantidade integer := 80;  
    BEGIN  
        RAISE NOTICE 'Aqui a quantidade é %', quantidade; -- A quantidade aqui é 80  
    END;  
    RAISE NOTICE 'Aqui a quantidade é %', quantidade; -- A quantidade aqui é 50  
    RETURN quantidade;  
END;  
$$ LANGUAGE plpgsql;
```

Execute

```
SELECT func_escopo() as escopo;
```

Obs.: O psql exibe as mensagens disparadas pelo RAISE.

Obs2.: BEGIN e END na plpgsql não agrupa transações, apenas grupos de comandos.

Alias para Parâmetros de Funções

Sintaxe:

```
nome ALIAS FOR $n;
```

Exemplos:

```
CREATE FUNCTION vendas_taxa(real) RETURNS real AS $$  
DECLARE  
    subtotal ALIAS FOR $1;  
BEGIN  
    RETURN subtotal * 0.06;  
END;  
$$ LANGUAGE plpgsql;
```

\$1 é um alias para o parâmetro da função que é do tipo real.

```
CREATE FUNCTION instr(varchar, integer) RETURNS integer AS $$  
DECLARE  
    v_string ALIAS FOR $1;  
    index ALIAS FOR $2;  
BEGIN  
    -- Aqui fazemos alguns cálculos usando v_string e index  
END;  
$$ LANGUAGE plpgsql;  
CREATE FUNCTION concat_campos(tab tbl_clientes) RETURNS text AS $$  
BEGIN  
    RETURN tab.nome || " " || tab.email || " " || tab.cidade || " " || tab.estado;  
END;  
$$ LANGUAGE plpgsql;
```

Também podemos (mais claro) usar os nomes dos parâmetros explicitamente:

```
CREATE FUNCTION sales_tax(subtotal real, OUT tax real) AS $$  
BEGIN  
    tax := subtotal * 0.06;  
END;  
$$ LANGUAGE plpgsql;
```

O parâmetro OUT (output, saída) é mais útil quando retorna múltiplos parâmetros:

```
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
```

Executando Funções

Usa-se o select para executar uma função, como se fosse uma view ou consulta comum.

```
select sum_n_product(5, 6);
```

Tipos Polimórficos

Quanto tipos polimórficos (anyelement e anyarray) são usados para declarar de funções, um parâmetro especial (\$0) é criado. Este tipo de dados é o atual retorno da função. É inicializado como NULL e pode ser modificado pela função.

Exemplo

Função que trabalha com qualquer tipo de dados e que suporta o operador +:

```
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;
```

Ou:

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement,
                                 OUT sum anyelement)
AS $$  
BEGIN  
    sum := v1 + v2 + v3;  
END;  
$$ LANGUAGE plpgsql;
```

Veja, que ao usar OUT não há necessidade de retorno (já está implícito).

Copiando Tipos

variavel%TYPE

%TYPE fornece o tipo de dados de uma variável ou de um campo de tabela.

Para declarar uma variável com o mesmo tipo de dado de usuarios.id_usuario deve ser escrito:

```
id_usuario usuarios.id_usuario%TYPE;
```

Tipos row

Pode armazenar um registro resultante de um SELECT ou de um FOR.

```
nome nome_da_tabela%ROWTYPE;
nome nome_do_tipo_composto;
```

Os campos podem ser acessados com nomevariavel.nomecampo;

Exemplo de uso:

```
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 ... ;
```

Tipo registro (record)

```
nome record;
```

Renomeando Variáveis

```
RENAME nome_antigo TO novo_nome;
```

```
CREATE FUNCTION logfunc1(logtxt text) RETURNS timestamp AS $$  
BEGIN  
INSERT INTO logtable VALUES (logtxt, 'now');  
RETURN 'now';  
END;  
$$ LANGUAGE plpgsql;  
e  
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;
```

Atribuições

identificador := expressão;

```
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;
```

Execução de Expressão ou Consulta sem Resultado

PERFORM create_mv('cs_session_page_requests_mv', my_query);

Não Fazer Nada

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;
```

Execução de Comandos Dinâmicos

EXECUTE cadeia_de_caracteres_do_comando;

```
EXECUTE 'UPDATE tbl SET '
|| quote_ident(nome_da_coluna)
|| '='
|| quote_literal(novo_valor)
|| ' WHERE key = '
|| quote_literal(valor_chave);
```

```
EXECUTE 'UPDATE tbl SET '
|| quote_ident(nome_da_coluna)
|| '=' $$'
|| novo_valor
|| '$$ WHERE key = '
|| quote_literal(valor_chave);
```

Estruturas de Controle

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();`

Condicionais

- IF ... THEN
- IF ... THEN ... ELSE
- IF ... THEN ... ELSE IF
- IF ... THEN ... ELSIF ... THEN ... ELSE
- IF ... THEN ... ELSEIF ... THEN ... ELSE

```
IF linha_demo.sex = 'm' THEN
    sexo_extenso := 'masculino';
ELSE
    IF linha_demo.sex = 'f' THEN
        sexo_extenso := 'feminino';
    END IF;
END IF;

IF expressão_boleana THEN
    instruções
[ ELSIF expressão_boleana THEN
    instruções
[ ELSIF expressão_boleana THEN
    instruções
...]
[ ELSE
    instruções ]
END IF;

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;
```

Laços

```
[<<rótulo>>]
LOOP
instruções
END LOOP;
```

Exit

```
EXIT [ rótulo ] [ WHEN expressão ];

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;

WHILE quantia_devida > 0 AND saldo_do_certificado_de_bonus > 0 LOOP
-- algum processamento
END LOOP;
WHILE NOT expressão_boleana LOOP
-- algum processamento
END LOOP;
```

For (laços internos)

```
[<<rótulo>>]
FOR nome IN [ REVERSE ] expressão .. expressão LOOP
instruções
END LOOP;

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;
```

Laços através do resultado de consultas

```
[<<rótulo>>]
FOR registro_ou_linha IN comando LOOP
instruções
END LOOP;

CREATE FUNCTION cs_refresh_mvviews() RETURNS integer AS $$
DECLARE
    mvviews RECORD;
BEGIN
    PERFORM cs_log('Atualização das visões materializadas...');
    FOR mvviews IN SELECT * FROM cs_materialized_views ORDER BY sort_key LOOP
        -- Agora "mvviews" possui um registro de cs_materialized_views
        PERFORM cs_log('Atualizando a visão materializada ' ||
        quote_ident(mvviews.mv_name) || ' ...');
        EXECUTE 'TRUNCATE TABLE ' || quote_ident(mvviews.mv_name);
        EXECUTE 'INSERT INTO ' || quote_ident(mvviews.mv_name) || ' ' || mvviews.mv_query;

    END LOOP;
    PERFORM cs_log('Fim da atualização das visões materializadas.');
    RETURN 1;
END;
$$ LANGUAGE plpgsql;

[<<rótulo>>]
FOR registro_ou_linha IN EXECUTE texto_da_expressão LOOP
instruções
END LOOP;
```

Capturar Erros

```
[ <<rótulo>> ]
[ DECLARE
declarações ]
BEGIN
instruções
EXCEPTION
WHEN condição [ OR condição ... ] THEN
instruções_do_tratador
[ WHEN condição [ OR condição ... ] THEN
instruções_do_tratador
... ]
END;

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;
```

Declaração de Variáveis do tipo 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 ;
```

Exemplos:

```
DECLARE
curs1 refcursor;
curs2 CURSOR FOR SELECT * FROM tenk1;
curs3 CURSOR (chave integer) IS SELECT * FROM tenk1 WHERE unicol = chave;

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('funcursor');
reffunc
-----
funcursor
(1 linha)
FETCH ALL IN funcursor;

COMMIT;
```

Erros e Mensagens

```
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`.

Procedimentos de Gatinho (Trigger)

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**

TG_ARGS[]

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.

Exemplo:

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
- 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;

```

Gatilho para registrar inserções e atualizações

```

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;

```

Gatilho para auditoria

Todas as operações na tabela emp serão registradas na tabela emp_audit

```

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;

SELECT * FROM emp_audit;

```

Autoditoria ao nível de campos

```

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;

```

```
SELECT * FROM emp_audit;
```

Gatilho 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();

```

Resumo da documentação oficial em inglês e em português do Brasil, que devem ser consultados para informações mais detalhadas.

`CREATE LANGUAGE plpgsql`

```
CREATE FUNCTION populate() RETURNS integer AS $$  
DECLARE  
    -- declarações  
BEGIN  
    PERFORM minha_funcao();  
END;  
$$ LANGUAGE plpgsql;
```

```
CREATE OR REPLACE FUNCTION funcao_teste(integer) RETURNS integer AS $$  
....  
$$ LANGUAGE plpgsql;
```

Um bloco é definido como:

```
[ <<rótulo>> ]  
[ DECLARE  
    declarações ]  
BEGIN  
    instruções  
END;
```

```
CREATE FUNCTION func_escopo() RETURNS integer AS $$  
DECLARE  
    quantidade integer := 30;  
BEGIN  
    RAISE NOTICE 'Aqui a quantidade é %', quantidade; -- A quantidade aqui é 30  
    quantidade := 50;  
    --  
    -- Criar um sub-bloco  
    --  
    DECLARE  
        quantidade integer := 80;  
    BEGIN  
        RAISE NOTICE 'Aqui a quantidade é %', quantidade; -- A quantidade é 80  
    END;  
  
    RAISE NOTICE 'Aqui a quantidade é %', quantidade; -- A quantidade aqui é 50  
  
    RETURN quantidade;  
END;  
$$ LANGUAGE plpgsql;
```

```
=> SELECT func_escopo();
```

Abaixo seguem alguns exemplos de declaração de variáveis:

```
id_usuario integer;
quantidade numeric(5);
url      varchar;
minha_linha nome_da_tabela%ROWTYPE;
meu_campo   nome_da_tabela.nome_da_coluna%TYPE;
uma_linha  RECORD;
```

A sintaxe geral para declaração de variáveis é:

```
nome [ CONSTANT ] tipo [ NOT NULL ] [ { DEFAULT | := } expressão ];
CREATE FUNCTION taxa_de_venda(subtotal real) RETURNS real AS $$  
BEGIN  
    RETURN subtotal * 0.06;  
END;  
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION taxa_de_venda(real) RETURNS real AS $$  
DECLARE  
    subtotal ALIAS FOR $1;  
BEGIN  
    RETURN subtotal * 0.06;  
END;  
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION instr(varchar, integer) RETURNS integer AS $$  
DECLARE  
    v_string ALIAS FOR $1;  
    index    ALIAS FOR $2;  
BEGIN  
    -- algum processamento neste ponto  
END;  
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION concatenar_campos_selecionados(in_t nome_da_tabela)
RETURNS text AS $$  
BEGIN  
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;  
END;  
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION somar_tres_valores(v1 anyelement, v2 anyelement, v3 anyelement)
```

```

RETURNS anyelement AS $$

DECLARE
    resultado ALIAS FOR $0;
BEGIN
    resultado := v1 + v2 + v3;
    RETURN resultado;
END;
$$ LANGUAGE plpgsql;

```

```

SELECT somar_tres_valores(10,20,30);
SELECT somar_tres_valores(1.1,2.2,3.3);

```

```

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 logfunc1(logtxt text) RETURNS timestamp AS $$

BEGIN
    INSERT INTO logtable VALUES (logtxt, 'now');
    RETURN 'now';
END;
$$ LANGUAGE plpgsql;

```

e

```

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;

```

```

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;

```

```

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;
```

```
PERFORM create_mv('cs_session_page_requests_mv', my_query);
```

```
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;
```

```
EXECUTE 'UPDATE tbl SET '
|| quote_ident(nome_da_coluna)
|| ' = '
|| quote_literal(novo_valor)
|| ' WHERE key = '
|| quote_literal(valor_chave);
```

```
EXECUTE 'UPDATE tbl SET '
|| quote_ident(nome_da_coluna)
|| ' = $$'
|| novo_valor
|| '$$ WHERE key = '
|| quote_literal(valor_chave);
GET DIAGNOSTICS variavel_inteira = ROW_COUNT;
```

```
IF v_id_usuario <> 0 THEN
  UPDATE usuarios SET email = v_email WHERE id_usuario = v_id_usuario;
END IF;
```

```
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 linha_demo.sexo = 'm' THEN
    sexo_extenso := 'masculino';
ELSE
    IF linha_demo.sexo = 'f' THEN
        sexo_extenso := 'feminino';
    END IF;
END IF;
```

IF-THEN-ELSIF-ELSE

```

IF expressão_booleana THEN
    instruções
[ ELSIF expressão_booleana THEN
    instruções
[ ELSIF expressão_booleana THEN
    instruções
    ...]]
[ ELSE
    instruções ]
END IF;
```

```

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;
```

```

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;

```

[<<rótulo>>]

```

WHILE expressão LOOP
    instruções
END LOOP;

```

```

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;

```

[<<rótulo>>]

```

FOR nome IN [ REVERSE ] expressão .. expressão LOOP
    instruções
END LOOP;

```

```

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;

```

[<<rótulo>>]

```

FOR registro_ou_linha IN comando LOOP
    instruções
END LOOP;

```

```

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;

```

[<<rótulo>>]
FOR registro_ou_linha IN EXECUTE texto_da_expressão LOOP
 instruções
END LOOP;

Captura de erros

[<<rótulo>>]
[DECLARE
 declarações]
BEGIN
 instruções
EXCEPTION
 WHEN condição [OR condição ...] THEN
 instruções_do_tratador
 [WHEN condição [OR condição ...] THEN
 instruções_do_tratador
 ...]
END;

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;
```

Cursores

nome CURSOR [(argumentos)] FOR comando ;

DECLARE

```

curs1 refcursor;
curs2 CURSOR FOR SELECT * FROM tenk1;
curs3 CURSOR (chave integer) IS SELECT * FROM tenk1 WHERE unico1 = chave;
```

OPEN curs1 FOR SELECT * FROM foo WHERE chave = minha_chave;

OPEN curs1 FOR EXECUTE 'SELECT * FROM ' || quote_ident(\$1);

OPEN curs2;

OPEN curs3(42);

```

FETCH curs1 INTO variável_linha;
FETCH curs2 INTO foo, bar, baz;
```

CLOSE curs1;

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('funcursor');

reffunc

funcursor
(1 linha)

FETCH ALL IN funcursor;

col

123
(1 linha)

COMMIT;

```
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;
```

```
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

```
RAISE NOTICE 'Chamando cs_create_job(%)', v_job_id;
```

```
RAISE EXCEPTION 'ID inexistente --> %', id_usuario;
```

Gatilhos escritos em PL/pgSQL

```
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;
```

```
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;

```

```
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;
```

```
SELECT * FROM emp_audit;
```

```

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;
```

```
SELECT * FROM emp_audit;
```

```
-- 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();

```

```

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();
```

```
=> 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;
```

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 parâmetros 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(anyelement, anyelement) 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(anyelement, anyelement) RETURNS boolean AS $$ 
    SELECT $1 > $2;
$$ LANGUAGE SQL;

SELECT is_greater(1, 2);

CREATE FUNCTION dup (f1 anyelement, OUT f2 anyelement, 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 \'escorpio\' 
        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 portável 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;
arow 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_taxsubtotal mas no segundo caso não pode.

```
CREATE FUNCTION instr(varchar, 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 portável).

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_funct(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;
```

```
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. As 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 usuário 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 apóstrofos. 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 apóstrofos ou aspas. Um exemplo (**assumindo que está sendo utilizada a delimitação por cifrão para a função como um todo e, portanto, os apóstrofos 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 apóstrofos, 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
variável_inteira integer;
begin
GET DIAGNOSTICS variável_inteira = ROW_COUNT;
return variável_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 muita 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.sex = 'm' THEN
    sexo_extenso := 'masculino';
ELSE
    IF linha_demo.sex = '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
[ ELSIF expressão_booleana THEN
    instruções
[ ELSIF 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-ELSIF-ELSE

ELSIF 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;  
    startspos   ALIAS FOR $3;
```

```

eendpos      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 calcsum(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';
```

```

        END IF;
    END IF;
    RETURN result;
END;
' LANGUAGE 'plpgsql';

```

Cursos

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 cursos 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 unico1 = 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 quer 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('funcursor');

      reffunc
-----
      funcursor
(1 linha)

FETCH ALL IN funcursor;

      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 apóstrofos

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 apóstrofos ('), então os apóstrofos dentro do corpo da função devem ser duplicados; da mesma maneira, as contrabarras dentro do corpo da função (\) devem ser duplicadas. Duplicar os apóstrofos 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 apóstrofos 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 apóstrofos. Na abordagem delimitada por cifrão os apóstrofos 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 apóstrofos 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 apóstrofos (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 apóstrofos *devem* aparecer em pares.

2 apóstrofos

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 apóstrofos

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 apóstrofos

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 apóstrofos

Quando é necessário colocar dois apóstrofos em uma constante cadeia de caracteres (que necessita de 8 apóstrofos), e estes dois apóstrofos estão adjacentes ao final da constante cadeia de caracteres (mais 2 apóstrofos). 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 apóstrofos 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; -- Dígito 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, não 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;

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 pgsql nomedatabase

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 ( ISCACHABLE );
```

-- 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;
            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 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 periodo 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[]);
```

Osvaldo

```

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 alguem tiver algum link para indicar também ajuda.

Rúben Lício Reis

```

CREATE FUNCTION r(c1 out text,c2 out text)
LANGUAGE plpgsql;
AS $c$
BEGIN

```

```

SELECT 'teste1' AS foo,'teste2' AS bar
    INTO $1,$2;
END;
$c$;
SELECT * FROM r();

```

Leo

Validação de CPF com PI/ PgSQL

A partir de hoje passamos a divulgar algoritmos de funções e consultas que sejam de utilidade pública. A validação de CPF com PI/ PgSQL 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>

32 - Gatilhos

- 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 cascatapear gatilhos. Não existe limitação direta do número de níveis de cascataemento. É possível que o cascataemento 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 |
data_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;
END;

```

```

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 compromiso 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 Kauí 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: KAUÍ 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 ENTAO 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 escrita 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 - KAUAIRES@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: KAUAI 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: KAUAI 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 FUNÇAO
DECLARE
  -- PARAMETROS DE VERIFICAÇÃO E VARIAVEIS
  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;

```

\$empregado_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			
kaui_aires	24/10/2006 15:34	public.empregado	INSERT	172.25.0.200
kaui_aires	24/10/2006 15:34	public.empregado	INSERT	172.25.0.200
kaui_aires	24/10/2006 15:34	public.empregado	INSERT	172.25.0.200
kaui_aires	24/10/2006 15:34	public.empregado	INSERT	172.25.0.200
kaui_aires	24/10/2006 15:34	public.empregado	UPDATE	172.25.0.200
kaui_aires	24/10/2006 15:34	public.empregado	DELETE	172.25.0.200

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.

33 - 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 camposl_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
    );
```

Vejamos 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';
```

Vejamos novamente o que ocorreu na tabela shoelace_log:

```
SELECT * FROM shoelace_log;
```

sl_name	sl_avail	log_who	log_when
sl7	6	A1	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.

Derrubar usuário Conectado no PostgreSQL No Linux e no Windows

No Linux

Para retornar o PID dos usuários e suas respectivas consultas execute a consulta:

```
SELECT pg_stat_get_backend_pid(s.backendid) AS pid,
       pg_stat_get_backend_activity(s.backendid) AS current_query
  FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

Cuidado, evite usar o comando "kill -9 PID" para derrubar um usuário do PostgreSQL.

"-Nunca- faça isso. Se você executar um 'kill -9 XXXX' você irá derrubar a conexão problemática e todas as outras ativas no momento. Isso porque todas as conexões compartilham uma área de memória comum (aka shared buffers) e, se você a corrompe o PostgreSQL tem que garantir que os dados ali estejam consistentes.

Se você quer cancelar uma consulta demorada utilize pg_cancel_backend()

[1]. Existia uma função pg_terminate_backend() (não me recordo se ele chegou a estar em alguma versão do 8.0) mas ela se tornou obsoleta por motivos de confiabilidade. Esta mesma função utiliza 'kill -TERM XXXX' para derrubar os processos do postgres. Esta seria a maneira "correta"

de terminar os processos postgres mas ...

O que aconselho fazer é:

- (i) pg_cancel_backend(pid)
- (ii) kill -TERM pid

ou

- (i) pg_cancel_backend(pid)
- (ii) pg_ctl kill TERM pid

[1] <http://www.postgresql.org/docs/8.3/static/functions-admin.html>"

Dica do Euler na lista pgbr-geral

No Windows:

Baixar o utilitário kill for windows de <http://www.mattkruse.com/utilities/kill-tlist.zip>
Descompactar e copiar o KILL.EXE para c:\windows

Teclar Ctrl+Alt+Del
Clicar em Exibir
Selecionar Colunas
Selecionar PID (Identificador de processos)

Abrir um terminal e executar o psql como superusuário:
psql -U postgres

Abrir um segundo terminal com um usuário comun, que tenha privilégio de login:

```
psql -U dba1 clientes
```

```
SELECT pg_stat_get_backend_pid(s.backendid) AS pid,
       pg_stat_get_backend_activity(s.backendid) AS current_query
  FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

Anotar o pid.

No primeiro terminal, como superusuário, sem sair do psql, executar:

```
\!kill -f pid
```

Derrubará o usuário dba1 somente.

Triggers (Gatilhos)

Capítulo 32 do manual oficial. e:

<http://pgdocptbr.sourceforge.net/pg80/sql-createtrigger.html>

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';
```

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
    Ribamar FS – http://ribafs.net – ribafs@ribafs.net
    DBA PostgreSQL           89/90
);
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
    );

```

Vejamos 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';
```

Vejamos 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.

34 - Exemplos de Funções em SQL no PostgreSQL

As funções em SQL somente podem utilizar comandos do SQL.

```
CREATE FUNCTION limpar_emp() RETURNS void AS '
    DELETE FROM empregados
        WHERE salario < 0;
' LANGUAGE SQL;
```

Testando:

```
SELECT limpar_emp();
```

Passando parâmetros:

```
CREATE FUNCTION debitar (integer, numeric) RETURNS integer AS $$ 
    UPDATE banco
        SET saldo = saldo - $2
        WHERE conta = $1;
    SELECT 1;
$$ LANGUAGE SQL;
```

```
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;
```

Obs.: Os parâmetros são tratados no corpo da função como \$1, \$2, ...

Retornando tipo composto:

```
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;
```

```
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
```

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;
```

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';
```

Exemplos simples e práticos de uso de funções em plpgsql no PostgreSQL

Com o uso das funções plpgsql (stored procedures) podemos impedir que os programadores tenham acesso direto às tabelas, passando a usar somente o que programamos nas funções.

As operações do CRUD serão feitas através das stored procedures.

O delimitador \$\$ é recomendado.

Exemplo simples:

```
CREATE or replace FUNCTION f_ola_mundo() RETURNS varchar AS  
$$  
declare  
    ola text;  
begin  
    ola :='Olá mundo!';  
    perform ola;  
    return ola;  
end;  
$$  
language 'plpgsql';
```

```
select f_ola_mundo();
```

Outro:

```
create function texto(texto1 text, texto2 text) returns char as
$$
declare
    resultado text;
begin
    resultado := texto1 || texto2; -- || é o caracter para concatenação (como no SQL).
    return resultado;
end;
$$ language 'plpgsql';
```

Recebe 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';

select calc_comprim('DNOCS');
```

Exemplo com escopo da função:

```
CREATE or replace 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;
```

```
select f_escopo();
```

Alguns exemplos de declaração de variáveis:

```
user_id integer;
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

Atribuição de variáveis (seção Begin):

```
quantity = 32;
url := 'http://mysite.com';
user_id := 10;
```

Alias para parâmetros de funções

Na criação da função (recomendado):

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$  
BEGIN  
    RETURN subtotal * 0.06;  
END;  
$$ LANGUAGE PLpgsql;
```

Função que retorna um 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 ... ;
```

Outros exemplos:

```
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';

```

Usando FOR

```

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;

```

```
CREATE FUNCTION countc(text, text, int4, int4) RETURNS int4 AS '
```

```

DECLARE
    intext    ALIAS FOR $1;
    inchar    ALIAS FOR $2;
    startpos  ALIAS FOR $3;
    eendpos   ALIAS FOR $4;
    tmp       text;
    i         int4;
    len      int4;
    result    int4;
BEGIN
    result = 0;
    len := LENGTH(intext);
    FOR i IN startpos..eendpos LOOP
        tmp := substr(intext, i, 1);

```

```

    IF tmp = inchar THEN
        result := result + 1;
    END IF;
END LOOP;
RETURN result;
END;
' LANGUAGE 'plpgsql';

```

Sem FOR, com loop/next:

```

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';

```

Captura de erros:

```

CREATE FUNCTION calcsum(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';

```

Usando Cursos:

```

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;

```

Outro

```

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;

```

Retorno de Erros:

```

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);

```

Função que remove 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;

```

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 periodo 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;

```

Retornar vários campos:

```

CREATE FUNCTION r(c1 out text,c2 out text)
LANGUAGE plpgsql;
AS $c$
BEGIN
SELECT 'teste1' AS foo,'teste2' AS bar

```

```

INTO $1,$2;
END;
$c$;
```

```

SELECT * FROM r();
--Leo na lista pg-geral
```

Função 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;
    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';

```

Retorno:

inválido -3
válido 1

Passando 2 no primeiro campo não validará o documento.

```

SELECT f_cnpjcpf( 1, '12312312345' ); -- retorna -3 (inválido)
SELECT f_cnpjcpf( 2, '12312312345' ); -- retorna 1 (válido), devido ao valor 2

```

```

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)
);

```

INSERT INTO cadastro (nome, tipopessoa, cpfcnpj)

```
VALUES ( 'Juliano S. Ignacio', 1, '12106836368');
```

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.

Este exemplo acima é do Juliano num dos seus artigos do iMasters.com.br.

Usando DATAs:

```
-- Função que retorna último dia do mês informado
CREATE OR REPLACE FUNCTION sp_obtem_ultimo_dia(Mes INTEGER, Ano
INTEGER)
RETURNS INTEGER AS $$

DECLARE
    UltimoDiaMes INTEGER;
BEGIN
    SELECT EXTRACT(DAY FROM ((Ano||'/'||(Mes + 1)||'/01'):: DATE - 1))
    INTO UltimoDiaMes;

    RETURN UltimoDiaMes;
END; $$

LANGUAGE 'plpgsql';
-- Autor: Jackson na pgbr-geral
```

```
select sp_obtem_ultimo_dia(8,2008);
```

```
-- Retorna o último dia do mês da data informada
create or replace function last_day(date) returns date as 'select
cast(date_trunc("month", $1) + "1 month"::interval as date) - 1'
language sql;
```

```
select last_day('03-09-2008')
```

Função (Macro para PGAdmin)

```
-- Macro: descreve tabelas com seus campos, tipos, constraints, chaves, índices,etc
-- Chamar com: select sp_xyz_desc_tabela_view('nome_tabela')
-- Function: sp_xyz_desc_tabela_view(character varying)

-- DROP FUNCTION sp_xyz_desc_tabela_view(character varying);
```

```

CREATE OR REPLACE FUNCTION sp_xyz_desc_tabela_view(character varying)
RETURNS void AS
$BODY$DECLARE
prTabela      ALIAS FOR $1;
rec           RECORD;
sSQL          TEXT;
s TEXT;
BEGIN
sSQL := 'SELECT b.attname AS campo, c.typname AS tipo,(CASE b.attlen
When -1 then (b.atttypmod - 4) else b.attlen end)::varchar as digitos, (CASE b.attnotnull
When ''|| quote_literal('t') || '' Then '' ||
quote_literal('S') || '' Else ''|| quote_literal('N') || '' END) AS nulo,
(CASE b.atthasdef When ''|| quote_literal('t') || '' Then ''|| quote_literal('S') || '' Else ''||
quote_literal('N') || '' END)
AS default
FROM pg_class a
JOIN pg_attribute b ON (b.attrelid = a.relfilenode)
JOIN pg_type c ON (c.typelem = b.atttypid AND c.typlen = -1)
WHERE b.attstattarget = -1 AND a.relname = '' ||
quote_literal(prTabela) || '
ORDER BY b.attnum';
s := '
*** ATRIBUTOS
Nome' || lpad(' ', 26) || 'Campo' || lpad(' ', 10) || 'TAMANHO NULO DEFAULT';
FOR rec IN EXECUTE sSQL LOOP
s := s || '
' || rec.campo || lpad(' ', 30 - length(rec.campo))
|| rec.tipo || lpad(' ', 15 -length(rec.tipo))
|| rec.digitos || lpad(' ', 10 -length(rec.digitos))
|| rec.nulo || lpad(' ', 6 -
length(rec.nulo))
|| rec.default || lpad(' ', 3 -length(rec.default));
END LOOP;
--raise notice '%',s;
s := s || '

*** CHAVES PRIMARIAS';
sSQL := 'SELECT b.relname FROM pg_catalog.pg_index a LEFT JOIN pg_class b ON
(b.relfilenode = a.indexrelid)
LEFT JOIN pg_class c ON (a.indrelid =c.relfilenode)
WHERE c.relname = ''||quote_literal(prTabela) || '' AND a.indisprimary = '
|| quote_literal('t');
FOR rec IN EXECUTE sSQL LOOP
s := s || '
' || rec.relname;
END LOOP;

s := s || '

```

```

*** RESTRIÇÕES DE UNICIDADE';
sSQL := 'SELECT b.relname FROM pg_catalog.pg_index a LEFT JOIN pg_class b ON
(b.relfilenode =a.indexrelid)
    LEFT JOIN pg_class c ON (a.indrelid =c.relfilenode)
    WHERE c.relname = '||quote_literal(prTabela) || ' AND a.indisprimary = '
||quote_literal('f') || '
        AND a.indisunique = '|| quote_literal('t');
FOR rec IN EXECUTE sSQL LOOP
    s := s || '
' || rec.relname;
END LOOP;

s := s || '

*** INDICES';
sSQL := 'SELECT b.relname FROM pg_catalog.pg_index a LEFT JOIN pg_class b ON
(b.relfilenode =a.indexrelid)
    LEFT JOIN pg_class c ON (a.indrelid =c.relfilenode)
    WHERE c.relname = '||quote_literal(prTabela) || ' AND a.indisprimary = '
||quote_literal('f') || '
        AND a.indisunique = '|| quote_literal('f');
FOR rec IN EXECUTE sSQL LOOP
    s := s || '
' || rec.relname;
END LOOP;

s := s || '

*** RESTRIÇÕES';
sSQL := 'SELECT a.conname, a.consrc
        FROM pg_catalog.pg_constraint a LEFT JOIN pg_class b ON (a.conrelid
= b.relfilenode)
        WHERE contype = '|| quote_literal('c')
|| ' AND b.relname = '|| quote_literal(prTabela);
FOR rec IN EXECUTE sSQL LOOP
    s := s || '
' || rec.conname || lpad(' ', 30 -length(rec.conname)) || rec.consrc;
END LOOP;

s := s || '

*** RELACIONAMENTOS';
sSQL := 'SELECT a.conname
        FROM pg_catalog.pg_constraint a LEFT JOIN pg_class b ON (a.conrelid
= b.relfilenode)
        WHERE contype = '|| quote_literal('f') || ' AND b.relname = '|| quote_literal(prTabela);
FOR rec IN EXECUTE sSQL LOOP
    s := s || '
'

```

```

' || rec.conname;
END LOOP;

raise notice '%

.',s;

sSQL := 'DROP FUNCTION sp_xyz_desc_tabela_view(varchar)';
EXECUTE sSQL;
RAISE EXCEPTION '';
END; $BODY$
LANGUAGE 'plpgsql' VOLATILE;

```

Exemplos práticos e simples de uso de Triggers no PostgreSQL

Exemplo de Criação e uso de Trigger

Criando as tabelas de exemplo:

```

create table clientes(cpf char(11), nome char(45), telefone char(10));
create table t_clientes(cpf char(11), nome char(45), telefone char(10));
insert into clientes values('11111111111', 'João Brito', '32343235');
insert into clientes values('22222222222', 'Pedro Brito', '34343235');

```

Criando a função de gatilho:

```

create or replace function gera_log() returns trigger as
$$
declare
    r record;
begin
    insert into t_clientes values('2222', 'Trigger', '234323');
    return r;
end;
$$
language plpgsql;

```

Trigger da tabela clientes:

```

create trigger t_clientes after delete on clientes
for each row execute procedure gera_log();

```

```
delete from clientes where cpf='22222222222';
```

Outro Exemplo:

<http://www.postgresql.org/docs/8.2/static/plpgsql.html>

```

CREATE FUNCTION prc_valida_estoque(integer, integer) RETURNS INTEGER AS $$ 
DECLARE
    PCOD_MEDIC ALIAS $1;
    PQTDE_MEDIC ALIAS FOR $2;
    vestoque INTEGER;
BEGIN
    select into vestoque estoque
    from medicamentos
    where cd_medicamento = :pcod_medic;

    -- PL/pgSQL nao tem excecoes, entao faremos a funcao retornar -1
    -- em caso de falha e 1 em caso de sucesso

    if (:pqtde_medic > :vestoque) then
        RETURN -1;

        RETURN 1;
    END;
$$ LANGUAGE plpgsql;

```

Triggers tem que ser implementados como funcoes especiais sem argumentos que retornam o tipo TRIGGER.

<http://www.postgresql.org/docs/8.2/static/plpgsql-trigger.html>

```

CREATE FUNCTION deletar_item_ordem_saida() RETURNS TRIGGER AS $$ 
BEGIN
    -- Tens certeza que a WHERE clause ta' certa?
    UPDATE medicamentos SET estoque = estoque + OLD.qtde_medic
    WHERE cd_medicamento = OLD.qtde_medic;

    -- Retornamos NULL por que e' um AFTER trigger
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;

```

```

CREATE TRIGGER deletar_item_ordem_saida
    AFTER DELETE ON item_ordem_saida
    FOR EACH ROW EXECUTE PROCEDURE deletar_item_ordem_saida();

```

-Roberto

Outro:

```

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);
```

```
INSERT INTO empregados (codigo,nome, salario) VALUES (5,'Pedro',NULL);
```

```
INSERT INTO empregados (codigo,nome, salario) VALUES (5,'Pedro',-1000);
```

Outro Exemplo (Auditoria):

```
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
);

```

NEW – Para INSERT e UPDATE

OLD – Para DELETE

```

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();

```

SELECT * FROM empregados_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;
```

Exemplos simples e práticos do uso do sistema de Rules no PostgreSQL

Digamos que se deseja acompanhar as alterações na coluna cad_sap_num_par_disp da relação tbl_cadarço. Para essa finalidade é criada uma tabela de acompanhamento, e uma regra que escreve sob condição uma entrada de acompanhamento quando é executada uma atualização na tabela tbl_cadarço.

```
CREATE TABLE tbl_cadarço (
    cad_sap_nome      text,    -- nome do catarço do sapato - chave primária
    cad_sap_num_par_disp integer, -- número de pares disponíveis
    cad_sap_cor        text,    -- cor do catarço
    cad_sap_comp       real,   -- comprimento do catarço
    cad_sap_unid       text     -- unidade de comprimento
);
```

```
CREATE TABLE tbl_cadarço_log (
    cad_sap_nome      text,    -- nome do catarço do sapato
    cad_sap_num_par_disp integer, -- novo valor disponível
    log_quem          text,    -- quem fez isto
    log_quando         timestamp -- quando
);
```

```
INSERT INTO tbl_cadarço VALUES ('cad1', 5, 'preto', 80.0, 'cm');
INSERT INTO tbl_cadarço VALUES ('cad2', 6, 'preto', 100.0, 'cm');
INSERT INTO tbl_cadarço VALUES ('cad3', 0, 'preto', 35.0, 'inch');
INSERT INTO tbl_cadarço VALUES ('cad4', 8, 'preto', 40.0, 'inch');
INSERT INTO tbl_cadarço VALUES ('cad5', 4, 'marrom', 1.0, 'm');
INSERT INTO tbl_cadarço VALUES ('cad6', 0, 'marrom', 0.9, 'm');
INSERT INTO tbl_cadarço VALUES ('cad7', 7, 'marrom', 60, 'cm');
INSERT INTO tbl_cadarço VALUES ('cad8', 1, 'marrom', 40, 'inch');
```

```
CREATE RULE reg_cadarço_upd AS ON UPDATE TO tbl_cadarço
    WHERE NEW.cad_sap_num_par_disp <> OLD.cad_sap_num_par_disp
    DO INSERT INTO tbl_cadarço_log VALUES (
        NEW.cad_sap_nome,
        NEW.cad_sap_num_par_disp,
        current_user,
        current_timestamp
    );
```

Testando:

```
SELECT * FROM tbl_cadarço_log;
```

```
UPDATE tbl_cadarço SET cad_sap_num_par_disp = 6 WHERE cad_sap_nome = 'cad7';
```

e olhada a tabela de acompanhamento, será encontrado:

```
SELECT * FROM tbl_cadarço_log;
```

cad_sap_nome	cad_sap_num_par_disp	log_quem	log_quando
cad7	6	teste	2005-12-03 07:45:28.500131

(1 linha)

```
create or replace function fatorial(int) returns int as
```

```
' declare
    a int;
    ret int;
begin
    a := $1 - 1;

    if a = 0 then
        ret := $1;
    else
        ret := $1 * fatorial(a);
    end if;

    return ret;
end; '
language 'plpgsql';
```

Funções na Linguagem SQL

O PostgreSQL tem 4 tipos de funções:

- Nativas
- Linguagem SQL
- Procedurais
- Em C

Sintaxe de funções SQL

```
create function nome ([parametro1, parametro2, ...])
returns retorno AS'
    corpo da função
' LANGUAGE SQL;
```

Exemplo usando aspas simples:

```
create function soma_um(integer)
returns integer as'
    select $1 + 1;
'language sql;
```

Usando

```
select soma_um(5);
```

Exemplo usando \$\$

```
create function soma_valores(integer, integer)
returns integer as $$ 
    select $1 + $2;
$$
language sql;
```

```
select soma_valores(3,8);
```

Outros exemplos

Trigger/Gatilho

São procedimentos armazenados que são disparados por eventos do banco.

A implementação de uma trigger pode ser em plpgsql.

A função de uma trigger deve ter um retorno do tipo trigger.

O PostgreSQL provém uma função nativa para triggers/gatilhos:

`suppress_redundant_update_trigger` - esta função deve prevenir qualquer atualização que não mude nada no registro ao contrário do comportamento normal que sempre executa atualização mesmo que não tenha alterações.

Idealmente devemos evitar atualizações que não alterem dados.

Exemplo de uso:

```
create trigger z-min-update
before update on tabela
for each row execute procedure
suppress_redundant_updates_trigger();
```

Função de Eventos de Trigger

O PostgreSQL tem uma função nativa de eventos de trigger, que é a:

`pg_event_trigger_dropped_objects`

Mostra uma lista de objetos excluídos.

Modelagem de Bancos de Dados

- 1. Modelo Relacional**
- 2. Normalização**
- 3. Integridade Referencial**
- 4. Expressões Regulares**
- 5. Uso de tipos personalizados e domínios**
- 6. Ferramentas úteis**
- 7. Modelos de Dados**

Modelo Relacional (MR)

- Consiste de uma coleção de relações (tabelas no modelo físico), cada uma com um nome único (por esquema).
- Cada relação é composta por atributos (campos, no modelo físico).
- Cada atributo tem seu tipo ou domínio.
- Temos também as constraints (restrições). Relacionamentos são formados por constraints. No caso chamado de integridade referencial.
- Uma relação é formada por um conjunto de tuplas (registros no modelo físico).
- No modelo relacional uma relação tanto representa os dados quanto as relações entre eles.

NULOS - acarretam sérias dificuldades e devem ser evitados.

Chaves:

- **primária** – Uma chave composta por um ou mais campos e que não repete em nenhum registro. Formata internamente pela constraint UNIQUE e a NOT NULL.
- **estrangeira** - Uma Chave Estrangeira é uma Chave de Relacionamento ou seja ela tem como propósito representar os Relacionamentos entre Tabelas num BDR.

A chave estrangeira de uma tabela pode ser definida como um conjunto de atributos da tabela (um ou mais atributos) com a propriedade de estabelecer relacionamento entre linhas de tabelas.

Uma chave estrangeira corresponde sempre a uma chave primária previamente definida em alguma tabela.

Em outras palavras um valor de chave estrangeira sempre aponta para um valor de chave primária previamente existente no banco de dados. Essa restrição é denominada de “Restrição de Integridade Referencial”.

Uma chave estrangeira pode conter valor nulo.

Chave Estrangeira é o mecanismo através do qual o Modelo de Dados Relacional

implementa relacionamentos entre tabelas. Não constitui um atributo da entidade. Só aparece no momento do projeto físico.

- **candidata** – Chave criada com a adição da constraint UNIQUE em um campo.
- **super** – uma chave que contém mais campos que o necessário para garantir que seja PK.
- **artificial** – uma chave formada por um campo que nada representa para a tabela, como um ID, registro, código, etc.
- **natural** – uma chave formada por um ou mais campos que de fato representam todos os registros de uma tabela.

Uma chave é uma propriedade do conjunto de entidades e não de uma entidade específica.

Um documento com boas informações sobre modelagem e normalização:

Modelagem e Administração de Dados em PostgreSQL

Fundamentos e práticas em bases de dados livres

De Leandro Guimarães Faria Corcete DUTRA (para a Conferência PostgreSQL Brasil)

Em: <http://leandro.gfc.dutra.googlepages.com/adpg.art.pdf>

Modelo Relacional segundo C. J. Date

- Um modelo de dados é uma definição abstrata, lógica, dos objetos, operadores (funções) e demais.
- Objetos são usados para modelar a estrutura de dados
- Operadores são usados para modelar o comportamento dos dados
- A implementação de um modelo é a realização física na máquina real.
- Diagramas E/R não representam tudo que interessa dos relacionamentos entre as entidades.
- Uma relação que contém nulos não é uma relação.
- Um modelo relacional com nulos não é um modelo relacional.
- Nulos são um erro e nunca deveriam ter sido adotados.
- Toda função é um operador mas nem todo operador é uma função.
- Relações - Atributos - Tipos

Correspondência de termos entre os modelos lógico e físico:

- Relações - Tabelas
- Tupla - Registros
- Atributos - Campos ou colunas
- Modelo - Implementação

As quatro Operações das Relações:

- Relações são normalizadas
- Atributos são desordenados, da esquerda para a direita
- Tuplas são desordenadas, de cima para baixo
- Não existem tuplas duplicadas

Dicionário de dados - dados sobre dados - metadados (catálogo do sistema no PostgreSQL).

Esta última subentende chave natural em todas as tabelas. Mas se os SGBDs implementarem assim, como vamos vazer nossos testes e demonstrações? (nota)

- Valores de atributos são valores simples, mas esses valores podem ser absolutamente qualquer coisa. Nós rejeitamos a antiga noção de "valor atômico".

Ou seja, justificando tipos como geométricos, arrays, etc. (nota)

- A cardinalidade de um conjunto é seu número de elementos e de uma relação é seu número de tuplas.

- O projeto de um banco de dados tem mais de arte que de ciência.

DER

Cuidado com diagramas, use-os apenas para ajudar no processo do projeto, lembrando que um DER não faz todo o trabalho de modelagem e nem representa tudo de um modelo.

(CJ Date em seu livro: An Introduction To Database Systems)

Normalizando Tabelas com Leandro Dutra

- Atributos Multivvalorados (telefones, municipio, uf, etc) indicam necessidade de criação de outra tabela.
- A repetição de valores de atributos também deve ser evitada criando-se uma outra tabela.
- A presença de nulos também é resolvida com a normalização e consequente criação de outras tabelas.
- O modelo relacional foi fundamentado na teoria dos conjuntos e na lógica dos predicados.

Se fosse buscar por inspiração no sentido mais exato, diria que era na crise de software: facilitar o desenvolvimento de grandes bases de dados usadas simultaneamente por muitos usuários.

- Seus termos principais são: relações, atributos, restrições e tipos!
- Relacionamento não é um termo técnico deste modelo, mas do MER, aqui temos as restrições de integridade referencial.
- A linguagem SQL não é inteiramente relacional. SGBD relacionais restringem o modelo para usarem essa linguagem.

Por isso mesmo não os chamo de SGBDRs, mas de SGBDs SQL. Os SGBDRs que conheço são o IBM BS/12, o Ingres QUEL original, o Alphora Dataphor e outros atualmente em desenvolvimento, anteriormente listados.

- É bom distinguir MR (modelo relacional) de MER (modelo entidade relacionamento). Este último surgiu depois do relacional, mas a grande maioria dos SGBDs atuais implementam o modelo relacional ou uma versão adaptada dele.

Em princípio, todas as chaves naturais são boas, e todas as artificiais são ruins.

O caso é que tem muita gente, principalmente usuários de ORMs, que não gosta de chaves compostas. Acha que fica difícil programar. Eu nunca usei ORM, nunca senti na pele, e daí vem minha impressão de que muitos ORMs criam tantos problemas quanto resolvem.

Imagine uma relação pai com uma chave natural composta razoável, digamos mais de três atributos; e uma relação filha com muito mais tuplas, digamos milhões ou dezenas de milhões a mais, isso rodando num sistema que é gargalo de desempenho. Nesse caso, pode ser que, após testes, valha a pena uma chave primária artificial para emagrecer a tabela filha.

(Trechos de respostas na lista pgbr-geral, por Leandro Dutra)

Algumas Demonstrações sobre Normalização

Valor default

Chaves naturais x artificiais

Null

Default

```
create table nula(c1 serial primary key, c2 int, c3 int default 0);
insert into nula (c1) values (default),(default),(default),(default);
select * from nula;
c1 | c2 | c3
----+----+----
```

```
1 || 0
2 || 0
3 || 0
4 || 0
(4 registros)
```

Veja só que “riqueza” de registros! Tudo isso graças a permissão de nulo e ao valor default.

Nulo

```
create table nula2(c1 int primary key, c2 int check(c2 > 0), c3 int);
insert into nula2(c1,c2,c3) values (1,default,4); -- Será válido. Importante: use not null
insert into nula2(c1,c2,c3) values (2,-3,4)
select * from nula2;
c1 | c2 | c3
----+----+
1 | 4
(1 registro)
```

Uma "incoerência" no comportamento do nulo, que reforça a recomendação de se evitar seu uso.

Chave artificial

```
create table artificial(c1 serial primary key, t1 text, t2 text);
insert into artificial(t1,t2) values ('a','b'),('a','b'),('a','b'),('a','b'),('a','b'),('a','b');
select * from artificial;
c1 | t1 | t2
----+----+
1 | a | b
2 | a | b
3 | a | b
4 | a | b
5 | a | b
6 | a | b
(6 registros)
```

Este ganha dos demais, em minha opinião. O cara cria uma chave tipo ID, que ela é a única coisa que não pode ser duplicada.

Então veja que todos os registros estão duplicados, pois a responsabilidade do SGBD é apenas a de não duplicar o campo c1.

Sugestão: para tabelas secundárias (daquelas com código e descrição), sempre usar a restrição UNIQUE no campo descrição.

Para tabelas primárias sempre que possível use chaves naturais.

Integridade Referencial

Garante que um mesmo valor apareça em duas relações.

Tradução livre do documentação "CBT Integrity Referential":

http://techdocs.postgresql.org/college/002_referentialintegrity/

Integridade Referencial (relacionamento) é onde uma informação em uma tabela se refere à informações em outra tabela e o banco de dados reforça a integridade.

Onde é Utilizado?

Onde pelo menos em uma tabela precisa se referir para informações em outra tabela e ambas precisam ter seus dados sincronizados.

Exemplo: uma tabela com uma lista de clientes e outra tabela com uma lista dos pedidos efetuados por eles.

Com integridade referencial devidamente implantada nestas tabelas, o banco irá garantir que você nunca irá cadastrar um pedido na tabela pedidos de um cliente que não exista na tabela clientes.

O banco pode ser instruído para automaticamente atualizar ou excluir entradas nas tabelas quando necessário.

Primary Key (Chave Primária) - é o campo de uma tabela criado para que as outras tabelas relacionadas se refiram a ela por este campo. Impede mais de um registro com valores iguais. É a combinação interna de UNIQUE e NOT NULL.

Qualquer campo em outra tabela do banco pode se referir ao campo chave primária, desde que tenham o mesmo tipo de dados e tamanho da chave primária.

Exemplo:

clientes (codigo INTEGER, nome_cliente VARCHAR(60))

codigo nome_cliente

1 PostgreSQL inc.

2 RedHat inc.

pedidos (relaciona-se à Clientes pelo campo cod_cliente)

cod_pedido cod_cliente descricao

Caso tentemos cadastrar um pedido com cod_cliente 2 ele será aceito.

Mas caso tentemos cadastrar um pedido com cod_cliente 3 ele será recusado pelo banco.

Criando uma Chave Primária

Deve ser criada quando da criação da tabela, para garantir valores exclusivos no campo.

CREATE TABLE clientes(cod_cliente BIGINT, nome_cliente VARCHAR(60))

PRIMARY KEY (cod_cliente));

Criando uma Chave Estrangeira (Foreign Keys)

É o campo de uma tabela que se refere ao campo Primary Key de outra.

O campo pedidos.cod_cliente refere-se ao campo clientes.codigo, então pedidos.cod_cliente é uma chave estrangeira, que é o campo que liga esta tabela a uma outra.

```
CREATE TABLE pedidos(
cod_pedido BIGINT,
cod_cliente BIGINT REFERENCES clientes,
descricao VARCHAR(60)
);
```

Outro exemplo:

```
FOREIGN KEY (campo1, campo2)
REFERENCES tabela1 (campo1, campo2)
ON UPDATE CASCADE
ON DELETE CASCADE);
```

Cuidado com exclusão em cascata. Somente utilize com certeza do que faz.

Dica: Caso desejemos fazer o relacionamento com um campo que não seja a chave primária, devemos passar este campo entre parênteses após o nome da tabela e o mesmo deve obrigatoriamente ser UNIQUE.

...
cod_cliente BIGINT REFERENCES clientes(nomecampo),
...

Parâmetros Opcionais:

ON UPDATE parametro e ON DELETE parametro.

ON UPDATE paramentros:

NO ACTION (RESTRICT) - quando o campo chave primária está para ser atualizado a atualização é abortada caso um registro em uma tabela referenciada tenha um valor mais antigo. Este parâmetro é o default quando esta cláusula não recebe nenhum parâmetro.

Exemplo: ERRO Ao tentar usar:

UPDATE clientes SET codigo = 5 WHERE codigo = 2.

Ele vai tentar atualizar o código para 5 mas como em pedidos existem registros do cliente 2 haverá o erro.

CASCADE (Em Cascata) - Quando o campo da chave primária é atualizado, registros na tabela referenciada são atualizados.

Exemplo: Funciona: Ao tentar usar "UPDATE clientes SET codigo = 5 WHERE codigo = 2.

Ele vai tentar atualizar o código para 5 e vai atualizar esta chave também na tabela pedidos.

SET NULL (atribuir NULL) - Quando um registro na chave primária é atualizado, todos os campos dos registros referenciados a este são setados para NULL.

Exemplo: UPDATE clientes SET codigo = 9 WHERE codigo = 5;

Na clientes o codigo vai para 5 e em pedidos, todos os campos cod_cliente com valor 5 serão setados para NULL.

SET DEFAULT (assumir o Default) - Quando um registro na chave primária é atualizado, todos os campos nos registros relacionados são setados para seu valor DEFAULT.

Exemplo: se o valor default do codigo de clientes é 999, então UPDATE clientes SET codigo = 10 WHERE codigo = 2. Após esta consulta o campo código com valor 2 em clientes vai para 999 e também todos os campos cod_cliente em pedidos.

ON DELETE parametros:

NO ACTION (RESTRICT) - Quando um campo de chave primária está para ser deletado, a exclusão será abortada caso o valor de um registro na tabela referenciada seja mais velho.

Este parâmetro é o default quando esta cláusula não recebe nenhum parâmetro.

Exemplo: ERRO em DELETE FROM clientes WHERE codigo = 2. Não funcionará caso o cod_cliente em pedidos contenha um valor mais antigo que codigo em clientes.

CASCADE - Quando um registro com a chave primária é excluído, todos os registros relacionados com aquela chave são excluídos.

SET NULL - Quando um registro com a chave primária é excluído, os respectivos campos na tabela relacionada são setados para NULL.

SET DEFAULT - Quando um registro com a chave primária é excluído, os campos respectivos da tabela relacionada são setados para seu valor DEFAULT.

Excluindo Tabelas Relacionadas

Para excluir tabelas relacionadas, antes devemos excluir a tabela com chave estrangeira.

PK 1 -----> N FK

Do lado 1 é exigida uma PK ou uma constraint UNIQUE.

Lado 1 não permite nulos.

Lado N permite nulos mas se existir a integridade garantida.

PK (Chave Primária) - é formada internamente por UNIQUE e NOT NULL

UNIQUE (Chave candidata) - permite nulos

FK (Chave Estrangeira) - permite nulos, mas se um campo for nulo estará satisfeita a constraint em consequência em consequência violada a integridade.

Recomendação: sempre usar NOT NULL nos campos da FK.

35 - Expressões Regulares para uso em Modelagem de Bancos de Dados

As expressões regulares são um grande recurso para ajudar a garantir a integridade das informações, em especial no uso com domínios.

Em ciência da computação, uma expressão regular (ou o estrangeirismo regex, abreviação do inglês regular expression) provê uma forma concisa e flexível de identificar cadeias de caracteres de interesse, como caracteres particulares, palavras ou padrões de caracteres. Expressões regulares são escritas numa linguagem formal que pode ser interpretada por um processador de expressão regular, um programa que ou serve um gerador de analisador sintático ou examina o texto e identifica partes que casam com a especificação dada.

O termo deriva do trabalho do matemático norte-americano Stephen Cole Kleene, que desenvolveu as expressões regulares como uma notação ao que ele chamava de álgebra de conjuntos regulares. Seu trabalho serviu de base para os primeiros algoritmos computacionais de busca, e depois para algumas das mais antigas ferramentas de tratamento de texto da plataforma Unix.

O uso atual de expressões regulares inclui procura e substituição de texto em editores de texto e linguagens de programação, validação de formatos de texto (validação de protocolos ou formatos digitais), realce de sintaxe e filtragem de informação.
(Wikipedia - http://pt.wikipedia.org/wiki/Express%C3%B5es_regulares)

Correspondência com o Padrão e Expressões Regulares no PostgreSQL

O PostgreSQL suporta várias formas de correspondência com o padrão (pattern matching): o tradicional operador SQL LIKE, o mais recente operador SIMILAR TO (adicionado no SQL 1999) e as expressões regulares estilo POSIX (também implementada na função substring).

As regex são um recurso muito útil aos DBAs. As expressões regulares oferecem força e agilidade.

LIKE

LIKE - case sensitive

ILIKE - case insensitive

Caracteres Coringa:

% - 0 ou mais caracteres

_ - 1 único caractere

NOT LIKE

~~ = LIKE

~~* = ILIKE

!~~ = NOT LIKE

!~~* = NOT ILIKE

SIMILAR TO

Semelhante ao LIKE mas usa expressões regulares do SQL.

Assim como o LIKE somente é válido quando toda a string corresponde ao padrão.

Em expressões regulares qualquer parte da string pode corresponder ao padrão.

Caracteres coringa:

% - 0 ou mais caracteres

_ - 1 único caractere

Adiciona:

| - ou

* - 0 ou mais vezes

+ - 1 ou mais vezes

() - agrupar itens

[] - similar ao POSIX

POSIX

~ - case sensitive e corresponde

~* - case insensitive e corresponde

!~ - case sensitive e não corresponde

!~* - case insensitive e não corresponde

Átomos do Padrão

(er) - expressão regular. Correspondência para a er

[caracteres] - corresponde a qualquer dos caracteres

\k - caractere não alfanumérico

\c - caractere alfanumérico

. - qualquer único caractere

x - este é o único caractere sem função, representa 'x' mesmo

[:alnum:] Caracteres alfanuméricos, o que no caso de ASCII corresponde a [A-Za-z0-9].

[:alpha:] Caracteres alfabéticos, o que no caso de ASCII corresponde a [A-Za-z].

[:blank:] Espaço e tabulação, o que no caso de ASCII corresponde a [\t].

[:cntrl:] Caracteres de controle, o que no caso de ASCII corresponde a [\x00-\x1F\x7F].

[:digit:] Dígitos, o que no caso de ASCII corresponde a [0-9]. O Perl oferece o atalho \d.

[:graph:] Caracteres visíveis, o que no caso de ASCII corresponde a [\x21-\x7E].

[:lower:] Caracteres em caixa baixa, o que no caso de ASCII corresponde a [a-z].

[:print:] Caracteres visíveis e espaços, o que no caso de ASCII corresponde a [\x20-\x7E].

[:punct:] Caracteres de pontuação, o que no caso de ASCII corresponde a [-!"#\$

%&'(*)+,.:/;<=>?@[\\\]_`{}~].

[:space:] Caracteres de espaços em branco, o que no caso de ASCII corresponde a [\t\r\n\f]. O Perl oferece o atalho \s, que, entretanto, não é exatamente equivalente; diferente do \s, a classe ainda inclui um tabulador vertical, \x11 do ASCII.[4]

[:upper:] Caracteres em caixa alta, o que no caso de ASCII corresponde a [A-Z].

[:xdigit:] Dígitos hexadecimais, o que no caso de ASCII corresponde a [A-Fa-f0-9].

Expressões regulares não podem terminar com \.

Quantificadores dos caracteres do Padrão:

* - uma seqüência de 0 ou mais correspondências do átomo

+ - uma seqüência de 1 ou mais correspondências do átomo

? - uma seqüência de 0 ou 1 correspondência do átomo

{m} - uma seqüência de exatamente m correspondências do átomo

{m,} - uma seqüência de m ou mais correspondências do átomo

{m,n} - uma seqüência de m a n (inclusive) correspondências do átomo; m não pode ser maior do que n

Caracteres Delimitadores do Padrão

^ - início da string

\$ - final da string

Alguns exemplos:

```
SELECT regexp_matches('foobarbequebazilbarfbonk', '(b[^b]+)(b[^b]+)', 'g');
regexp_split_to_table function splits a string using a POSIX regular expression pattern as a delimiter.
```

The `regexp_split_to_array` function behaves the same as `regexp_split_to_table`, except that `regexp_split_to_array` returns its result as an array of text. It has the syntax `regexp_split_to_array(string, pattern [, flags])`. The parameters are the same as for `regexp_split_to_table`.

```
SELECT foo FROM regexp_split_to_table('the quick brown fox jumped over the lazy dog',
E'\\s+') AS foo;
```

```
foo
```

```
-----
```

```
the
quick
brown
fox
jumped
over
the
lazy
dog
```

```
(9 rows)
```

```
SELECT regexp_split_to_array('the quick brown fox jumped over the lazy dog', E'\\s+');
      regexp_split_to_array
```

```
{the,quick,brown,fox,jumped,over,the,lazy,dog}
```

```
(1 row)
```

```
SELECT foo FROM regexp_split_to_table('the quick brown fox', E'\\s*') AS foo;
```

Example of how to escape "_" in a simple query

```
create table foo (str varchar(16));
insert into foo (str) values ('abc.defghi');
insert into foo (str) values ('abc_defghi');
```

I want to select all strings starting with abc_def

```
select * from foo where str like E'abc\\_def%';
```

> Fernando Brombatti wrote:

>> Alguém já usou função para extrair números de uma string?

```
>>
>> Ex.: AB345CD234 => 345234
>
>
> lista=# select regexp_replace('AB345CD234', '[A-Z]', ''g');
>   regexp_replace
> -----
> 345234
> (1 row)
>
```

Dica na lista pgbe-geral:

Complementando a resposta do Shander:

Caso sua string possa conter outros caracteres não numéricos, além das letras [A-Z], o uso de '[^[:digit:]]' é mais abrangente.

<http://www.postgresql.org/docs/current/interactive/functions-matching.html#FUNCTIONS-POSIX-REGEXP>

```
bdteste=# SELECT regexp_replace('AB3,45CD/xz234', '[^[:digit:]]', '',
'g');
regexp_replace
-----
345234
Osvaldo
```

Bom tutorial no site:

http://www.oreillynet.com/pub/a/databases/2006/02/02/postgresq_regexes.html

Reproduzindo aqui os exemplos do tutorial adaptados.

Vamos criar uma tabela:

CREATE DATABASE regex;

CREATE TABLE myrecords(record text);

Insira os registros:

```
insert into myrecords (record) values
('a'),
('ab'),
('abc'),
```

```
('123abc'),
('132abc'),
('123ABC'),
('abc123'),
('4567'),
('5678'),
('6789');
```

Consultas simples usam o operador ~ (til) seguido por uma string e retornam somente os que atendem ao case.

`SELECT record FROM myrecords WHERE record ~ '1';` -- Retornaram todos os registros que contenham '1'.

`SELECT record FROM myrecords WHERE record ~ 'a';`

`SELECT record FROM myrecords WHERE record ~ 'A';`

`SELECT record FROM myrecords WHERE record ~ '3a';`

Para retornar sem olhar o case usamos ~*:

`SELECT record FROM myrecords WHERE record ~* 'a';`

`SELECT record FROM myrecords WHERE record ~* '3a';`

Agora não trazendo o que contém a string e sensível ao case !~:

`SELECT record FROM myrecords WHERE record !~ '1';`

Trazendo sem olhar o case e não trazendo onde tem a string !~*:

`SELECT record FROM myrecords WHERE record !~* 'c';`

Trazendo as strings que comecem com um certo caractere (^):

`SELECT record FROM myrecords WHERE record ~ '^1';`

`SELECT record FROM myrecords WHERE record ~ '^a';`

`SELECT record FROM myrecords WHERE record ~* '^a';`

Terminados com (\$):

`SELECT record FROM myrecords WHERE record ~ 'c$';`

`SELECT record FROM myrecords WHERE record ~ 'bc$';`

`SELECT record FROM myrecords WHERE record ~* 'bc$';`

Veja agora algumas consultas e analise seus resultados:

`SELECT record FROM myrecords WHERE record ~ '[a]';` -- Qualquer que tenha a

`SELECT record FROM myrecords WHERE record ~ '[A]';` -- Qualquer que tenha A

`SELECT record FROM myrecords WHERE record ~* '[a]';` -- Qualquer que tenha a ou A

`SELECT record FROM myrecords WHERE record ~ '[ac]';` -- Qualquer que tenha a ou c

`SELECT record FROM myrecords WHERE record ~ '[ac7]';` -- Qualquer que tenha a ou c ou 7

SELECT record FROM myrecords WHERE record ~ '[a7A]'; -- Qualquer que tenha a ou 7 ou A

SELECT record FROM myrecords WHERE record ~* '[ac7]'; -- Qualquer que tenha a ou c ou 7 sem olhar o case

SELECT record FROM myrecords WHERE record ~ '[z]';

SELECT record FROM myrecords WHERE record ~ '[z7]';

SELECT record FROM myrecords WHERE record !~ '[4a]';

Procurar por uma faixa de valores:

SELECT record FROM myrecords WHERE record ~ '[1-4]';

Outros de faixa:

SELECT record FROM myrecords WHERE record ~ '[a-c5]';

SELECT record FROM myrecords WHERE record ~* '[a-c5]';

SELECT record FROM myrecords WHERE record ~ '[a-cA-C5-7]'; -- 3 faixas, a-c, A-C e 5-7

Correspondendo 2 ou mais caracteres:

SELECT record FROM myrecords WHERE record ~ '3[a]';

SELECT record FROM myrecords WHERE record ~ '[3][a]';

SELECT record FROM myrecords WHERE record ~ '[1-3]3[a]';

SELECT record FROM myrecords WHERE record ~ '[23][a]';

SELECT record FROM myrecords WHERE record ~ '[2-3][a]';

SELECT record FROM myrecords WHERE record ~ '[a-b][b-c]';

Nesta ordem:

SELECT record FROM myrecords WHERE record ~ '[a][c]';

Iniciando com dígitos:

SELECT record FROM myrecords WHERE record ~ '^[0-9]\$';

Fazendo escolhas (ou |):

SELECT record FROM myrecords WHERE record ~ '^a|c\$';

Começando com a ou 5 ou terminando com c:

SELECT record FROM myrecords WHERE record ~ '^a|c\$|^5';

SELECT record FROM myrecords WHERE record ~ '^[^0-9|^a-z]';

Repetindo Caracteres:

SELECT record FROM myrecords WHERE record ~ 'a*'; -- 0 ou mais

SELECT record FROM myrecords WHERE record ~ 'b+'; -- 1 ou mais

SELECT record FROM myrecords WHERE record ~ 'a?'; -- 0 ou 1

SELECT record FROM myrecords WHERE record ~ '[0-9]{3}'; -- Exatamente uma quantidade, usar {#}

SELECT record FROM myrecords WHERE record ~ '[0-9]{4,}'; -- Exatamente ou mais, usar {#,}

SELECT record FROM myrecords WHERE record ~ '[a-zA-Z0-9]{2,3}'; -- Exatamente de 2 até 3, inclusive

Exemplos com a função Substring:

CREATE TABLE log(record text);

Inserir registros:

```
insert into log (record) values
('a'),
('ab'),
('abc'),
('123abc'),
('132abc'),
('123ABC'),
('abc123'),
('4567'),
('5678'),
('6789');
```

SELECT substring(record, '[a-zA-Z0-9]..){1,}') FROM log LIMIT 1;

SELECT date(substring(record, '[a-zA-Z]{1,}[0-9]{1,}') || ' 2005') AS "Date" FROM log LIMIT 1;

SELECT substring('Nov 3 07:37:51 localhost', '[:0-9]{2,}') AS "Time";

```
SELECT substring('Nov 30 07:37:51 localhost', '[:0-9]{2,}') AS "Time";  
  
SELECT substring('Nov 30 07:37:51 localhost', '[:0-9]{3,}') AS "Time";  
  
SELECT substring(record, '[:0-9]{3,}') AS "Time" FROM log LIMIT 1;  
  
SELECT substring(record, 'SRC=*( [.0-9]{2,})') AS "IP Address" FROM log LIMIT 1;  
  
SELECT substring(record, 'SPT=*( [.0-9]{2,})') AS "Remote Source Port"  
      FROM log LIMIT 1;  
SELECT substring(record, 'DPT=*( [.0-9]{2,})') AS "Destination Port"  
      FROM log LIMIT 1;
```

O SQL completo:

```
SELECT  
    date(substring(record, '[a-zA-Z ]{1,}[0-9]{1,}') || ' 2005') AS "Date",  
    substring(record, '[:0-9]{3,}') AS "Time",  
    substring(record, 'SRC=*( [.0-9]{2,})') AS "Remote IP Address",  
    substring(record, 'SPT=*( [.0-9]{2,})') AS "Remote Source Port",  
    substring(record, 'DPT=*( [.0-9]{2,})') AS "Destination Port"  
FROM log;
```

Mais detalhes no tutorial e na documentação do PostgreSQL:
<http://pgdocptbr.sourceforge.net/pg80/functions-matching.html>
<http://www.postgresql.org/docs/8.3/interactive/functions-matching.html>

36 - Tipos e Domínios

A criação de novos tipos e domínios reforçam e muito a robustez dos bancos de dados. Podemos criar um tipo que seja mais adequado para nosso campo e depois, criar um domínio em cima desse tipo para que seja reforçada a restrição, usando-se constraints como CHECK, NOT NULL, UNIQUE e outras. Veja exemplos.

Tipos

```
create type t_humor as enum(
'triste','normal','alegre'
);
create table humores(chave int, humor t_humor);
insert into tipos values(1, 'alegre');

create type t_sexo as enum(
'masculino','feminino'
);
create table sexos(chave int, sexo t_sexo);
insert into sexos values(1, lower('Masculino')::tsexo);
select * from sexos;
select chave, initcap(sexo::text) from sexos;
```

Domínios

```
CREATE DOMAIN dom_cep AS text
CONSTRAINT chk_cep CHECK (VALUE ~ '^\d{8}$') NOT NULL;
-- Um domínio pode ser usado como tipo, com vantagem de ampliar as restrições
-- Veja este: numérico, tamanho 8
CREATE FUNCTION formata_cep(cep dom_cep) RETURNS TEXT AS $$%
BEGIN
RETURN substr(cep,1,5) || '-' || substr(cep,6,3);
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TABLE tbl_cep (cep dom_cep);
insert into tbl_cep values ('60430440');
insert into tbl_cep values (60430440);
insert into tbl_cep values ('123mjhyu');
insert into tbl_cep values ('123456789');
```

```
SELECT formata_cep(CAST('60420440' AS dom_cep)); -- retorna 60420-440
```

```
SELECT formata_cep('60420440');
```

```
CREATE DOMAIN dom_cep_nn AS text  
CONSTRAINT chk_cep CHECK (VALUE ~ '^\d{8}$') NOT NULL;
```

Exemplo na versão 8.0:

```
create type sexo as
(
m text,f text
);
```

```
CREATE DOMAIN dsexo AS text
CONSTRAINT chk_sexo CHECK (VALUE = 'feminino' OR VALUE = 'masculino');
create table tab_sexo(s dsexo, sx sexo);
insert into tab_sexo values('feminino', ('d','g'));
insert into tab_sexo values('feminino', ('d',8));
insert into tab_sexo values('masculino', ('d',8));
insert into tab_sexo values('dará erro', ('d',8));
```

37 - Fases de um Projeto de Banco de Dados

Projeto de Banco de Dados Relacional

Objetivo - gerar um banco de dados que permita armazenar informações sem redundância e recuperá-las com facilidade.

Fases de um Projeto

Na modelagem de um banco de dados devemos primeiramente ter conhecimento do ambiente real ao qual será aplicado o modelo. Durante o processo de modelagem deverá ser usado o bom senso e as regras inerentes ao ambiente de aplicação do banco de dados.

1) Levantamento de Requisitos

Um modelo de dados de alto nível oferece ao projetista conceitos que o possibilitam especificar as necessidades dos usuários e como o banco será estruturado para atender plenamente todas as necessidades. Aqui são importantes as entrevistas e a avaliação do projetista.

Resultado dessa fase - Especificação das necessidades dos usuários (levantamento de requisitos).

2) Projeto Conceitual

A fase conceitual depende muito da habilidade do projetista e das qualidades do modelo de dados adotado.

Escolha do modelo de dados, para com ele transcrever as necessidades e informações coletadas para um esquema de banco de dados.

O projeto conceitual indicará as necessidades funcionais da empresa, as consultas, exclusões, etc.

Então deve-se rever o esquema dos dados para adequar às necessidades funcionais.

O projeto conceitual gera o esquema conceitual. No projeto conceitual não se leva em conta o SGBD que será utilizado.

O propósito do projeto conceitual é descrever o conteúdo de informação do banco de dados ao invés das estruturas de armazenamento.

3) Projeto Lógico

Tem por objetivo avaliar o esquema conceitual frente às necessidades de uso do banco de dados pelos usuários e aplicações, realizando possíveis refinamentos com a finalidade de melhorar o desempenho das operações.

Um esquema lógico é uma descrição da estrutura do banco de dados que pode ser processada por um SGBD.

Depende do modelo de dados adotado pelo SGBD, mas não especificamente do SGBD.

Neste mapeamos o modelo conceitual para o modelo de implementação (físico).

O projeto lógico gera o esquema lógico.

4) Projeto Físico

Este toma por base o esquema lógico para gerar o esquema físico e é direcionado para um específico SGBD.

O projeto físico gera o esquema físico.

Fases práticas:

- Análise de requisitos
- Identificação das relações e atributos
- Identificar as chaves das relações
- Analisar as pendências anteriores e aprofundar a modelagem
- Focar nos atributos, seus tipos, domínios e constraints. Substituir multivalorados, repetidos e nulos
- Gerar relacionamentos

Um Bom Projeto de Banco de Dados Evita:

- Inconsistência e redundância
- Dificuldade de acesso pela falta de planejamento
- Isolamento de dados
- Problemas de integridade
- Problema na falta de atomicidade nas transações
- Anomalias no acesso concorrente
- Problemas de segurança
- Operações entre disco e memória (minimizar)

Para normalizar as informações precisamos colher informações detalhadas sobre a empresa real para a qual iremos modelar o banco de dados.

O que evitar?

- informações repetidas
- dificuldade na recuperação de informações

Repetições

- desperdiçam espaço
- dificultam (engessam) atualizações

Exemplo:

Temos um cadastro de clientes assim:

(nome, fone, numero, rua, bairro, cidade, uf).

Com uma grande quantidade de registros, caso sofra alteração em algum dos campos multivalorados, como uf, teremos que atualizar todos os registros dos clientes.

Caso normalizemos a tabela de clientes e ufs sejam uma tabela separada apenas se relacionando com clientes, ao alterar uma uf apenas atualizaremos um registro na tabela ufs, sem contar que não cadastraremos a uf em cada cliente, mas apenas uma vez na tabela ufs.

Decomposição

Quando decomponemos uma relação em várias, devemos ter cuidado para que não haja perda de informações nas dependências.

38 - Ferramentas

Validação e Geração de IE em JavaScript

Como tive dificuldade para encontrar funções tanto em JavaScript quanto em PL/PgSQL, acabei desenvolvendo funções nessas linguagens para a Inscrição Estadual, tanto para validar quanto para gerar números de teste.

Estão no mesmo diretório dos modelos:

<http://pg.ribafs.net/down/modelagem>

Como a Inscrição Estadual é diferente para cada estado, apenas criei para o Ceará. Quem quiser criar para seu estado basta pegar o algoritmo no SINTEGRA e adaptar a função existente.

As funções em PL/PgSQL estão no arquivo sql do modelo pessoa, juntamente com uma função de validação de CPF e CNPJ de autoria do Juliano, encontrada no iMasters.

Links Úteis para Modelagem

Consulta online de CNPJ no site da Receita

http://www.receita.fazenda.gov.br/PessoaJuridica/CNPJ/cnpjreva/Cnpjreva_Solicitacao.asp

Gerador/Validador de CNPJ e CPF online

<http://highportal.no.sapo.pt/geradorcpf.html>

Obs.: Como é javascript dentro do próprio arquivo HTML, pode ser baixado para uso off line.

SINTEGRA (Inscrição Estadual)

<http://www.sintegra.gov.br/>

Exemplo: Ceará

<http://www.sefaz.ce.gov.br/Sintegra/Sintegra.Asp?estado=CE>

(Consulta do CNPJ e da IE. Consulte sem máscara)

Busca de CEP

http://www.correios.com.br/servicos/cep/cep_default.cfm

Testador online para expressões POSIX

<http://www.spaweditor.com/scripts/regex/index.php>

Informações Úteis sobre estes documentos:

CNPJ - é único (até as filiais também têm o seu). Órgãos públicos são assim, a sede administrativa tem um CNPJ e suas unidades têm um CNPJ emelhante

IE (Inscrição Estadual) - único, sem nulo, mas quando não existir informar ISENTO (recomendação do SINTEGRA). A formação é própria de cada estado.

IM (Inscrição Municipal) - único, mas é exigido somente para empresas de prestação de serviço e outros específicos.

A máscara do CNPJ é 99.999.999/9999-99. O número do CNPJ mesmo são os 8 dígitos antes da barra e os 4 dígitos seguintes a barra são o numero da empresa e os 2 últimos são dígitos verificadores.

Logo em caso de matriz e filiais os 8 primeiros são iguais já que são da mesma empresa e os 4 dígitos seguintes são 0001 para matriz e a sequência são filiais.

Empresas com filiais, usam o mesmo CNPJ, com a diferença de os 4 dígitos finais:

Se a matriz for 00.000.000/0001-00

A filia será 00.000.000/0002-00

Exemplos de CNPJ

00043711000143

00043711001115

Exemplo de IE (Ceará)

060000015

39 - Modelagem de Bancos de Dados

SQL - Select Query Language

A SQL usa uma combinação de construtores em álgebra e cálculo relacional. Possui recursos para consultas, definição de estrutura, modificação e especificação de restrições de segurança dos dados.

O modelo proposto por Codd é hoje considerado a base de trabalho para qualquer Sistema de Gestão de Base de Dados Relacional (SGBDR).
(Detalhes na Wikipédia - <http://pt.wikipedia.org/wiki/SQL>).

SGBD - é formado por um conjunto de dados inter-relacionados e uma coleção de programas para auxiliar no gerenciamento dos dados.

Projeto de Banco de Dados Relacional

Objetivo - gerar um banco de dados que permita armazenar informações sem redundância e recuperá-las com facilidade.

Fases de um Projeto

Na modelagem de um banco de dados devemos primeiramente ter conhecimento do ambiente real ao qual será aplicado o modelo. Durante o processo de modelagem deverá ser usado o bom senso e as regras inerentes ao ambiente de aplicação do banco de dados.

1) Levantamento de Requisitos

Um modelo de dados de alto nível oferece ao projetista conceitos que o possibilitam especificar as necessidades dos usuários e como o banco será estruturado para atender plenamente todas as necessidades. Aqui são importantes as entrevistas e a avaliação do projetista.

Resultado dessa fase - Especificação das necessidades dos usuários (levantamento de requisitos).

2) Projeto Conceitual

A fase conceitual depende muito da habilidade do projetista e das qualidades do modelo de dados adotado.

Escolha do modelo de dados, para com ele transcrever as necessidades e informações coletadas para um esquema de banco de dados.

O projeto conceitual indicará as necessidades funcionais da empresa, as consultas, exclusões, etc.

Então deve-se rever o esquema dos dados para adequar às necessidades funcionais. O projeto conceitual gera o esquema conceitual. No projeto conceitual não se leva em conta o SGBD que será utilizado.

O propósito do projeto conceitual é descrever o conteúdo de informação do banco de dados ao invés das estruturas de armazenamento.

3) Projeto Lógico

Tem por objetivo avaliar o esquema conceitual frente às necessidades de uso do banco de dados pelos usuários e aplicações, realizando possíveis refinamentos com a finalidade de melhorar o desempenho das operações.

Um esquema lógico é uma descrição da estrutura do banco de dados que pode ser processada por um SGBD.

Depende do modelo de dados adotado pelo SGBD, mas não especificamente do SGBD.

Neste mapeamos o modelo conceitual para o modelo de implementação (físico).

O projeto lógico gera o esquema lógico.

4) Projeto Físico

Este toma por base o esquema lógico para gerar o esquema físico e é direcionado para um específico SGBD.

O projeto físico gera o esquema físico.

Fases práticas:

- Análise de requisitos
- Identificação das relações e atributos
- Identificar as chaves das relações
- Analisar as pendências anteriores e aprofundar a modelagem
- Focar nos atributos, seus tipos, domínios e constraints. Substituir multivalorados, repetidos e nulos
- Gerar relacionamentos

Um Bom Projeto de Banco de Dados Evita:

- Inconsistência e redundância
- Dificuldade de acesso pela falta de planejamento
- Isolamento de dados
- Problemas de integridade
- Problema na falta de atomicidade nas transações
- Anomalias no acesso concorrente
- Problemas de segurança
- Operações entre disco e memória (minimizar)

Funções do DBA

- Definir esquema do banco de dados
- Definir estrutura dos dados e o método de acesso
- Alterações na estrutura
- Acesso ao banco de dados
- Regras de integração

Funções do Programador

- Trabalhar numa linguagem que acesse um banco de dados usando a DML.

Para normalizar as informações precisamos colher informações detalhadas sobre a empresa real para a qual iremos modelar o banco de dados.

O que evitar?

- informações repetidas
- dificuldade na recuperação de informações

Repetições

- desperdiçam espaço
- dificultam (engessam) atualizações

Exemplo:

Temos um cadastro de clientes assim:

(nome, fone, numero, rua, bairro, cidade, uf).

Com uma grande quantidade de registros, caso sofra alteração em algum dos campos multivvalorados, como uf, teremos que atualizar todos os registros dos clientes.

Caso normalizemos a tabela de clientes e ufs sejam uma tabela separada apenas se relacionando com clientes, ao alterar uma uf apenas atualizaremos um registro na tabela ufs, sem contar que não cadastraremos a uf em cada cliente, mas apenas uma vez na tabela ufs.

Decomposição

Quando decomponemos uma relação em várias, devemos ter cuidado para que não haja perda de informações nas dependências.

Arquivos de dados em disco:

- arquivos de dados
- dicionário de dados (catálogo)
- índices
- dados estatísticos

Modelo Relacional segundo C. J. Date

- Um modelo de dados é uma definição abstrata, lógica, dos objetos, operadores (funções) e demais.
- Objetos são usados para modelar a estrutura de dados
- Operadores são usados para modelar o comportamento dos dados
- A implementação de um modelo é a realização física na máquina real.
- Diagramas E/R não representam tudo que interessa dos relacionamentos entre as entidades.
- Uma relação que contém nulos não é uma relação.
- Um modelo relacional com nulos não é um modelo relacional.
- Nulos são um erro e nunca deveriam ter sido adotados.
- Toda função é um operador mas nem todo operador é uma função.
- Relações - Atributos - Tipos

Correspondência de termos entre os modelos lógico e físico:

- Relações - Tabelas
- Tupla - Registros
- Atributos - Campos ou colunas
- Modelo - Implementação

As quatro Operações das Relações:

- Relações são normalizadas
- Atributos são desordenados, da esquerda para a direita

- Tuplas são desordenadas, de cima para baixo
- Não existem tuplas duplicadas

Dicionário de dados - dados sobre dados - metadados (catálogo do sistema no PostgreSQL).

Esta última subentende chave natural em todas as tabelas. Mas se os SGBDs implementarem assim, como vamos vazer nossos testes e demonstrações? (nota)

- Valores de atributos são valores simples, mas esses valores podem ser absolutamente qualquer coisa. Nós rejeitamos a antiga noção de "valor atômico".

Ou seja, justificando tipos como geométricos, arrays, etc. (nota)

- A cardinalidade de um conjunto é seu número de elementos e de uma relação é seu número de tuplas.

- O projeto de um banco de dados tem mais de arte que de ciência.

DER

Cuidado com diagramas, use-os apenas para ajudar no processo do projeto, lembrando que um DER não faz todo o trabalho de modelagem e nem representa tudo de um modelo.

(CJ Date em seu livro: An Introduction To Database Systems)

Modelo Relacional (MR)

- Consiste de uma coleção de relações (tabelas no modelo físico), cada uma com um nome único (por esquema).
- Cada relação é composta por atributos (campos, no modelo físico).
- Cada atributo tem seu tipo ou domínio.
- Temos também as constraints (restrições). Relacionamentos são formados por constraints. No caso chamado de integridade referencial.
- Uma relação é formada por um conjunto de tuplas (registros no modelo físico).
- No modelo relacional uma relação tanto representa os dados quanto as relações entre eles.

NULOS - acarretam sérias dificuldades e devem ser evitados.

Normalizando Tabelas

- Atributos Multivalorados (telefones, municipio, uf, etc) indicam necessidade de criação de outra tabela.
- A repetição de valores de atributos também deve ser evitada criando-se uma outra tabela.
- A presença de nulos também é resolvida com a normalização e consequente criação de outras tabelas.
- O modelo relacional foi fundamentado na teoria dos conjuntos e na lógica dos predicados.

Se fosse buscar por inspiração no sentido mais exato, diria que era na crise de software: facilitar o desenvolvimento de grandes bases de dados usadas simultaneamente por muitos usuários.

- Seus termos principais são: relações, atributos, restrições e tipos!
- Relacionamento não é um termo técnico deste modelo, mas do MER, aqui temos as restrições de integridade referencial.
- A linguagem SQL não é inteiramente relacional. SGBD relacionais restringem o modelo para usarem essa linguagem.

Por isso mesmo não os chamo de SGBDRs, mas de SGBDs SQL. Os SGBDRs que conheço são o IBM BS/12, o Ingres QUEL original, o Alphora Dataphor e outros atualmente em desenvolvimento, anteriormente listados.

- É bom distinguir MR (modelo relacional) de MER (modelo entidade relacionamento). Este último surgiu depois do relacional, mas a grande maioria dos SGBDs atuais implementam o modelo relacional ou uma versão adaptada dele.

(E-mail respondido na lista pgbr-geral, por Leandro Dutra)

Normalização

Leandro Dutra (na lista pgbr-geral).

Em princípio, todas as chaves naturais são boas, e todas as artificiais são ruins.

O caso é que tem muita gente, principalmente usuários de ORMs, que não gosta de chaves compostas. Acha que fica difícil programar. Eu nunca usei ORM, nunca senti na pele, e daí vem minha impressão de que muitos ORMs criam tantos problemas quanto resolvem.

Imagine uma relação pai com uma chave natural composta razoável, digamos mais de três atributos; e uma relação filha com muito mais tuplas, digamos milhões ou dezenas de milhões a mais,

isso rodando num sistema que é gargalo de desempenho. Nesse caso, pode ser que, após testes, valha a pena uma chave primária artificial para emagrecer a tabela filha.

Uma vez cheguei num acordo interessante com uma equipe apaixonada pelo Hibernate: eu deixava criar chaves artificiais quando as naturais tinham mais de três atributos. Doutra vez, tive de aceitar as chaves artificiais mas lavei as mãos quanto a desempenho e manutenibilidade (essa palavra existe?) do modelo e da aplicação, e ainda impus chaves alternativas naturais.

(Leandro)

Mais alguns tópicos sobre normalização e cia:

Valor default

Chaves naturais x artificiais

Null

--Default

```
create table nula(c1 serial primary key, c2 int, c3 int default 0);
insert into nula (c1) values (default),(default),(default),(default);
select * from nula;
c1 | c2 | c3
----+---+---
 1 |   | 0
 2 |   | 0
 3 |   | 0
 4 |   | 0
(4 registros)
```

Veja só que “riqueza” de registros! Tudo isso graças a permissão de nulo e ao valor default.

--Nulo

```
create table nula2(c1 int primary key, c2 int check(c2 > 0), c3 int);
insert into nula2(c1,c2,c3) values (1,default,4); -- Será válido. Importante: use not null
insert into nula2(c1,c2,c3) values (2,-3,4)
select * from nula2;
```

```
c1 | c2 | c3
----+---+---
 1 |   | 4
(1 registro)
```

Uma "incoerência" no comportamento do nulo, que reforça a recomendação de se evitar seu uso.

--Chave artificial

```
create table artificial(c1 serial primary key, t1 text, t2 text);
insert into artificial(t1,t2) values ('a','b'),('a','b'),('a','b'),('a','b'),('a','b'),('a','b');
select * from artificial;
c1 | t1 | t2
----+---+---
 1 | a | b
 2 | a | b
 3 | a | b
 4 | a | b
 5 | a | b
 6 | a | b
(6 registros)
```

Este ganha dos demais, em minha opinião. O cara cria uma chave tipo ID, que ela é a única coisa que não pode ser duplicada.

Então veja que todos os registros estão duplicados, pois a responsabilidade do SGBD é apenas a de não duplicar o campo c1.

Sugestão: para tabelas secundárias (daquelas com código e descrição), sempre usar a restrição UNIQUE no campo descrição.

40 - Modelando um Banco Pessoa

Este modelo encontra-se no site:

<http://pg.ribafs.net/down/modelagem>

Veja também no mesmo site o modelo do banco de CEPs e os anteriores (controle de estoque e vídeo locadoras), mas lembrando que esses dois iniciais tem vários ajustes que são requeridos, para que sejam modelos mais robustos.

Modelagem de um Banco de Dados pessoa no PostgreSQL

Observações úteis sobre esta modelagem:

Até o último banco de dados que criei, o cadastro de pessoas era geralmente algo como:

pessoa(codigo, nome, rua, numero, bairro, cidade, uf, cep)

Ou seja, desnortinalizado, quese todos os campos permitindo valores duplicados.

Para mim foi muito proveitosa a discussão na lista de postgresql, como também algumas leituras sobre o assunto, que me fizeram perceber sua grande importância.

Inclusive comparando este modelo com os dois anteriores (controle de estoque e vídeo locadoras), a diferença é grande:

- O CPF permitia nulos no controle de estoque
- Agora ele tem um domínio que valida seus dados e um índice parcial que permite a entrada "informal" e entradas válidas para CPF (sendo que para estas últimas é aplicado o índice único).
- As tabelas clientes, fornecedores e ceps foram normalizadas.

As três Primeiras Formas Normais (aplicadas neste modelo):

1FN - todos os atributos possuem valores simples (nenhum é composto)

2FN - deve estari na 1FN e não possuir dependência funcional parcial (todos os atributos dependem integralmente da chave primária)

2FN - deve estar na 2FN e não possuir nenhuma dependência funcional transitiva (nenhum atributo pode depender de outro atributo que não seja PK)

Quando existem dependências cíclicas ou multivaloradas devemos aplicar a 4FN e a 5FN.

Após a divulgação do modelo controle de estoque, tivemos um longo debate na lista de PostgreSQL, o que melhorou meus conhecimentos sobre o modelo relacional, modelagem, normalização e cia. Agora trago mais um modelo, mais simples mas acredito que mais coerente e mais robusto, mas mesmo assim deve ter itens a melhorar.

Algo muito importante é a implementação de chaves naturais (pelo que vejo um dos maiores responsáveis por garantir a unicidade dos registros). Veja que as relações pessoas, municipios, ceps, enderecos e juridicas estão com chaves naturais, mas o mais importante aqui é a idéia, o conceito, que devemos ter na chave primária, campo(s) que sejam realmente representativos da relação. Fechando, a chave deve impedir totalmente duplicações.

Só para exemplificar, veja as duas tabelas abaixo:

```
create table juridicas
(
    cnpj dom_cnpj primary key,
    inscricao_estadual dom_ie_ce,
    site dom_url
);
```

```
create table juridicas2
(
    id int primary key,
    cnpj char(20),
    inscricao_estadual char(10),
    site char50
);
```

Na tabela juridicas2, podemos inserir registros com nomes duplicados com grande facilidade, já que não existe nenhum controle sobre isso pelo SGBD. Para o SGBD apenas será fiscalizado se o campo juridica (um ID) não será duplicado. Já na tabela juridicas é praticamente impossível duplicar um registro, pois a chave é um CNPJ, o SGBD não deixa duplicar.

No modelo original (do Ary Júnior) o endereço guarda pessoa, assim como o telefone também guarda pessoa.

Então fui refletir um pouco e encontrei pelo menos um bom motivo para fazer diferente,

adicionar o
endereço em pessoas:

Sabemos que existem mais pessoas que endereços.

Para cada nova pessoa cadastrada teremos que cadastrar também um novo endereço. E o mais grave é que, em sendo pessoas físicas,
haverá duplicação de endereços para as pessoas de um mesmo endereço.

Colaboração de Ribamar FS (<http://pg.ribafs.net>).

Artigo fonte de inspiração: Estudo de Caso de Projeto de Bancos de Dados para Contas a Pagar e Receber,
de Ary Júnior na SQL Magazine 52.

-- Domínios

```
-- SELECT * FROM information_schema.domains WHERE domain_schema='public';
```

Domínios

Um domínio se baseia em um determinado tipo base e, para muitas finalidades, é intercambiável com o seu tipo base. Entretanto, o domínio pode ter restrições limitando os valores válidos a um subconjunto dos valores permitidos pelo tipo base subjacente.

Se a coluna for baseada em um domínio, esta coluna se refere ao tipo subjacente do domínio (e o domínio é identificado em domain_name e nas colunas associadas).

```
CREATE DOMAIN nome [AS] tipo_de_dado
[ DEFAULT expressão ]
[ restrição [ ... ] ]
```

onde restrição é:

```
[ CONSTRAINT nome_da_restrição ]
{ NOT NULL | NULL | CHECK (expressão) }
```

O comando CREATE DOMAIN cria um domínio. O domínio é, essencialmente, um tipo de dado com restrições opcionais (restrições no conjunto de valores permitidos). O usuário que cria o domínio se torna o seu dono.

Se for fornecido o nome do esquema (por exemplo, CREATE DOMAIN meu_esquema.meu_dominio ...), então o domínio será criado no esquema especificado, senão será criado no esquema corrente. O nome do domínio deve ser único entre os tipos

e domínios existentes no esquema do domínio.

Domínios são úteis para reunir restrições comuns em campos em um único local para manutenção. Por exemplo, várias tabelas podem conter colunas de endereço de correio eletrônico, todas requerendo a mesma restrição de verificação (CHECK). Em vez de definir as restrições em cada tabela individualmente, pode ser definido um domínio.

```

ALTER DOMAIN nome
  { SET DEFAULT expressão | DROP DEFAULT }
ALTER DOMAIN nome
  { SET | DROP } NOT NULL
ALTER DOMAIN nome
  ADD restrição_de_domínio
ALTER DOMAIN nome
  DROP CONSTRAINT nome_da_restrição [ RESTRICT | CASCADE ]
ALTER DOMAIN nome
  OWNER TO novo_dono

```

Exemplos:

```
ALTER DOMAIN cep SET NOT NULL;
```

```
ALTER DOMAIN cep ADD CONSTRAINT chk_cep CHECK (char_length(VALUE) = 8);
```

Criando:

```

CREATE DOMAIN dom_cep AS text
  CONSTRAINT chk_cep CHECK (VALUE ~ '^\d{8}$') NOT NULL;

```

Exemplos de funções que adicionam e tiram máscaras:

```

-- Recebe assim: 60420440 e exibe assim: 60420-440
CREATE FUNCTION f_cep_tela(cep dom_cep) RETURNS TEXT AS $$ 
BEGIN
  RETURN substr(cep,1,5) || '-' || substr(cep,6,3);
END;
$$ LANGUAGE plpgsql;
CREATE TABLE tbl_cep (cep dom_cep);

-- Recebe assim: 60420-440 e insere assim: 60420440
CREATE FUNCTION f_cep_banco(cep dom_cep) RETURNS TEXT AS $$
```

```

BEGIN
  RETURN substr(cep,1,5) || substr(cep,7,3);
END;
$$ LANGUAGE plpgsql;
CREATE TABLE tbl_cep (cep dom_cep);

```

Estas funções acima são uma pequena variação da função encontrada na documentação oficial em português do comando CREATE DOMAIN:

<http://pgdocptbr.sourceforge.net/pg80/sql-createdomain.html>

Usando:

Para exibir:

```
select cep_tela('60420440');
```

Para inserir no banco, mas usando num insert, ao invés:

```
select cep_banco('60420-440');
```

Obs.: Ao implementar validação através de domínio isso fica transparente para o usuário que geralmente tem que implementar a validação, tornando a programação no aplicativo algo mais leve.

Expressões regulares:

Página da documentação oficial do PostgreSQL:

<http://pgdocptbr.sourceforge.net/pg80/functions-matching.html>

Testador online para expressões POSIX:

<http://www.spaweditor.com/scripts/regex/index.php>

Data (Formato dd/mm/aaaa) - ^([0-9][0,1,2][0-9]|3[0,1])/([\\d]1[0,1,2])\\d{4}\$

Data (Formato aaaa-mm-dd) - ^\\d{4}-(0[0-9]|1[0,1,2])-([0,1,2][0-9]|3[0,1])\$

Hora (HH:MM) - ^([0-1][0-9]|2[0-3]):([0-5][0-9])){1,2}\$

Nome completo - ^[a-zA-Z][a-zA-Z][a-zA-Z]* [a-zA-Z]*\$

Numero Decimal - ^\\d*[0-9](\\.\\d*[0-9])?\$\$

Arquivos - ^[a-zA-Z0-9-_\\.]+.(pdf|txt|doc|csv)\$

Código Cor HTML - ^#?(a-f|[A-F][0-9])\\{3}\\(([a-f]|[A-F][0-9])\\{3})?\$\$ (exemplo: #00ccff

Imagen - ^[a-zA-Z0-9-_\\.]+.(jpg|gif|png)\$

IP - ^((25[0-5]|2[0-4][0-9]|1[0-9]{2}|[0-9]{1,2})\.){3}(25[0-5]|2[0-4][0-9]|1[0-9]{2}|[0-9]{1,2})\$
Arquivos Multimedia - ^[a-zA-Z0-9_\.]+\.(swf|mov|wma|mpg|mp3|wav)\$ (Exemplo:
company-presentation.swf)

CNPJ (com máscara) - ^[0-9]{2}.[0-9]{3}.[0-9]{3}/[0-9]{4}-[0-9]{2}\$ (Exemplo:
00.043.711/0001-43)

CNPJ (sem máscara) - ^[0-9]{2}[0-9]{3}[0-9]{3}[0-9]{4}[0-9]{2}\$ (Exemplo:
00043711000143)

CPF (com máscara) - ^[0-9]{3}.[0-9]{3}.[0-9]{3}-[0-9]{2} (Exemplo: 123.456.789-22)

CPF (sem máscara) - ^[0-9]{3}[0-9]{3}[0-9]{3}[0-9]{2} (Exemplo: 12345678922)

Inscrição Estadual (SP, com máscara) - ^[0-9]{3}.[0-9]{3}.[0-9]{3}.[0-9]{3} (Exemplo:
110.042.490.114)

Inscrição Estadual (SP, com máscara) - ^[0-9]{3}[0-9]{3}[0-9]{3}[0-9]{3} (Exemplo:
110042490114)

Exemplo Ceará: ^[0-9]{3}[0-9]{3}[0-9]{3} (Exemplo: 060000015)

Exemplo Ceará: ^[0-9]{2}.[0-9]{6}-[0-9]{1} (Exemplo: 06.000001-5)

Telefone (Brasil com DDD) - ^\([0-9]\d{2}\)-\d{4}-\d{4}\$ (085)-3423-4542

Telefone (Brasil sem DDD) - ^\d{4}-\d{4}\$ (2634-3454)

Telefone (US) - ^[2-9]\d{2}-\d{3}-\d{4}\$ 250-555-4542

Telefone Internacional ^(([0-9]{1})*- .)*([0-9a-zA-Z]{3})*[- .]*[0-9a-zA-Z]{3}*- .]*[0-9a-zA-Z]{4}+\$ (Exemplo: 1.245.532.3422)

Código Postal (Brasil, sem máscara) - ^[:digit:]{8}\$ 60420440

Código Postal (Brasil, com máscara) ^[0-9]{5}-[0-9]{3}\$ 60420-440

Código Postal (EUA) ^([A-Z][0-9]{3})\$ V2B2S3 Testar

uf CHAR(2) -- unidade da federação

CONSTRAINT chk_uf

CHECK (uf ~ '^A(C|L|M|P)|BA|CE|DF|ES|GO|M(A|G|S|T)|P(A|B|E||R)|R(J|N|O|R|S)|S(C|E|P)|TO\$')

E-mail - ^[a-zA-Z]@\w+\.[a-zA-Z]+\.\w+\.[a-zA-Z]+\$

URL - ^(http[s]?://|ftp://)?(www\.)?[a-zA-Z0-9\.-]+\.(com|org|net|mil|edu|ca|co.uk|com.au|gov|br)\$

=====

Informações Fiscais Úteis:

Consulta de CNPJ online -

http://www.receita.fazenda.gov.br/PessoaJuridica/CNPJ/cnpjreva/Cnpjreva_Solicitacao.aspx

p

Gerador/Validador de CNPJ e CPF online:

<http://highportal.no.sapo.pt/geradorcpf.html>

Obs.: Como é javascript dentro do próprio arquivo HTML, pode ser baixado para uso off line.

SINTEGRA (IE) - <http://www.sintegra.gov.br/>

Exemplo: Ceará - <http://www.sefaz.ce.gov.br/Sintegra/Sintegra.Asp?estado=CE> (Consulta do CNPJ e da IE. Consulte sem máscara)

Busca de CEP:

http://www.correios.com.br/servicos/cep/cep_default.cfm

CNPJ - é único (até as filiais também têm o seu). Órgãos públicos são assim, a sede administrativa tem um CNPJ e suas unidades têm um CNPJ semelhante

IE (Inscrição Estadual) - único, sem nulo, mas quando não existir informar ISENTO (recomendação do SINTEGRA). A formação é própria de cada estado.

IM (Inscrição Municipal) - único, mas é exigido somente para empresas de prestação de serviço e outros específicos.

A máscara do CNPJ é 99.999.999/9999-99. O número do CNPJ mesmo são os 8 dígitos antes da barra e os 4 dígitos seguintes a barra são o número da empresa e os 2 últimos são dígitos verificadores.

Logo em caso de matriz e filiais os 8 primeiros são iguais já que são da mesma empresa e os 4 dígitos seguintes são 0001 para matriz e a sequência são filiais.

Empresas com filiais, usam o mesmo CNPJ, com a diferença de os 4 dígitos finais:

Se a matriz for 00.000.000/0001-00

A filia será 00.000.000/0002-00

Exemplos de CNPJ:

00043711000143

00043711001115

Exemplo de IE:

060000015

Tipos e Domínios

Tipos

```
create type t_humor as enum
(
  'triste','normal','alegre'
);
```

```
create table humores(chave int, humor t_humor);
insert into tipos values(1, 'alegre');
```

```
create type t_sexo as enum
(
  'masculino','feminino'
);
```

```
create table sexos(chave int, sexo t_sexo);
```

```
insert into sexos values(1, lower('Masculino')::tsexo);
select * from sexos;
select chave, initcap(sexo::text) from sexos;
```

Domínios

```
CREATE DOMAIN dom_cep AS text
  CONSTRAINT chk_cep CHECK (VALUE ~ '^\d{8}$') NOT NULL;
-- Um domínio pode ser usado como tipo, com vantagem de ampliar as restrições
-- Veja este: numérico, tamanho 8
```

```
CREATE FUNCTION formata_cep(cep dom_cep) RETURNS TEXT AS $$ 
BEGIN
  RETURN substr(cep,1,5) || '-' || substr(cep,6,3);
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TABLE tbl_cep (cep dom_cep);
insert into tbl_cep values ('60430440');
insert into tbl_cep values (60430440);
insert into tbl_cep values ('123mjhyu');
insert into tbl_cep values ('123456789');
```

```
SELECT formata_cep(CAST('60420440' AS dom_cep)); -- retorna 60420-440
```

```
SELECT formata_cep('60420440');
```

```
CREATE DOMAIN dom_cep_nn AS text
```

CONSTRAINT chk_cep CHECK (VALUE ~ '^\\d{8}\$') NOT NULL;

Exemplo na versão 8.0:

```
create type sexo as
(
    m text,f text
);
```

```
CREATE DOMAIN dsexo AS text
```

```
    CONSTRAINT chk_sexo CHECK (VALUE = 'feminino' OR VALUE = 'masculino');
```

```
create table tab_sexo(s dsexo, sx sexo);
```

```
insert into tab_sexo values('feminino', ('d','g'));
insert into tab_sexo values('feminino', ('d',8));
insert into tab_sexo values('masculino', ('d',8));
insert into tab_sexo values('dará erro', ('d',8));
```

Regras de Integridade

Garantem que alterações no banco de dados não resultam em perda da consistência dos dados.

As regras:

- chaves
- restrições de tipos e de domínios

A cláusula check permite modos poderosos de restrições para os domínios.

Restringindo um Domínio a um Conjunto de Valores:

```
create domain d_noma char(5)
constraint chk_nome check(value in('João', 'Pedro'));
```

Integridade Referencial

Garante que um mesmo valor apareça em duas relações.

Tradução livre do documentação "CBT Integrity Referential":

http://techdocs.postgresql.org/college/002_referentialintegrity/

Integridade Referencial (relacionamento) é onde uma informação em uma tabela se refere à informações em outra tabela e o banco de dados reforça a integridade.

Onde é Utilizado?

Onde pelo menos em uma tabela precisa se referir para informações em outra tabela e ambas precisam ter seus dados sincronizados.

Exemplo: uma tabela com uma lista de clientes e outra tabela com uma lista dos pedidos efetuados por eles.

Com integridade referencial devidamente implantada nestas tabelas, o banco irá garantir que você nunca irá cadastrar um pedido na tabela pedidos de um cliente que não exista na tabela clientes.

O banco pode ser instruído para automaticamente atualizar ou excluir entradas nas tabelas quando necessário.

Primary Key (Chave Primária) - é o campo de uma tabela criado para que as outras tabelas relacionadas se refiram a ela por este campo. Impede mais de um registro com valores iguais. É a combinação interna de UNIQUE e NOT NULL.

Qualquer campo em outra tabela do banco pode se referir ao campo chave primária, desde que tenham o mesmo tipo de dados e tamanho da chave primária.

Exemplo:

clientes (codigo INTEGER, nome_cliente VARCHAR(60))

codigo nome_cliente
1 PostgreSQL inc.
2 RedHat inc.

pedidos (relaciona-se à Clientes pelo campo cod_cliente)
cod_pedido cod_cliente descricao

Caso tentemos cadastrar um pedido com cod_cliente 2 ele será aceito.

Mas caso tentemos cadastrar um pedido com cod_cliente 3 ele será recusado pelo banco.

Criando uma Chave Primária

Deve ser criada quando da criação da tabela, para garantir valores exclusivos no campo.

```
CREATE TABLE clientes(cod_cliente BIGINT, nome_cliente VARCHAR(60)
PRIMARY KEY (cod_cliente));
```

Criando uma Chave Estrangeira (Foreign Keys)

É o campo de uma tabela que se refere ao campo Primary Key de outra. O campo pedidos.cod_cliente refere-se ao campo clientes.codigo, então pedidos.cod_cliente é uma chave estrangeira, que é o campo que liga esta tabela a uma outra.

```
CREATE TABLE pedidos(
cod_pedido BIGINT,
cod_cliente BIGINT REFERENCES clientes,
descricao VARCHAR(60)
);
```

Outro exemplo:

```
FOREIGN KEY (campoa, campob)
REFERENCES tabela1 (campoa, campob)
ON UPDATE CASCADE
ON DELETE CASCADE);
```

Cuidado com exclusão em cascata. Somente utilize com certeza do que faz.

Dica: Caso desejemos fazer o relacionamento com um campo que não seja a chave primária, devemos passar este campo entre parênteses após o nome da tabela e o mesmo deve obrigatoriamente ser UNIQUE.

```
...
cod_cliente BIGINT REFERENCES clientes(nomecampo),
...
```

Parâmetros Opcionais:

ON UPDATE parametro e ON DELETE parametro.

ON UPDATE paramentros:

NO ACTION (RESTRICT) - quando o campo chave primária está para ser atualizado a atualização é abortada caso um registro em uma tabela referenciada tenha um valor mais antigo. Este parâmetro é o default quando esta cláusula não recebe nenhum parâmetro.

Exemplo: ERRO Ao tentar usar:

UPDATE clientes SET codigo = 5 WHERE codigo = 2.

Ele vai tentar atualizar o código para 5 mas como em pedidos existem registros do cliente 2 haverá o erro.

CASCADE (Em Cascata) - Quando o campo da chave primária é atualizado, registros na tabela referenciada são atualizados.

Exemplo: Funciona: Ao tentar usar "UPDATE clientes SET codigo = 5 WHERE codigo = 2.

Ele vai tentar atualizar o código para 5 e vai atualizar esta chave também na tabela pedidos.

SET NULL (atribuir NULL) - Quando um registro na chave primária é atualizado, todos os campos dos registros referenciados a este são setados para NULL.

Exemplo: UPDATE clientes SET codigo = 9 WHERE codigo = 5;

Na clientes o codigo vai para 5 e em pedidos, todos os campos cod_cliente com valor 5 serão setados para NULL.

SET DEFAULT (assumir o Default) - Quando um registro na chave primária é atualizado, todos os campos nos registros relacionados são setados para seu valor DEFAULT.

Exemplo: se o valor default do código de clientes é 999, então UPDATE clientes SET código = 10 WHERE código = 2. Após esta consulta o campo código com valor 2 em clientes vai para 999 e também todos os campos cod_cliente em pedidos.

ON DELETE parametros:

NO ACTION (RESTRICT) - Quando um campo de chave primária está para ser deletado, a exclusão será abortada caso o valor de um registro na tabela referenciada seja mais velho.

Este parâmetro é o default quando esta cláusula não recebe nenhum parâmetro.

Exemplo: ERRO em DELETE FROM clientes WHERE código = 2. Não funcionará caso o cod_cliente em pedidos contenha um valor mais antigo que código em clientes.

CASCADE - Quando um registro com a chave primária é excluído, todos os registros relacionados com aquela chave são excluídos.

SET NULL - Quando um registro com a chave primária é excluído, os respectivos campos na tabela relacionada são setados para NULL.

SET DEFAULT - Quando um registro com a chave primária é excluído, os campos respectivos da tabela relacionada são setados para seu valor DEFAULT.

Excluindo Tabelas Relacionadas

Para excluir tabelas relacionadas, antes devemos excluir a tabela com chave estrangeira.

PK 1 -----> N FK

Do lado 1 é exigida uma PK ou uma constraint UNIQUE.

Lado 1 não permite nulos.

Lado N permite nulos mas se existir a integridade garantida.

PK (Chave Primária) - é formada internamente por UNIQUE e NOT NULL

UNIQUE (Chave candidata) - permite nulos

FK (Chave Estrangeira) - permite nulos, mas se um campo for nulo estará satisfeita a constraint em consequência em consequência violada a integridade.

Recomendação: sempre usar NOT NULL nos campos da FK.

Dependência Funcional

É uma generalização da noção de chaves. São restrições ao conjunto de relações válidas.

Modos:

- Restrições sobre o conjunto de relações.
- Para verificação de relações.

Dependência de Existência - quando a existência de uma entidade depende de outra.

Chaves:

- primária – Uma chave composta por um ou mais campos e que não repete em nenhum registro. Formata internamente pela constraint UNIQUE e a NOT NULL.

- [Chave Estrangeira](#)

Uma Chave Estrangeira é uma Chave de Relacionamento ou seja ela tem como propósito representar os Relacionamentos entre Tabelas num BDR.

A chave estrangeira de uma tabela pode ser definida como um conjunto de atributos da tabela (um ou mais atributos) com a propriedade de estabelecer relacionamento entre linhas de tabelas.

Uma chave estrangeira corresponde sempre a uma chave primária previamente definida em alguma tabela.

Em outras palavras um valor de chave estrangeira sempre aponta para um valor de chave primária previamente existente no banco de dados. Essa restrição é denominada de “Restrição de Integridade Referencial”.

Uma chave estrangeira pode conter valor nulo.

Chave Estrangeira é o mecanismo através do qual o Modelo de Dados Relacional implementa relacionamentos entre tabelas. Não constitui um atributo da entidade. Só aparece no momento do projeto físico.

- candidata – Chave criada com a adição da constraint UNIQUE em um campo.
- super – uma chave que contém mais campos que o necessário para garantir que seja PK.
- artificial – uma chave formada por um campo que nada representa para a tabela, como um ID, registro, código, etc.
- natural – uma chave formada por um ou mais campos que de fato representam todos os registros de uma tabela.

Uma chave é uma propriedade do conjunto de entidades e não de uma entidade específica.

Um documento com boas informações sobre modelagem e normalização:

Modelagem e Administração de Dados em PostgreSQL

Fundamentos e práticas em bases de dados livres

De Leandro Guimarães Faria Corcete DUTRA (para a Conferência PostgreSQL Brasil)

Em: <http://leandro.gfc.dutra.googlepages.com/adpg.art.pdf>

Dicas de Modelagem de Dados

Projeto

Na criação das tabelas podemos mudar algumas de suas características para tornar as mesmas mais eficientes para armazenar dados e os dados corretos.

Um dos grandes desafios em se projetar um banco de dados com sucesso é a correta determinação das tabelas que comporão o banco de dados, bem como de cada campo que comporão as tabelas e suas constraints e relacionamentos entre as tabelas.

1aFN - todos os campos possuem valores simples e não são duplicados nem composto

Exemplo errado: clientes (id, endereço, telefone). Solução: criar outras tabelas.

2aFN - deve estar na 1FN e todos os campos dependem integralmente da chave primária

Exemplo errado: professores (cpf, curso_id, salario, curso_descricao). curso_descricao deve ser removido e criar outra tabela com ele e relacioná-la com professores.

3aFN - deve estar na 2FN e nenhum campo pode depender de outro campo que não seja a PK).

Chave Primária

Uma chave primária é a combinação interna de UNIQUE e NOT NULL. Exige que todos os registros tenham o valor do campo chave primária diferentes.

Toda tabela deve ter uma chave primária.

Bons exemplos de campos com vocação para chave primária:

- Matrícula de funcionário
- BM de móveis e imóveis
- RG
- CPF
- Matrícula de alunos
- Código de peça em tabela de cadastro de peças

Após ter definido um campo como chave primária o próprio SGBD não permitirá valores duplicados no campo.

Algumas vezes a chave primária precisa ser formada por dois ou mais campos. Devemos evitar isso, mas quando for necessário devemos suar. Quando acontecer de um único campo não ser suficiente para a unicidade.

Também as chaves UNIQUE podem precisar ser compostas por dois ou mais campos.

Campos requeridos devem ter a constraint Not Null.

Relacionamentos

Quando relacionamos duas tabelas, em uma fica a informação e na outra fica apenas uma referência para a informação. De forma que quando precisamos da informação usamos a referência para consultar a informação.

Chave Estrangeira ou Foreign Key

É o campo de uma tabela que se refere ao campo Primary Key de outra. No caso de materiais e contas, o conta_id de materiais se refere ao campo id de contas.

Para relacionar duas tabelas num relacionamento um para vários, devemos relacionar um campo de uma tabela com a chave primária da outra. Os campos relacionados devem ser do mesmo tipo e do mesmo tamanho.

Dica: podemos relacionar uma tabela com outra sem ser pelo campo chave primária, mas no caso o campo da tabela estrangeira deve ser UNIQUE e NOT NULL.

PK 1 ----> N FK
UNIQUE NN 1 ----> N FK

Exemplo: Materiais com Contas. Estão relacionadas através do campo conta_id em materiais e id em contas. Em contas fica a informação desejada em materiais, que é grupo. Através do campo relacionado podemos trazer para materiais qualquer campo de contas.

Sempre que uma tabela cresce muito idealmente devemos dividi-la em duas ou mais e relacionar as tabelas resultantes.

Também quando temos um caso de relacionamentos muitos para muitos devemos criar uma terceira tabela e relacioná-la com as outras duas.

Exemplo: pedidos e produtos. Temos muitos pedidos relacionados com muitos produtos.

Precisamos de uma tabela intermediária de detalhes do pedido, para que fique assim:

pedidos 1 ----> N pedido_detalhes N <---- 1 produtos

Dica: o tipo de dados dos campos relacionados obrigatoriamente devem ser o mesmo e do mesmo tamanho: inteiro com inteiro, texto com texto, etc. Exemplo: pedidos com pedido_detalhes, em pedido_detalhes terá um campo chamado pedido_id que se relacionará com o campo id de pedidos. Então ambos devem ser inteiro.

Integridade Referencial

Após implementar o relacionamento entre clientes e pedidos, o banco de dados não permitirá que seja cadastrado um pedido para um cliente que ainda não foi cadastrado, nem excluir um cliente que ainda tenha pedidos cadastrados em seu nome.

O banco pode receber instruções nossas para automaticamente atualizar ou excluir registros nas tabelas relacionadas.

Quando o código de um cliente for alterado é interessante que atualizemos automaticamente este código na tabela pedidos em todos os seus pedidos. Para fazer isso devemos usar ON UPDATE CASCADE na chave estrangeira.

Assim como quando um cliente for excluído que sejam excluídos todos os seus pedidos. Faz-se isso usando na chave estrangeira ON DELETE CASCADE. Mas isso deve ser pensado com cuidado e quem cria o banco de dados deve perguntar ao seu cliente ou ao responsável pelo banco ou sistema o que deve fazer. Se quisermos manter um histórico não devemos excluir, então usamos ao invés ON DELETE RESTRICT.

Geralmente devemos relacionar as tabelas do banco de dados. Não tem muito sentido ter tabelas não relacionadas. Não caracteriza bancos relacionais.

Ações em Relacionamentos

Comando	Ação
NO ACTION	Ação default; UPDATE e DELETE não serão executados para proteger a integridade referencial.
CASCADE	Todas as FK devem ser atualizadas quando a PK mudar (com ON UPDATE). Todas as FK devem ser deletadas quando a PK for deletada (ON DELETE).
SET NULL	A FK é setada para NULL quando a PK for atualizada ou deletada.
SET DEFAULT	A FK é setada para o valor DEFAULT do campo quando a PK for atualizada ou deletada.

Dicas diversas:

- Atributos Multivalorados (telefones, municipio, uf, etc) indicam necessidade de criação de outra tabela.
- A repetição de valores de atributos também deve ser evitada criando-se uma outra tabela.
- A presença de nulos também é resolvida com a normalização e consequente criação de outras tabelas.
- Não salvar em tabelas campo calculados. Exemplo: temos uma tabela pedidos, onde tem os campos quantidade e preco. Não adicione o campo total. Total deve ser fruto da operação em uma consulta, que multiplica quantidade por preco.

Cuidado com chaves artificiais, aquelas que não representam a tabela. Um exemplo é o id. Sempre que usar id como chave primária tenha outro campo que seja unique.

```
create table artificial(id serial primary key, campo1 text, campo2 text);
insert into artificial(campo1,campo2) values ('a','b'),('a','b'),('a','b'),('a','b'),('a','b');
select * from artificial;
c1 | campo1 | campo2
----+----+----
```

```
1 | a | b
2 | a | b
3 | a | b
4 | a | b
5 | a | b
6 | a | b
```

Veja que todos os registros são duplicados, mudando somente o valor do id, o que não garante unicidade. O SGBD apenas se preocupa em fazer o que você mandou, não duplicar o id.

Lembrando de materiais e contas, idealmente código deve ser a PK de materiais e grupo deve ser a PK de contas. Para facilitar nossa vida e ainda assim garantir unicidade deixemos a PK sendo id nas tabelas e mudemos o campo código de materiais para unique e not null. E em contas grupo de ve ser unique e not null.

Exemplos de tabelas que permitem erroneamente campos com valor duplicado:
pessoas (cpf, nome, rua, numero, bairro, cidade, uf, cep)

```
rua
bairro
cidade
uf
cep
```

Todos podem ser duplicados. Idealmente devemos criar uma tabela para cada um e relacioná-las com pessoas.

Dicas sobre o SGBD livre PostgreSQL

CREATE DATABASE nomebanco;

Scrips

MySQL ou PostgreSQL (este script funciona nos dois)

```
create table clientes
(
    cliente int primary key,
    cpf char(11),
    nome char(45) not null,
    credito_liberado char(1) not null,
    data_nasc date,
    email varchar(50)
```

```

);
create table funcionarios
(
    funcionario int primary key,
    cpf char(11),
    nome char(45) not null,
    senha char(32) not null,
    email varchar(50),
    data_nasc date not null
);
create table pedidos
(
    pedido int primary key,
    cliente_id int not null,
    funcionario_id int not null,
    data_pedido date not null,
    data_confirmacao date not null,
    FOREIGN KEY (cliente_id) REFERENCES clientes(cliente) ON DELETE RESTRICT,
    FOREIGN KEY (funcionario_id) REFERENCES funcionarios(funcionario) ON DELETE RESTRICT
);

```

Outra forma: constraint pedidos_cli_fk foreign key(cliente_id) REFERENCES clientes (cliente)

ALTER DATABASE

ALTER TABLE pedidos ADD CONSTRAINT pedidos_cli_fk FOREIGN KEY (cliente_id) REFERENCES clientes (cliente);

ALTER TABLE clientes ADD CHECK (length(cpf)=11);

ALTER TABLE estoque ALTER COLUMN data SET DEFAULT CURRENT_DATE;

Dicas Avançadas:

```

SELECT CASE WHEN 10*2=30 THEN '30 incorrect'
            WHEN 10*2=40 THEN '40 incorrect'
            ELSE 'Deve ser 10*2=20'
END;

```

```

SELECT CASE 10*2
        WHEN 20 THEN '20 correct'
        WHEN 30 THEN '30 incorrect'
        WHEN 40 THEN '40 incorrect'
END;

```

```
SELECT CASE WHEN 9>7
            THEN "TRUE"
            ELSE
            "FALSE"
            END
        AS "Resultado";
```

Trazer somente 1 registro posterior ao valor fornecido

```
SELECT id, title FROM topics WHERE id > $currentTopicId ORDER BY id ASC LIMIT 1
```

```
SELECT cliente, nome FROM clientes WHERE cliente > 1 ORDER BY cliente ASC LIMIT 1
```

Trazer somente 1 registro anterior ao valor fornecido

```
SELECT id FROM entrada_dados WHERE id < 10 ORDER BY id DESC LIMIT 1
SELECT data,total FROM entrada_dados WHERE data < '2009-07-01' ORDER BY total DESC LIMIT 1
```

Com outros parâmetros no WHERE

```
SELECT id, title FROM topics WHERE id > $currentTopicId and visible=1 ORDER BY id ASC LIMIT 1
```

Último registro

```
SELECT id FROM tab_pecas WHERE id = (SELECT MAX id from tab_pecas)
```

```
SELECT nm_dept FROM departamento ORDER BY codigo DESC LIMIT 1
```

```
SELECT entidades FROM entrada_dados ORDER BY id DESC LIMIT 1
```

```
select * from employee1 order by empid,name,dob asc limit 1;
```

```
SELECT LAST_INSERT_ID()
```

Datas

```
SELECT CURRENT_DATE();
```

```
sudo su
su - postgres
createdb joomlapg
createuser joomlapg
psql joomlapg
```

```
ALTER DATABASE joomlapg OWNER TO joomlapg;
```

```
alter role joomlapg password 'joomlapg';
```

```
REVOKE CONNECT ON DATABASE joomlapg FROM PUBLIC;
```

```
GRANT CONNECT ON DATABASE joomlapg TO joomlapg;
```

Integridade Referencial

Garante que um mesmo valor apareça em duas relações.

Tradução livre da documentação "CBT Integrity Referential":
http://techdocs.postgresql.org/college/002_referentialintegrity/

Integridade Referencial (relacionamento) é onde uma informação em uma tabela se refere à informações de outras tabelas e o banco de dados reforça sua integridade.

Onde é Utilizado?

Onde pelo menos em uma tabela precisa se referir para informações em outra tabela e ambas precisam ter seus dados sincronizados.

Exemplo: uma tabela com uma lista de clientes e outra tabela com uma lista dos pedidos efetuados por eles.

Com integridade referencial devidamente implantada nestas tabelas, o banco irá garantir que você nunca irá cadastrar um pedido na tabela pedidos de um cliente que não existe na tabela clientes.

O banco pode ser instruído para automaticamente atualizar ou excluir entradas nas tabelas quando necessário.

Primary Key (Chave Primária) - é o campo de uma tabela criado para que as outras tabelas relacionadas se refiram a ela por este campo. Impede mais de um registro com valores iguais. É a combinação interna de UNIQUE e NOT NULL.

Qualquer campo em outra tabela do banco pode se referir ao campo chave primária, desde que tenham o mesmo tipo de dados e tamanho da chave primária.

Exemplo:

clientes (codigo INTEGER, nome_cliente VARCHAR(60))

codigo nome_cliente

1 PostgreSQL inc.

2 RedHat inc.

pedidos (relaciona-se à Clientes pelo campo cod_cliente)

cod_pedido cod_cliente descricao

Caso tentemos cadastrar um pedido com cod_cliente 2 ele será aceito.

Mas caso tentemos cadastrar um pedido com cod_cliente 3 ele será recusado pelo banco.

Criando uma Chave Primária

Deve ser criada quando da criação da tabela, para garantir valores exclusivos no campo.

```
CREATE TABLE clientes(cod_cliente BIGINT, nome_cliente VARCHAR(60)
PRIMARY KEY (cod_cliente));
```

Criando uma Chave Estrangeira (Foreign Keys)

É o campo de uma tabela que se refere ao campo Primary Key de outra.
O campo pedidos.cod_cliente refere-se ao campo clientes.codigo, então
pedidos.cod_cliente é uma chave estrangeira, que é o campo que liga esta tabela a uma
outra.

```
CREATE TABLE pedidos(
cod_pedido BIGINT,
cod_cliente BIGINT REFERENCES clientes,
descricao VARCHAR(60)
);
```

Outro exemplo:

```
FOREIGN KEY (campo1, campo2)
REFERENCES tabela1 (campo1, campo2)
ON UPDATE CASCADE
ON DELETE CASCADE);
```

Cuidado com exclusão em cascata. Somente utilize com certeza do que faz.
Dica: Caso desejemos fazer o relacionamento com um campo que não seja a chave
primária, devemos passar este campo entre parênteses após o nome da tabela e o
mesmo deve obrigatoriamente ser UNIQUE.

...
cod_cliente BIGINT REFERENCES clientes(nomecampo),

Parâmetros Opcionais:

ON UPDATE parametro e ON DELETE parametro.
ON UPDATE paramentros:

NO ACTION (RESTRICT) - quando o campo chave primária está para ser atualizado a
atualização é abortada caso um registro em uma tabela referenciada tenha um valor mais
antigo. Este parâmetro é o default quando esta cláusula não recebe nenhum parâmetro.

Exemplo: ERRO Ao tentar usar:

UPDATE clientes SET codigo = 5 WHERE codigo = 2.

Ele vai tentar atualizar o código para 5 mas como em pedidos existem registros do cliente
2 haverá o erro.

CASCADE (Em Cascata) - Quando o campo da chave primária é atualizado, registros na
tabela referenciada são atualizados.

Exemplo: Funciona: Ao tentar usar "UPDATE clientes SET codigo = 5 WHERE codigo = 2.

Ele vai tentar atualizar o código para 5 e vai atualizar esta chave também na tabela pedidos.

SET NULL (atribuir NULL) - Quando um registro na chave primária é atualizado, todos os campos dos registros referenciados a este são setados para NULL.

Exemplo: UPDATE clientes SET codigo = 9 WHERE codigo = 5;

Na clientes o codigo vai para 5 e em pedidos, todos os campos cod_cliente com valor 5 serão setados para NULL.

SET DEFAULT (assumir o Default) - Quando um registro na chave primária é atualizado, todos os campos nos registros relacionados são setados para seu valor DEFAULT.

Exemplo: se o valor default do codigo de clientes é 999, então UPDATE clientes SET codigo = 10 WHERE codigo = 2. Após esta consulta o campo código com valor 2 em clientes vai para 999 e também todos os campos cod_cliente em pedidos.

ON DELETE parametros:

NO ACTION (RESTRICT) - Quando um campo de chave primária está para ser deletado, a exclusão será abortada caso o valor de um registro na tabela referenciada seja mais velho.

Este parâmetro é o default quando esta cláusula não recebe nenhum parâmetro.

Exemplo: ERRO em DELETE FROM clientes WHERE codigo = 2. Não funcionará caso o cod_cliente em pedidos contenha um valor mais antigo que codigo em clientes.

CASCADE - Quando um registro com a chave primária é excluído, todos os registros relacionados com aquela chave são excluídos.

SET NULL - Quando um registro com a chave primária é excluído, os respectivos campos na tabela relacionada são setados para NULL.

SET DEFAULT - Quando um registro com a chave primária é excluído, os campos respectivos da tabela relacionada são setados para seu valor DEFAULT.

Excluindo Tabelas Relacionadas

Para excluir tabelas relacionadas, antes devemos excluir a tabela com chave estrangeira.

PK 1 -----> N FK

Do lado 1 é exigida uma PK ou uma constraint UNIQUE.

Lado 1 não permite nulos.

Lado N permite nulos mas se existir a integridade garantida.

PK (Chave Primária) - é formada internamente por UNIQUE e NOT NULL
 UNIQUE (Chave candidata) - permite nulos

FK (Chave Estrangeira) - permite nulos, mas se um campo for nulo estará satisfeita a constraint em consequência em consequência violada a integridade.
Recomendação: sempre usar NOT NULL nos campos da FK.

Ação

Comando	Ação
NO ACTION	Ação default; UPDATE e DELETE não serão executados para proteger a integridade referencial.
CASCADE	Todas as FK devem ser atualizadas quando a PK mudar (com ON UPDATE). Todas as FK devem ser deletadas quando a PK for deletada (ON DELETE).
SET NULL	A FK é setada para NULL quando a PK for atualizada ou deletada.
SET DEFAULT	A FK é setada para o valor DEFAULT do campo quando a PK for atualizada ou deletada.

Algumas Demonstrações sobre Normalização

Valor default

Chaves naturais x artificiais

Null

Default

```
create table nula(c1 serial primary key, c2 int, c3 int default 0);
```

```
insert into nula (c1) values (default),(default),(default),(default);
```

```
select * from nula;
```

```
c1 | c2 | c3
```

```
----+---+---
```

```
1 || 0
```

```
2 || 0
```

```
3 || 0
```

```
4 || 0
```

```
(4 registros)
```

Veja só que “riqueza” de registros! Tudo isso graças a permissão de nulo e ao valor default.

Nulo

```
create table nula2(c1 int primary key, c2 int check(c2 > 0), c3 int);
insert into nula2(c1,c2,c3) values (1,default,4); -- Será válido. Importante: use not null
insert into nula2(c1,c2,c3) values (2,-3,4)
select * from nula2;
c1 | c2 | c3
----+---+---
1 | | 4
(1 registro)
```

Uma "incoerência" no comportamento do nulo, que reforça a recomendação de se evitar seu uso.

Chave artificial

```
create table artificial(c1 serial primary key, t1 text, t2 text);
insert into artificial(t1,t2) values ('a','b'),('a','b'),('a','b'),('a','b'),('a','b'),('a','b');
select * from artificial;
c1 | t1 | t2
----+---+---
1 | a | b
2 | a | b
3 | a | b
4 | a | b
5 | a | b
6 | a | b
(6 registros)
```

Este ganha dos demais, em minha opinião. O cara cria uma chave tipo ID, que ela é a única coisa que não pode ser duplicada.

Então veja que todos os registros estão duplicados, pois a responsabilidade do SGBD é apenas a de não duplicar o campo c1.

Sugestão: para tabelas secundárias (daquelas com código e descrição), sempre usar a restrição UNIQUE no campo descrição.

Para tabelas primárias sempre que possível use chaves naturais.

Normalizando Tabelas com Leandro Dutra

- Atributos Multivalorados (telefones, municipio, uf, etc) indicam necessidade de criação de outra tabela.
- A repetição de valores de atributos também deve ser evitada criando-se uma outra tabela.
- A presença de nulos também é resolvida com a normalização e consequente criação de outras tabelas.
- O modelo relacional foi fundamentado na teoria dos conjuntos e na lógica dos predicados.

Se fosse buscar por inspiração no sentido mais exato, diria que era na crise de software: facilitar o desenvolvimento de grandes bases de dados usadas simultaneamente por muitos usuários.

- Seus termos principais são: relações, atributos, restrições e tipos!
- Relacionamento não é um termo técnico deste modelo, mas do MER, aqui temos as restrições de integridade referencial.
- A linguagem SQL não é inteiramente relacional. SGBD relacionais restringem o modelo para usarem essa linguagem.

Por isso mesmo não os chamo de SGBDRs, mas de SGBDs SQL. Os SGBDRs que conheço são o IBM BS/12, o Ingres QUEL original, o Alphora Dataphor e outros atualmente em desenvolvimento, anteriormente listados.

- É bom distinguir MR (modelo relacional) de MER (modelo entidade relacionamento). Este último surgiu depois do relacional, mas a grande maioria dos SGBDs atuais implementam o modelo relacional ou uma versão adaptada dele.

Em princípio, todas as chaves naturais são boas, e todas as artificiais são ruins.

O caso é que tem muita gente, principalmente usuários de ORMs, que não gosta de chaves compostas. Acha que fica difícil programar. Eu nunca usei ORM, nunca senti na pele, e daí vem minha impressão de que muitos ORMs criam tantos problemas quanto resolvem.

Imagine uma relação pai com uma chave natural composta razoável, digamos mais de três atributos; e uma relação filha com muito mais tuplas, digamos milhões ou dezenas de milhões a mais,

isso rodando num sistema que é gargalo de desempenho. Nesse caso, pode ser que, após testes, valha a pena uma chave primária artificial para emagrecer a tabela filha.

(Trechos de respostas na lista pgbr-geral, por Leandro Dutra)

Modelo Relacional segundo C. J. Date

- Um modelo de dados é uma definição abstrata, lógica, dos objetos, operadores (funções) e demais.
- Objetos são usados para modelar a estrutura de dados
- Operadores são usados para modelar o comportamento dos dados
- A implementação de um modelo é a realização física na máquina real.
- Diagramas E/R não representam tudo que interessa dos relacionamentos entre as entidades.
- Uma relação que contém nulos não é uma relação.
- Um modelo relacional com nulos não é um modelo relacional.
- Nulos são um erro e nunca deveriam ter sido adotados.
- Toda função é um operador mas nem todo operador é uma função.
- Relações - Atributos - Tipos

Correspondência de termos entre os modelos lógico e físico:

- Relações - Tabelas
- Tupla - Registros
- Atributos - Campos ou colunas
- Modelo - Implementação

As quatro Operações das Relações:

- Relações são normalizadas
- Atributos são desordenados, da esquerda para a direita
- Tuplas são desordenadas, de cima para baixo
- Não existem tuplas duplicadas

Dicionário de dados - dados sobre dados - metadados (catálogo do sistema no PostgreSQL).

Esta última subentende chave natural em todas as tabelas. Mas se os SGBDs implementarem assim, como vamos vazer nossos testes e demonstrações? (nota)

- Valores de atributos são valores simples, mas esses valores podem ser absolutamente qualquer coisa. Nós rejeitamos a antiga noção de "valor atômico".

Ou seja, justificando tipos como geométricos, arrays, etc. (nota)

- A cardinalidade de um conjunto é seu número de elementos e de uma relação é seu número de tuplas.
- O projeto de um banco de dados tem mais de arte que de ciência.

DER

Cuidado com diagramas, use-os apenas para ajudar no processo do projeto, lembrando que um DER não faz todo o trabalho de modelagem e nem representa tudo de um modelo.

(CJ Date em seu livro: An Introduction To Database Systems)

Fases de um Projeto de Banco de Dados

Projeto de Banco de Dados Relacional

Objetivo - gerar um banco de dados que permita armazenar informações sem redundância e recuperá-las com facilidade.

Fases de um Projeto

Na modelagem de um banco de dados devemos primeiramente ter conhecimento do ambiente real ao qual será aplicado o modelo. Durante o processo de modelagem deverá ser usado o bom senso e as regras inerentes ao ambiente de aplicação do banco de dados.

1) Levantamento de Requisitos

Um modelo de dados de alto nível oferece ao projetista conceitos que o possibilitam especificar as necessidades dos usuários e como o banco será estruturado para atender plenamente todas as necessidades. Aqui são importantes as entrevistas e a avaliação do projetista.

Resultado dessa fase - Especificação das necessidades dos usuários (levantamento de requisitos).

2) Projeto Conceitual

A fase conceitual depende muito da habilidade do projetista e das qualidades do modelo de dados adotado.

Escolha do modelo de dados, para com ele transcrever as necessidades e informações coletadas para um esquema de banco de dados.

O projeto conceitual indicará as necessidades funcionais da empresa, as consultas, exclusões, etc.

Então deve-se rever o esquema dos dados para adequar às necessidades funcionais. O projeto conceitual gera o esquema conceitual. No projeto conceitual não se leva em conta o SGBD que será utilizado.

O propósito do projeto conceitual é descrever o conteúdo de informação do banco de dados ao invés das estruturas de armazenamento.

3) Projeto Lógico

Tem por objetivo avaliar o esquema conceitual frente às necessidades de uso do banco de dados pelos usuários e aplicações, realizando possíveis refinamentos com a finalidade de melhorar o desempenho das operações.

Um esquema lógico é uma descrição da estrutura do banco de dados que pode ser processada por um SGBD.

Depende do modelo de dados adotado pelo SGBD, mas não especificamente do SGBD.

Neste mapeamos o modelo conceitual para o modelo de implementação (físico).

O projeto lógico gera o esquema lógico.

4) Projeto Físico

Este toma por base o esquema lógico para gerar o esquema físico e é direcionado para um específico SGBD.

O projeto físico gera o esquema físico.

Fases práticas:

- Análise de requisitos
- Identificação das relações e atributos
- Identificar as chaves das relações
- Analisar as pendências anteriores e aprofundar a modelagem
- Focar nos atributos, seus tipos, domínios e constraints. Substituir multivalorados, repetidos e nulos
- Gerar relacionamentos

Um Bom Projeto de Banco de Dados Evita:

- Inconsistência e redundância
- Dificuldade de acesso pela falta de planejamento
- Isolamento de dados
- Problemas de integridade
- Problema na falta de atomicidade nas transações
- Anomalias no acesso concorrente
- Problemas de segurança
- Operações entre disco e memória (minimizar)

Para normalizar as informações precisamos colher informações detalhadas sobre a empresa real para a qual iremos modelar o banco de dados.

O que evitar?

- informações repetidas
- dificuldade na recuperação de informações

Repetições

- desperdiçam espaço
- dificultam (engessam) atualizações

Exemplo:

Temos um cadastro de clientes assim:

(nome, fone, numero, rua, bairro, cidade, uf).

Com uma grande quantidade de registros, caso sofra alteração em algum dos campos multivalorados, como uf, teremos que atualizar todos os registros dos clientes.

Caso normalizemos a tabela de clientes e ufs sejam uma tabela separada apenas se relacionando com clientes, ao alterar uma uf apenas atualizaremos um registro na tabela ufs, sem contar que não cadastraremos a uf em cada cliente, mas apenas uma vez na tabela ufs.

Decomposição

Quando decomponemos uma relação em várias, devemos ter cuidado para que não haja perda de informações nas dependências.

Tipos e Domínios

A criação de novos tipos e domínios reforçam e muito a robustez dos bancos de dados. Podemos criar um tipo que seja mais adequado para nosso campo e depois, criar um domínio em cima desse tipo para que seja reforçada a restrição, usando-se constraints como CHECK, NOT NULL, UNIQUE e outras. Veja exemplos.

Tipos

```
create type t_humor as enum(
'triste','normal','alegre'
);
create table humores(chave int, humor t_humor);
insert into tipos values(1, 'alegre');
```

```
create type t_sexo as enum(
'masculino','feminino'
);
create table sexos(chave int, sexo t_sexo);
insert into sexos values(1, lower('Masculino')::tsexo);
select * from sexos;
select chave, initcap(sexo::text) from sexos;
```

Domínios

```

CREATE DOMAIN dom_cep AS text
CONSTRAINT chk_cep CHECK (VALUE ~ '^\d{8}$') NOT NULL;
-- Um domínio pode ser usado como tipo, com vantagem de ampliar as restrições
-- Veja este: numérico, tamanho 8
CREATE FUNCTION formata_cep(cep dom_cep) RETURNS TEXT AS $$ 
BEGIN
RETURN substr(cep,1,5) || '-' || substr(cep,6,3);
END;
$$ LANGUAGE plpgsql;

```

```

CREATE TABLE tbl_cep (cep dom_cep);
insert into tbl_cep values ('60430440');
insert into tbl_cep values (60430440);
insert into tbl_cep values ('123mjhyu');
insert into tbl_cep values ('123456789');

```

```
SELECT formata_cep(CAST('60420440' AS dom_cep)); -- retorna 60420-440
```

```
SELECT formata_cep('60420440');
```

```

CREATE DOMAIN dom_cep_nn AS text
CONSTRAINT chk_cep CHECK (VALUE ~ '^\d{8}$') NOT NULL;

```

Exemplo na versão 8.0:

```

create type sexo as
(
m text,f text
);
CREATE DOMAIN dsexo AS text
CONSTRAINT chk_sexo CHECK (VALUE = 'feminino' OR VALUE = 'masculino');
create table tab_sexo(s dsexo, sx sexo);
insert into tab_sexo values('feminino', ('d','g'));
insert into tab_sexo values('feminino', ('d',8));
insert into tab_sexo values('masculino', ('d',8));
insert into tab_sexo values('dará erro', ('d',8));

```

41 - Melhorando a Performance do PostgreSQL

O Que Esperar do Tuning do SGBD

Tunning de banco de dados não é nenhuma panacéia.

Antes que o Tuning do SGBD possa trazer resultados concretos, é necessário que haja cuidado

na:

Modelagem conceitual e física das tabelas

Escrita dos comandos SQL enviados pelas aplicações

Definição de índices, clustering de tabelas, etc

Dimensionamento da rede de comunicação entre clientes e o servidor

Metodologia para Tuning

Defina requisitos de desempenho

Isole medições sobre o SO, SGBD, rede e clientes

Realize benchmarks em um sistema sem carga, para obter um parâmetro de "melhor possível"

Realize benchmarks continuamente, para medir a melhora (ou piorar) a cada etapa de tuning

Pare quanto atingir os requisitos!

Tunning da Aplicação

Modelagem física

Otimização de comandos SQL

Modelagem Física

Utilize tabelas em CLUSTER caso sejam frequentemente acessadas segundo uma única ordenação

Evite a denormalização de dados; em geral, o custo de manutenção da redundância (fora o risco de perda da integridade dos dados) não compensa os pequenos ganhos de performance obtidos

Utilize bancos replicados para aplicações com padrões de acesso muito diferenciados, ex: aplicação OLAP de linha-de-negócios e aplicação OLTP, gerencial

Otimização Otimização de Comandos SQL

Utilize o comando EXPLAIN para visualizar o plano de acesso dos comandos SQL utilizados pela aplicação

Experimente criar índices para otimizar junções, filtros, agrupamento ou ordenação, e verifique os

resultados com o comando EXPLAIN

Execute o comando VACUUM ANALYSE regularmente, para atualizar as estatísticas de tabelas e índices

Verifique periodicamente mudanças nos planos de acesso – pode ser necessário mudar os índices!

Arquitetura do PostgreSQL

Processos
Memória
Disco

Processos Processos do PostgreSQL

Um único processo postmaster escuta por conexões TCP ou Unix Sockets, e inicia um processo postgres para cada conexão

Assim sendo, o PostgreSQL utiliza naturalmente múltiplos processadores, mas para usuários

simultâneos, não para acelerar um único comando SQL

Dois processos postgres estão sempre ativos, mas não atendem a conexões. Eles cuidam da gravação física de blocos de log ou tabelas, e da manutenção de estatísticas

Memória Memória e o PostgreSQL

Todos os processos postgres compartilham duas áreas de memória:

O buffer cache armazena blocos lidos (ou modificados) de tabelas e índices

O write-ahead log armazena temporariamente o log de transações, até que ele possa ser armazenado em disco

Além disso, cada processo postgres tem uma área individual para operações de ordenação (sort buffer)

Disco e o PostgreSQL

Bancos de dados são nada mais do que diretórios

Tabelas e índices são arquivos dentro destes diretórios

Um diretório em separado armazena um ou mais segmentos de log de transações

Não há no PostgreSQL estruturas similares a tablespaces, log groups, etc

Conta-se com a capacidade do SO em alocar espaço em disco de forma eficiente, e com espelhamento pelo HW

Tunning do SO

O que medir

Como medir

Tunning do SO

Um SGBD, como qualquer outra aplicação, utiliza processos, memória, disco e conexões de rede

Portanto, obtenha do seu SO medidas de cada uma dessas áreas!

Está havendo muito swap?

A fila de requisições ao disco está crescendo?

A fila de requisições à placa de rede está crescendo?
O processador está ocioso?

Tunning do SO

Muitos contadores do Monitor de Performance do Windows NT/2000 apresentam valores incorretos, especialmente em discos IDE

Sistemas Unix (Linux, FreeBSD, etc) são ricos em ferramentas de monitoração – aprenda a usa-las!

vmstat

netstat

ps

/proc

Espalhando Arquivos pelos Discos

Para obter máxima performance de E/S, utilize discos dedicados para:

Log de transações

Índices de tabelas

Tabelas muito/pouco consultadas

Tabelas muito/pouco atualizadas

Utilize links simbólicos para mover tabelas, índices e etc para os discos dedicados

Objetos x Arquivos

Dado o banco de dados, qual o seu diretório:

select datname, oid from pg_database;

teste | 16976

Dado a tabela, qual o seu arquivo:

select relname, relfilenode from pg_class;

contato_pkey | 16979

contato | 16977

Então os dados da tabela "contato" do banco "teste"

estão no arquivo base/16976/16977

Recomendações

25% da RAM para shared buffer cache

2-4% da RAM para sort buffers

Pode ser necessário recompilar o kernel (do Linux)

para sistemas com mais do que 2Gb de RAM

Algumas versões do Linux não suportam mais do que
2Gb de memória compartilhada (shared buffer cache)

Apresentação de: **Fernando Lozano** www.lozano.eti.br

PostgreSQL Tuning: O elefante mais rápido que um leopardo O Banco está Lento – Problemas Comuns

- 60% dos problemas são relacionados ao mau uso da linguagem SQL;
- 20% dos problemas são relacionados a má modelagem do banco de dados;
- 10% dos problemas são relacionados a má configuração do SGDB;
- 10% dos problemas são relacionados a má configuração do SO.

O Banco está Lento – Decisões erradas

Concentração de regras de negócio na aplicação para processos em lote;

- Integridade referencial na aplicação
- Mal dimensionamento de I/O (CPU, Plataforma, Disco)
- Ambientes virtualizados (Vmware, XEN, etc..) em AMD64/EMT64
- Uso de configurações padrões do SO e/ou do PostgreSQL

Melhor Hardware

Servidores dedicados para o PostgreSQL

- Storage com Fiber Channel e iSCSI: Grupos de RAID dedicados
- RAID 5 ou 10: por Hardware
- Mais memória! (Até 4GB em 32 bits)
- Processadores de 64 bits: Performance até 3 vezes do que os 32 bits (AMD64 e EMT64 - Intel)

Melhor SO

Sistemas Operacionais *nix: Linux (Debian, Gentoo), FreeBSD, Solaris, etc

- Em Linux: use Sistemas de arquivos XFS (noatime), Ext3 (writeback, noatime), Ext2
- Instale a última versão do PostgreSQL (atualmente 8.2) e à partir do código-fonte
- Não usar serviços concorrentes (Apache, MySQL, SAMBA...) em discos, semáforos e shared memory
- Usar, se possível, um kernel (linux) mais recente (e estável)

Parâmetros do SO – Modificando o *nix

```
echo "2" > /proc/sys/vm/overcommit_memory
echo "25%" > /proc/sys/kernel/shmmax
echo "25%/64" > /proc/sys/kernel/shmall
echo "deadline" > /sys/block/sda/queue/scheduler
echo "250 32000 100 128" > /proc/sys/kernel/sem
echo "65536" > /proc/sys/fs/file-max
ethtool -s eth0 speed 1000 duplex full autoneg off
echo "16777216" > /proc/sys/net/core/rmem_default
echo "16777216" > /proc/sys/net/core/wmem_default
echo "16777216" > /proc/sys/net/core/wmem_max
echo "16777216" > /proc/sys/net/core/rmem_max
pmanson:~# su - postgres
postgres@pmanson:~$ ulimit 65535
```

```
/etc/security/limits.conf
postgres soft nofile 4096
postgres hard nofile 63536
```

Como Organizar os Discos – O Melhor I/O

Discos ou partições distintas para:

- Logs de transações (WAL)
 - Índices: Ext2
 - Tabelas (particionar tabelas grandes)
 - Tablespace temporário (em ambiente BI)*
 - Archives
 - SO + PostgreSQL
 - Log de Sistema
- * Novo no PostgreSQL 8.3.

postgresql.conf – Memória

- max_connections: O menor número possível
- shared_buffers: 33% do total -> Para operações em execução
- temp_buffers: Acesso às tabelas temporárias
- work_mem: Para agregação, ordenação, consultas complexas
- maintenance_work_mem: 75% da maior tabela ou índice
- max_fsm_pages: Máximo de páginas necessárias p/ mapear espaço livre. Importante para operações de UPDATE/DELETE.

postgresql.conf – Disco e WAL

- wal_sync_method: open_sync, fdatasync, open_datasync
- wal_buffers: tamanho do cache para gravação do WAL
- commit_delay: Permite efetivar várias transações na mesma chamada de fsync
- checkpoint_segments: tamanho do cache em disco para operações de escrita
- checkpoint_timeout: intervalo entre os checkpoints
- wal_buffers: 8192kB -> 16GB
- bgwriter: ?????
- joinCollapse_limit = > 8

Tuning de SQL

- Analyze:
test_base=# EXPLAIN ANALYZE SELECT foo FROM bar;
- Ferramentas:
 - PgFouine;
 - PgAdmin3;
 - PhpPgAdmin;

Manutenção

- Autovacuum X Vacuum: Depende do uso (Aplicações Web, OLTI, BI)
 - Vacuum:
 - vacuum_cost_delay: tempo de atraso para vacuum executar automaticamente nas tabelas grandes
 - Autovacuum (ativado por padrão a partir da versão 8.3):
 - autovacuum_naptime: tempo de espera para execução do autovacuum.

Ferramentas de Stress

- Pgbench: no diretório do contrib do PostgreSQL, padrão de transações do tipo TPC-B.
- DBT-2: Ferramenta da OSDL, padrão de transações do tipo TPC-C.
- BenchmarkSQL: Ferramenta Java para benchmark em SQL para vários banco de dados (JDBC), padrão de transações do tipo TPC-C.
- Jmeter: Ferramenta Java genérica para testes de stress, usado para aplicações (Web, ...) e também pode ser direto para um banco de dados.

Quando o tuning não resolve

- Escalabilidade vertical:
 - Mais e melhores discos;
 - Mais memória;
 - Melhor processador (quad core, 64bits)
 - Escalabilidade horizontal:
 - Pgpool I (distribuição de carga de leitura e pool de conexões)
 - PgPool II (PgPool I + paralelização de grandes consultas)
 - Slony I (Replicação Multi-Master Assíncrona)
 - Warm Stand By
 - PgBouncer + PL/Proxy + Slony
- Apresentação de: Fernando Ike e Fábio Telles

Checklist de performance do PostgreSQL 8.0

Autor: Fábio Telles Rodriguez

Tradução livre do texto "PostgreSQL 8.0 Performance Checklist", publicado por Josh Berkus em 12/01/2005 em:

- <http://www.powerpostgresql.com/PerfList>

Copyright (c) 2005 by Josh Berkus and Joe Conway. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v1.0 or later (the latest version is presently available at <http://www.opencontent.org/openpub/>).

Aqui está um conjunto de regras para configurar seu servidor PostgreSQL 8.0. Muito do que está abaixo é baseado em evidências ou testes de escalabilidade práticos; há muito sobre performance de bancos de dados que nós e a OSDL, ainda estamos trabalhando. Contudo, isto deve ser um inicio. Todas as informações abaixo são úteis a partir de 12 de janeiro de 2005 e serão atualizadas depois. Discussões sobre configurações abaixo superam as que eu realizei no General Bits.

Cinco Princípios de Hardware para Configurar o seu Servidor PostgreSQL

1. Discos > RAM > CPU

Se você vai gastar dinheiro em um servidor PostgreSQL, gaste em arranjos de discos de alta performance e tenha processadores medianos e uma memória adequada. Se você tiver um pouco mais de dinheiro, adquira mais RAM. PostgreSQL, como outros SGDBs que suportam ACID, utilizam E/S muito intensamente e é raro uma aplicação utilizar mais a CPU do que a placa SCSI (com algumas exceções, claro). Isto se aplica tanto a pequenos como grandes servidores; obtenha uma CPU com custo baixo se isso permitir você comprar uma placa RAID de alta performance ou vários discos.

2. Mais unidades de discos == Melhor

Tendo múltiplos discos, o PostgreSQL e a maioria dos sistemas operacionais irão paralelizar as requisições de leitura e gravação no banco de dados. Isto faz uma enorme diferença em sistemas transacionais, e uma significativa melhoria em aplicações onde o banco de dados inteiro não cabe na RAM. Com os tamanhos mínimos de discos (72GB) você será tentado a utilizar apenas um disco ou um único par espelhado em RAID 1; contudo, você verá que utilizando 4, 6 ou até 14 discos irá render um impulso na performance. Ah, e SCSI é ainda significativamente melhor em fluxo de dados em BD que um IDE ou mesmo um Serial ATA.

3. Separe o Log de Transações do Banco de Dados

Assumindo que você já investiu dinheiro num arranjo com tamanho decente num conjunto de discos, existe um monte de opções mais inteligentes do que jogar tudo num único RAID. De inicio coloque o log de transações (pg_xlog) no seu próprio recurso de discos (um arranjo ou um disco), o que causa uma diferença de cerca de 12% na performance de bancos de dados com grande volume de gravações. Isto é especialmente vital em pequenos sistemas com discos SCSI ou IDE lentos: mesmo em um servidor com dois discos, você pode colocar o log de transações sobre o disco do sistema operacional e tirar algum benefício.

4. RAID 1+0/0+1 > RAID 5

RAID 5 com 3 discos tem sido um desafortunado padrão entre vendedores de servidores econômicos. Isto possibilita a mais lenta configuração de discos possível para o PostgreSQL; você pode esperar pelo menos 50% a menos de velocidade nas consultas em relação ao obtido com discos SCSI normais. Por outro lado, foque em RAID 1 ou 1+0 para um conjunto de 2, 4 ou 6 discos. Acima de 6 discos, o RAID 5 começa a ter uma performance aceitável novamente, e a comparação tende a ser bem melhor com base na sua controladora individual. No entanto, o mais importante, usar uma placa RAID barata pode ser um risco; é sempre melhor usar RAID por software do que um incorporado numa placa Adaptec que vem com seu servidor.

5. Aplicações devem rodar bem junto

Outro grande erro que eu vejo em muitas organizações é colocar o PostgreSQL em um servidor com várias outras aplicações competindo pelos mesmos recursos. O pior caso é colocar o PostgreSQL junto com outros SGDBs na mesma máquina; ambos bancos de dados irão lutar pela banda de acesso aos discos e o cache de disco do SO, e ambos vão ter uma performance pobre. Servidores de arquivo e programas de log de segurança também são ruins. O PostgreSQL pode compartilhar a mesma máquina com aplicações que utilizam principalmente CPU e RAM intensamente, como o Apache, garantindo que exista RAM suficiente.

Doze Ajustes que Você Irá Querer Fazer no Seu Arquivo PostgreSQL.conf

Existem um monte de novas opções verdadeiramente assustadoras no arquivo PostgreSQL.conf. Mesmo as já familiares opções das 5 últimas versões mudaram de nomes e formato dos parâmetros. Elas tem a intenção de dar ao administrador de banco de dados mais controle, mas podem levar algum tempo para serem usados.

O que segue são configurações que a maioria dos DBAs vão querer alterar, focado no aumento de performance acima de qualquer outra coisa. Existem algumas poucas configurações que particularmente a maioria dos usuários não querem mexer, mas quem o fizer irá descobri-las indispensáveis. Para estes, vocês terão de aguardar pelo livro.

Lembre-se: as configurações no PostgreSQL.conf precisam ser descomentadas para fazerem efeito, mas recomentá-las não restaurará necessariamente o valor padrão!

Coneção

listen_addresses:

Substitui as configurações tcp_ip e o virtual_hosts do 7.4. O padrão é localhost na maioria das instalações, habilitando apenas conexões pelo console. A maioria dos DBAs irá querer mudar isto para "*", significando que todas as interfaces avaliables, após configurar as permissões em hba.conf apropriadamente, irão tornar o PostgreSQL acessível pela rede. Como uma melhoria sobre a versão anterior, o "localhost" permite conexões pela interface de "loopback", 127.0.0.1, habilitando vários utilitários baseados em servidores web.

max_connections:

Exatamente como na versão anterior, isto precisa ser configurado para o atual número de conexões simultâneas que você espera precisar. Configurações altas vão requerer mais memória compartilhada (shared_buffers). Como o overhead por conexão, tanto do PostgreSQL como do SO do host podem ser bem altos, é importante utilizar um pool de conexões se você precisar servir um número grande de usuários. Por exemplo, 150 conexões ativas em um servidor Linux com um processador médio de 32 bits consumirá recursos significativos, e o limite deste hardware é de 600. Claro que um hardware mais robusto irá permitir mais conexões.

Memória

shared_buffers:

Como um lembrete: Este não é a memória total do com o qual o PostgreSQL irá trabalhar. Este é o bloco de memória dedicado ao PostgreSQL utilizado para as operações ativas, e deve ser a menor parte da RAM total na máquina, uma vez que o PostgreSQL usa o cache de disco também. Infelizmente, o montante exato de shared buffers requer um complexo cálculo do total de RAM, tamanho do banco de dados, número de conexões e complexidade das consultas. Assim, é melhor seguir algumas regras na alocação, e monitorar o servidor (particularmente as visões pg_statio) para determinar ajustes.

Em servidores dedicados, valores úteis costumam ficar entre 8MB e 400MB (entre 1000 e 50.000 para páginas de 8K). Fatores que aumentam a quantidade de shared buffers são

grandes porções ativas do banco de dados, consultas grandes e complexas, grande número de conexões simultâneas, longos procedimentos e transações, maior quantidade de RAM disponível, CPUs mais rápidas ou em maior quantidade obviamente, outras aplicações na mesma máquina. Contrário a muitas expectativas, alocando, muita, demasiadamente shared_buffers pode até diminuir a performance, aumentando o tempo requerido para explorá-la. Aqui estão alguns exemplos baseados em experiências e testes TPC em máquinas Linux:

- Laptop, processador Celeron, 384MB RAM, banco de dados de 25MB: 12MB/1500;
- Servidor Athlon, 1GB RAM, banco de dados de 10GB para suporte a decisão: 120MB/15000;
- Servidor Quad PIII, 4GB RAM, banco de dados de 40GB, com 150 conexões e processamento pesado de transações: 240MB/30000;
- Servidor Quad Xeon, 8GB RAM, banco de dados de 200GB, com 300 conexões e processamento pesado de transações: 400MB/50000.

Favor notar que incrementando shared_buffer, e alguns outros parâmetros de memória, vão requerer que você modifique o System V do seu sistema operacional. Veja a documentação principal do PostgreSQL para instruções nisto.

work_mem:

Costuma ser chamado de sort_mem, mas foi renomeado uma vez que ele agora cobre ordenações, agregações e mais algumas operações. Esta memória não é compartilhada, sendo alocada para cada operação (uma a várias vezes por consulta); esta configuração está aqui para colocar um teto na quantidade de memória que uma única operação ocupar antes de ser forçada para o disco. Este deve ser calculado dividindo a RAM disponível (depois das aplicações e do shared_buffers) pela expectativa de máximo de consultas concorrentes vezes o número de memória utilizada por conexão.

Considerações devem ser tomadas sobre o montante de work_mem por consulta; processando grandes conjuntos de dados requisitará mais. Bancos de dados de aplicações Web geralmente utilizam números baixos, com numerosas conexões mas consultas simples, 512K a 2048K geralmente é suficiente. Contrariamente, aplicações de apoio a decisão com suas consultas de 160 linhas e agregados de 10 milhões de linhas precisam de muito, chegando a 500MB em um servidor com muita memória. Para bancos de dados de uso misto, este parâmetro pode ser ajustado por conexão, em tempo de execução, nesta ordem, para dar mais RAM para consultas específicas.

maintenance_work_mem:

Formalmente chamada de vacuum_mem, esta quantidade de RAM é utilizada pelo PostgreSQL para o VACUUM, ANALYZE, CREATE INDEX, e adição de chaves estrangeiras. Você deve aumentar quanto maior forem suas tabelas do banco de dados e quanto mais memória RAM você tiver de reserva, para fazer estas operações o mais rápidas possível. Uma configuração com 50% a 75% da sua maior tabela ou índice em disco é uma boa regra, ou 32MB a 256MB onde isto não pode ser determinado.

Disco e WAL

`checkpoint_segments`:

Define o tamanho do cache de disco do log de transações para operações de escrita. Você pode ignorar isto na maioria dos bancos de dados web com a maioria das operações em leitura, mas para bancos de dados de processamento de transações ou para bancos de dados envolvendo grandes cargas de dados, o aumento dele é crítico para a performance. Dependendo do volume de dados, aumente ele para algo entre 12 e 256 segmentos, começando conservadoramente e aumentando se você ver mensagens de aviso no log. O espaço requerido no disco é igual a $(\text{checkpoint_segments} * 2 + 1) * 16\text{MB}$, então tenha certeza de ter espaço em disco suficiente (32 significa mais de 1GB).

`max_fsm_pages`:

Dimensiona o registro que rastreia as páginas de dados parcialmente vazias para popular com novos dados; se configurado corretamente, torna o VACUUM mais rápido e remove a necessidade do VACUUM FULL ou REINDEX. Deve ser um pouco maior que o total de número páginas de dados que serão tocados por atualizações e remoções entre vacuums. Os dois modos de determinar este número são rodar o VACUUM VERBOSE ANALYZE, ou se estiver utilizando autovacuum (veja abaixo) configures este de acordo com o parâmetro `-V` como uma porcentagem do total de páginas de dados utilizado por seu banco de dados. `fsm_pages` requer muito pouco memória, então é melhor ser generoso aqui.

`vacuum_cost_delay`:

Se você tiver tabelas grandes e um significativo montante de atividades de gravações concorrentes, você deve querer fazer uso deste novo recurso que diminui a carga de I/O do VACUUM sobre o custo de fazê-las mais longas. Como este é um novo recurso, é um complexo de 5 configurações dependentes para o qual nós temos apenas poucos testes de performance. Aumentando o `vacuum_cost_delay` para um valor não zero ativa este recurso; use um atraso razoável, algo entre 50 e 200ms. Para um ajuste fino, aumente o `vacuum_cost_page_hit` e diminua o `vacuum_cost_page_limit` irá diminuir o impacto dos vacuums e tornará eles mais longos; em testes de Jan Wieck's num teste de processamento de transações, um delay de 200, page_hit de 6 e limit de 100 diminuiu o impacto do vacuum em mais de 80% enquanto triplicou o tempo de execução dele.

Planejador de Consultas

Estas configurações permitem o planejador de consultas fazer estimativas mais precisas dos custos de operação e assim escolher o melhor plano de execução. Os dois valores de configurações para se preocupar são:

`effective_cache_size`:

Diz ao planejador de consultas o mais largo objeto do banco de dados que pode se esperar ser cacheado. Geralmente ele deve ser configurado em cerca de 2/3 da RAM, se estiver num servidor dedicado. Num servidor de uso misto, você deve estimar quanto de RAM e cache de disco outras aplicações estarão utilizando e subtrair eles.

`random_page_cost`:

Uma variável que estima o custo médio em buscas por páginas de dados indexados. Em máquinas mais rápidas, com arranjos de discos velozes ele deve ser reduzido para 3.0, 2.5 ou até mesmo 2.0. Contudo, se a porção ativa do seu banco de dados é muitas vezes maior que a sua RAM, você vai querer aumentar o fator de volta para o valor padrão de 4.0. Alternativamente, você pode basear seus ajustes na performance. Se o planejador injustamente a favor de buscas seqüenciais sobre buscas em índices, diminua-o. Se ele estiver utilizando índices lentos quando não deveria, aumente-o. Tenha certeza de testar uma variedade de consultas. Não abaixe ele para menos de 2.0; se isto parecer necessário, você precisa de ajustes em outras áreas, como as estatísticas do planejador.

Logging

`log_destination`:

Isto substitui o intuitivo a configuração syslog em versões anteriores. Suas escolhas são usar o log administrativo do SO (syslog ou eventlog) ou usar um log separado para o PostgreSQL (stderr). O primeiro é melhor para monitorar o sistema; o último é melhor para encontrar problemas no banco de dados e para o tuning.

`redirect_stderr`:

Se você decidir usar um log separado para o PostgreSQL, esta configuração permitirá registrar num arquivo utilizando uma ferramenta nativa do PostgreSQL ao invés do redirecionamento em linha de comando, permitindo a rotação do log. Ajuste para True, e então ajuste o `log_directory` para dizer onde colocar os logs. A configuração padrão para o `log_filename`, `log_reoation_size` e `log_rotation` são bons para a maioria das pessoas.

Autovacuum e você

Assim que você entra em produção no 8.0, você vai querer fazer um plano de manutenção incluindo VACUUMs e ANALYZEs. Se seus bancos de dados envolvem um fluxo contínuo de escrita de dados, mas não requerem maciças cargas e apagamentos de dados ou freqüentes reinícios, isto significa que você deve configurar o `pg_autovacuum`. Isto é melhor que agendar `vaccuns` porque:

- Tabelas sofrem o vacuum baseados nas suas atividades, excluindo tabelas que apenas sofrem leituras.
- A freqüência dos vacuums cresce automaticamente com o crescimento da atividade no banco de dados.
- É mais fácil calcular o mapa de espaço livre e evitar o inchaço do banco de dados.

Carga de dados no banco: <http://pgdocptbr.sourceforge.net/pg80/populate.html>

Dicas de Performance em aplicações com PostgreSQL

Tradução do texto original de Josh Berkus

O que se segue é a versão editada de um conjunto de conselhos que eu tenho dado ao time da Sun no redesenho de uma aplicação em C++ que foi construída para MySQL, portado para o PostgreSQL, e nunca otimizado para performance. Ocorreu que estes conselhos podem ser geralmente úteis para a comunidade, então aí vão eles.

Projeto de aplicações para performance no PostgreSQL

Escrevendo regras de consultas

Para todos os sistemas gerenciadores de bancos de dados (SGDBs), o tempo por rodada é significativo. Este é o tempo que leva para uma consulta passar pelo analisador de sintaxe da linguagem, o driver, a interface da rede, o analisador de sintaxe do banco de dados, o planejador, o executor, o analisador de sintaxe novamente, voltar pela interface de rede, passar pelo manipulador de dados do driver e para o cliente da aplicação. SGDBs variam na quantidade de tempo e CPU que elas levam para processar este ciclo, e por uma variedade de razões, o PostgreSQL possui um alto consumo de tempo e recursos do sistema por rodada.

Contudo, o PostgreSQL tem um overhead significativo por transação, incluindo o log de saída e as regras de acesso que precisam ser ajustadas em cada transação. Enquanto você pode pensar que não está utilizando transações para um simples comando de leitura SELECT, de fato, cada simples comando no PostgreSQL é uma transação. Na ausência de uma transação explícita, o comando é por si mesmo implicitamente uma transação.

Passando por isto, o PostgreSQL é claramente o segundo depois do Oracle em processamento de consultas complexas e longas com transações com vários comandos com fácil resolução de conflitos de concorrência. Ele também suporta cursores, tanto rolável quanto não rolável.

Dica 1: Nunca use várias consultas pequenas quando uma grande consulta pode fazer o trabalho.

É comum em aplicações MySQL lidar com joins no código da aplicação, ou seja, consultando o ID de um registro relacionado e então iterando através dos registros filhos com aquele ID manualmente. Isto pode resultar em rodar centenas de consultas por tela de interface com o usuário. Cada uma destas consultas levam 2 a 6 milissegundos por rodada, o que significa que se você executar cerca de 1000 consultas, neste ponto você estará perdendo de 3 a 5 segundos. Comparativamente, solicitando estes registros numa única consulta levará apenas algumas centenas de milissegundos, economizando cerca de 80% do tempo.

Dica 2: Agrupe vários pequenos UPDATEs, INSERTs ou DELETEs em um único comando ou, se não for possível, em uma longa transação.

Antes, a falta de subselects nas versões anteriores do MySQL fizeram com que os desenvolvedores de aplicação projetassem seus comandos de modificação de dados (DML) da mesma forma que as junções em middleware. Esta é uma má idéia para o PostgreSQL. Ao invés, você irá tirar vantagem de subselects e joins no seu comando UPDATE, INSERT, e DELETE para tentar realizar modificações em lote com um único comando. Isto reduz o tempo da rodada e o overhead da transação.

Em alguns casos, contudo, não há uma única consulta que consiga alterar todas as linhas que você deseja e você irá usar um grupo de comandos em série. Neste caso, você irá querer se assegurar de envolver a sua série de comandos DML em uma transação explícita (ex. BEGIN; UPDATE; UPDATE; UPDATE; COMMIT;). Isto reduz o overhead de transação e corta o tempo de execução em até 50%.

Dica 3: Considere realizar cargas em lotes ao invés de INSERTs seriais.

O PostgreSQL prove um mecanismo de carga em lote chamado COPY, que pode pegar uma entrada de um arquivo ou pipe delimitado por tabulações ou CSV. Quando o COPY pode ser usado no lugar de centenas de INSERTs, ele pode cortar o tempo de execução em até 75%.

Dica 4: O DELETE é caro

É comum para um desenvolvedor de aplicação pensar que o comando DELETE é praticamente não tem custo. Você está apenas desligando alguns nós, correto? Errado. SGDBs não são sistemas de arquivo; quando você apaga uma linha, índices precisam ser atualizados, o espaço liberado precisa ser limpo, fazendo a exclusão de fato mais cara que a inserção. Assim, aplicações que habitualmente apagam todas as linhas de detalhe e repõe elas com novas toda vez que é realizada qualquer alteração estão economizando esforço no lado da aplicação e empurrando este dentro do banco de dados. Quando possível, isto deve ser substituído pela substituição mais discriminada das linhas, como atualizar apenas as linhas modificadas.

Além disso, quando for limpar toda uma tabela, sempre use o comando TRUNCATE TABLE ao invés de DELETE FROM TABLE. A primeira forma é até 100 vezes mais rápida que a posterior devido ao processamento da tabela como um todo ao invés de uma linha por vez.

Dica 5: Utilize o PREPARE/EXECUTE para iterações em consultas

Algumas vezes, mesmo tentando consolidar iterações de consultas semelhantes em um comando mais longo, nem sempre isto é possível de estruturar na sua aplicação. É para isto que o PREPARE ... EXECUTE serve; ele permite que o motor do banco de dados pule o analizador de sintaxe e o planejador para cada iteração da consulta. Por exemplo:

Preparar:

```
query_handle = query('SELECT * FROM TABLE WHERE id = ?')
(parameter_type = INTEGER)
```

Então inicie as suas iterações:

```
for 1..100
query_handle.execute(i);
end
```

Classes para a preparação de comandos no C++ são explicadas na documentação do libpqxx.

Isto irá reduzir o tempo de execução na direta proporção do tamanho do número de iterações.

Dica 6: Use pool de conexões efetivamente

Para uma aplicação web, você irá perceber que até 50% do seu potencial de performance pode ser controlado através do uso, e configuração apropriada de um pool de conexões. Isto é porque criar e destruir conexões no banco de dados leva um tempo significativo no sistema, e um excesso de conexões inativas continuarão a requerer RAM e recursos do sistema.

Há um número de ferramentas que você pode utilizar para fazer um pool de conexões no PostgreSQL. Uma ferramenta de terceiros de código aberto é o pgPool. Contudo, para uma aplicação em C++ com requisitos de alta disponibilidade, é provavelmente melhor utilizar a técnica de pseudo-pooling nativa do libpqxx chamada de "conexões preguiçosas". Eu sugiro contatar a lista de e-mail para mais informações sobre como utilizar isto.

Com o PostgreSQL, você irá querer ter quantas conexões persistentes (ou objetos de conexão) forem definidas no seu pico normal de uso de conexões concorrentes. Então, se o uso máximo normal (no início da manhã, digamos) é de 200 conexões concorrentes de agentes, usuários e componentes, então você irá querer que ter esta quantidade definida para que sua aplicação não tenha que esperar por novas conexões durante o pico onde será lenta na criação.

Fábio Telles em:

<http://www.midstorm.org/~telles/2007/01/05/dicas-de-performance-em-aplicacoes-com-postgresql/>

Veja Melhorando a Performance do seu HD:

Piter Punk em: <http://piterpunk.info02.com.br/artigos/hparm.html>

Gerenciando Recursos do Kernel:

<http://www.postgresql.org/docs/8.3/interactive/kernel-resources.html>

Escolhendo o sistema Operacional para o servidor do SGBD PostgreSQL

Sem discussão, o Linux/Unix ganha 25% a 50% de performance em cima do Windows. Isso se dá ao melhor gerenciamento de recursos.

<http://vivaolinux.com.br/dicas/verDica.php?codigo=9847>

Melhorando a performance do PostgreSQL com o comando VACUUM em:

http://imasters.uol.com.br/artigo/2421/postgresql/melhorando_a_performance_do_postgresql_com_o_comando_vacuum/

Otimizando bancos PostgreSQL - Parte 01

Olá! Neste meu primeiro artigo gostaria de descrever alguns ajustes finos do PostgreSQL que ajudam a melhorar a performance geral do banco de dados. Estas configurações são válidas para a versão 8.1. Caso você possua uma versão anterior, recomendo fortemente a atualização para a **8.1**, pois esta última é **muito** mais rápida que as anteriores, mesmo utilizando as configurações padrão.

Infelizmente, a maior parte da degradação de performance de um banco de dados está na estrutura e/ou nos comandos SELECT mal elaborados. Neste artigo, será feita uma abordagem única e específica nos ajustes de configuração do SGBD, fazendo-o usufruir do máximo dos recursos de hardware onde está instalado o serviço do PostgreSQL.

É importante salientar também que não existe fórmula mágica para as configurações, sendo que as opções de um servidor pode não ser a melhor opção para outro.

Edição do arquivo postgresql.conf

Para começar, localize o arquivo chamado **postgresql.conf**. Este arquivo encontra-se no diretório de dados do cluster (ou agrupamento de bancos de dados) o qual você está inicializando. Em instalações normais, você pode encontrá-lo em:

- Microsoft Windows:

- Pasta "Arquivos de Configuração" no Menu Iniciar/Programas
- ou
- C:\Arquivos de Programas\PostgreSQL\8.1\data\

- Linux (Red Hat e Fedora)

/var/lib/pgsql/data

Após encontrá-lo, certifique-se de criar uma cópia de backup do arquivo antes de alterá-lo, pois erros na configuração podem fazer com que o PostgreSQL não seja inicializado.

Feito o backup, abra o arquivo em modo de edição. Se estiver no Linux, certifique-se de estar logado com o usuário **root** ou **postgres**.

Os parâmetros devem ser preenchidos seguindo o padrão **nome_do_parametro = valor**. Lembrando que para valores do tipo data e texto deve-se envolver o valor com aspas simples, e para os valores numéricos não inserir separador de milhar. Não esqueça de remover o caracter cerquilha "#" do início das opções que deseja ativar. Todas as alterações serão efetivadas após a reinicialização do serviço PostgreSQL.

shared_buffers

Define o espaço de memória alocado para o PostgreSQL armazenar as consultas SQL antes de devolvê-las ao buffer do sistema operacional. Esta opção pode solicitar que parâmetros do Kernel sejam modificados para liberar mais memória compartilhada do sistema operacional, pois esta passa a ser utilizada também pelo Postgre, em maior quantidade. O valor desta configuração está expresso em blocos de 8 Kbytes (128 representa 1.024 Kbytes ou 1 Mb).

Uma boa pedida é utilizar valores de 8% a 12% do total de RAM do servidor para esta configuração. Caso, após mudar o valor deste parâmetro, o PostgreSQL não inicializar o cluster em questão, altere o sistema operacional para liberar mais memória compartilhada. Consulte o manual ou equivalentes do seu sistema operacional para obter instruções de como aumentar a memória compartilhada (Shared Memory) disponível para os programas.

Exemplo:

`shared_buffers = 2048 # Seta a memória compartilhada para 16 Mbytes`

work_mem

Configura o espaço reservado de memória para operações de ordem e manipulação/consulta de índices. Este parâmetro configura o tamanho em KBytes utilizado no servidor para cada conexão efetivada ao SGBD, portanto esteja ciente que o espaço total da RAM utilizado (valor da opção multiplicado pelo número de conexões simultâneas) não deve ultrapassar 20% do total disponível (valor aproximado).

Exemplo:

`work_mem = 2048 # Configura 2 Mbytes de RAM do servidor para operações de ORDER BY, CREATE INDEX e JOIN disponíveis para cada conexão ao banco.`

maintenance_work_mem

Expressa em KBytes o valor de memória reservado para operações de manutenção (como VACUUM e COPY). Se o seu processo de VACUUM está muito custoso, tente aumentar o valor deste parâmetro.

Nota: O total de memória configurada neste parâmetro é utilizado somente durante as operações de manutenção do banco de dados, sendo liberada durante o seu uso normal.

Exemplo:

`maintenance_work_mem = 16384 # 16 Mbytes reservados para operações de manutenção.`

max_fsm_pages

Em bancos de dados grandes, é ideal que a cada execução do VACUUM mais páginas "sujas" sejam removidas do banco de dados do que a quantidade padrão, principalmente por questões de espaço em disco, e claro, performance. Porém, a configuração padrão traz apenas 20000 páginas. Em bancos transacionais, com no mínimo 10 usuários, o número mensal de páginas sujas pode exceder este valor.

Para realizar uma limpeza maior, aumente o valor desta configuração. Nota: incrementando o valor deste parâmetro pode resultar em aumento do tempo para execução do VACUUM, e cada página ocupa em média 6 bytes de RAM constantemente. O comando VACUUM pode ser comparado ao comando PACK do DBASE (DBFs), porém possui mais funcionalidades.

Exemplo:

`max_fsm_pages = 120000 # Realiza a procura por até 120.000 páginas sujas na limpeza pelo VACUUM utilizando cerca de 71 Kb de RAM para isto.`

wal_buffers

Número de buffers utilizados pelo WAL (Write Ahead Log). O WAL garante que os registros sejam gravados em LOG para possível recuperação antes de fechar uma transação. Porém, para bancos transacionais com muitas operações de escrita, o valor padrão pode diminuir a performance. Entretanto, é importante ressaltar que aumentar muito o valor deste parâmetro pode resultar em perda de dados durante uma possível queda de energia, pois os dados mantém-se. Cada unidade representa 8 Kbytes de uso na RAM. Valores ideais estão entre 32 e 64. Se o disco conjunto do servidor (HW & SW) são quase infalíveis, tente usar 128 ou 256.

Exemplo:

`wal_buffers = 64 # Seta para 512 Kbytes a memória destinada ao buffer de escrita no WAL.`

effective_cache_size

Esta configuração dita o quanto de memória RAM será utilizada para cache efetivo (ou cache de dados) do banco de dados. Na prática, é esta a configuração que dá mais fôlego ao SGBD, evitando a constante leitura das tabelas e índices ,dos arquivos do disco rígido.

Como exemplo, se uma tabela do banco de dados possui 20 Mbytes, e o tamanho para esta configuração limita-se aos quase 8 Mbytes padrão, o otimizador de querys irá carregar a tabela por etapas, em partes, até que ela toda seja vasculhada em busca dos registros. Esta opção impacta diretamente aumentando a performance do banco de

dados, principalmente quando há concorrência, pois diminui consideravelmente as operações de I/O de disco.

Em consultas pela internet, encontrei várias referências para utilizar no máximo 25% da RAM total. Porém, se o servidor for dedicado, ou dispôr de uma grande quantidade de memória (512 Mbytes ou mais), recomendo o uso de 50% da RAM para esta configuração. Cada unidade corresponde a 8 Kbytes de RAM.

Exemplo:

```
effective_cache_size = 32768 # Seta o cache de dados do PostgreSQL para 256 Mbytes de RAM.
```

random_page_cost

Define o custo (em tempo) para a seleção do plano de acesso aos dados do banco. Se você possui discos rígidos velozes (SCSI, por exemplo), tente utilizar valores como 1 ou 2 para esta configuração. Para os demais casos, limite-se a 3 ou 4. Isto impacta no plano estabelecido pelo otimizador interno, que pode realizar um Index_Scan ou Table_Scan, etc, conforme aquilo que ele determinar ser mais otimizado.

Exemplo:

```
random_page_cost = 2 # Diminui o tempo para seleção aleatória de páginas do otimizador de consultas.
```

Notas gerais

Caso os valores inseridos possuam algum erro, é possível que o PostgreSQL não consiga inicializar o agrupamento de banco de dados no qual o arquivo postgresql.conf foi modificado. Se isto ocorrer, retorne o backup do mesmo e certifique-se de que as modificações estão corretamente setadas.

É possível obter ganhos de performance significativos com a correta seleção dos valores para as configurações acima. Tente várias combinações dependendo da capacidade e do uso do servidor (hardware) onde está instalado o PostgreSQL 8.1.

Otimizando bancos PostgreSQL - Parte 02

Criação de Índices Parciais (Partial Indexes)

Em tabelas com muitos registros, a utilização de índices normais pode causar um desempenho insatisfatório, principalmente quando se trata de colunas que representam abstrações de dados com pouca variação. É o caso de colunas representando tempo (DATE, TIME e TIMESTAMP), ou colunas numéricas representando tipos pré-definidos (Ex.: Regiões, Sexo, Faixa Salarial, etc.).

Tabelas de movimentação analítica com estes tipos de colunas podem conter milhares, ou até milhões de registros. Entretanto, em consultas SQL específicas por um determinado valor, um índice normal completo (Full Index) irá considerar na consulta todos os registros da tabela, organizados na ordem do índice.

O PostgreSQL possui um fantástico recurso para criação de índices que permite delimitar os registros que este irá considerar. Isto representa um enorme ganho de desempenho, especialmente em consultas SQL que utilizam filtros complexos no WHERE.

Vamos exemplificar este caso. Considere uma tabela de movimentação analítica de estoque de uma empresa de comércio comum. Vamos usar um modelo simples, apenas para demonstrar o caso. Utilize o código SQL abaixo para criar a tabela:

```
-- Criação da Tabela
CREATE TABLE estoque(
ID_Empresa INT2 NOT NULL,
ID_Produto INT4 NOT NULL,
ID_Local_Estoque INT2 NOT NULL,
TIPO_Estrada BOOLEAN NOT NULL DEFAULT false,
QTD_Quantidade NUMERIC(12,6) DEFAULT 0.000000,
VAL_Unitario NUMERIC(15,3) DEFAULT 0.000,
DT_Movimento DATE NOT NULL
);

-- Definição da chave primária
ALTER TABLE estoque ADD PRIMARY KEY(ID_Empresa, ID_Produto, ID_Local_Estoque);

-- Criação de Índice sobre o campo Data
CREATE INDEX idx_DATA ON estoque (DT_Movimento, ID_Empresa);
```

Imagine esta tabela com mais de 2.000.000 registros. O departamento de gerência de estoque emite relatórios mensais sobre a movimentação de estoque dos produtos para conferência. Um exemplo de relatório é de Entrada e Saída Consolidada, que considera os valores de entrada e saída por período. Um SQL típico para demonstrar as informações do mês de Dezembro de 2006 utilizaria o filtro no WHERE mencionando o campo DT_Movimento da seguinte forma: (...) WHERE DT_Movimento BETWEEN 2006-12-01 AND 2006-12-31.

Considere o volume de dados caso a empresa possua um movimento de mais de 50.000 registros por mês, mantendo esta marca desde 01012000. Ao utilizar o WHERE acima, uma varredura completa no índice idx_DATA seria feita, considerando a massa completa de dados no índice.

Dependendo de condições de uso dos registros estes podem estar na memória cache, então o resultado seria rapidamente apresentado. Entretanto, caso uma pesquisa aleatória não armazenada em cache for executada, o custo de IO do gerenciador de banco de dados seria problemático.

A solução neste caso - e uma medida muito satisfatória - é a criação dos índices parciais sobre o campo data, combinando-os com um índice normal completo. É possível criar os índices parciais para datas muito além das atuais, para prever a população de registros na tabela no futuro, de modo a garantir o desempenho. Lembre-se de que se não existirem registros com uma data prevista no índice, este não terá tamanho, portanto não será prejudicial em nenhum aspecto (espaço ou IO).

Para aperfeiçoar o acesso a dados nestas condições, os índices parciais consideram a cláusula SQL WHERE:

```
-- Criação de índice sobre o campo Data - Janeiro de 2006
CREATE INDEX idx_DATA_0106 ON estoque (DT_Movimento, ID_Empresa) WHERE
(DT_Movimento BETWEEN 2006-01-01 AND 2006-01-31);

-- Criação de índice sobre o campo Data - Fevereiro de 2006
CREATE INDEX idx_DATA_0206 ON estoque (DT_Movimento, ID_Empresa) WHERE
(DT_Movimento BETWEEN 2006-02-01 AND 2006-02-28);

(...)

-- Criação de índice sobre o campo Data - Dezembro de 2006
CREATE INDEX idx_DATA_1206 ON estoque (DT_Movimento, ID_Empresa) WHERE
(DT_Movimento BETWEEN 2006-12-01 AND 2006-12-31);

(...)
```

Desta forma, a todo SQL onde for utilizado a condição DT_Movimento BETWEEN 2006-12-01 AND 2006-12-31 ou sua equivalente DT_Movimento 2006-12-01 AND DT_Movimento 2006-12-31, o índice idx_DATA_0106 será apresentando para o otimizador interno como o mais eficaz, e portanto será usado.

Uma aplicação muito boa para os índices parciais é a utilização deste em tabelas que fazem parte de VIEWS (Visões) complexas. Todo o WHERE fixo da VIEW pode ser considerado em um índice parcial, o que resulta na diminuição considerável do tempo de resposta.

Tiago Adami em:

http://imasters.uol.com.br/artigo/4406/postgresql/otimizando_bancos_postgresql_-_parte_01/
http://imasters.uol.com.br/artigo/5328/postgresql/otimizando_bancos_postgresql_-_parte_02/

Otimizando operações de Entrada e Saída (I/O) em:

http://imasters.uol.com.br/artigo/4020/bancodedados/otimizando_operacoes_de_entrada_e_saida_io

Otimizar consultas SQL

Diferentes formas de otimizar as consultas realizadas em SQL.

A linguagem SQL é não procedural, ou seja, nas sentenças se indica o que queremos conseguir e não como tem que fazer o intérprete para consegui-lo. Isto é pura teoria, pois na prática todos os gerenciadores de SQL têm que especificar seus próprios truques para otimizar o rendimento.

Portanto, muitas vezes não basta com especificar uma sentença SQL correta, e sim que além disso, há que indicar como tem que fazer se quisermos que o tempo de resposta

seja o mínimo. Nesta seção, veremos como melhorar o tempo de resposta de nosso intérprete ante umas determinadas situações:

Design de tabelas

- * Normalize as tabelas, pelo menos até a terceira forma normal, para garantir que não haja duplicidade de dados e aproveitar o máximo de armazenamento nas tabelas. Se tiver que desnormalizar alguma tabela pense na ocupação e no rendimento antes de proceder.
- * Os primeiros campos de cada tabela devem ser aqueles campos requeridos e dentro dos requeridos primeiro se definem os de longitude fixa e depois os de longitude variável.
- * Ajuste ao máximo o tamanho dos campos para não desperdiçar espaço.
- * É normal deixar um campo de texto para observações nas tabelas. Se este campo for utilizado com pouca freqüência ou se for definido com grande tamanho, por via das dúvidas, é melhor criar uma nova tabela que contenha a chave primária da primeira e o campo para observações.

Gerenciamento e escolha dos índices

Os índices são campos escolhidos arbitrariamente pelo construtor do banco de dados que permitem a busca a partir de tal campo a uma velocidade notavelmente superior. Entretanto, esta vantagem se vê contra-arrestada pelo fato de ocupar muito mais memória (o dobro mais ou menos) e de requerer para sua inserção e atualização um tempo de processo superior.

Evidentemente, não podemos indexar todos os campos de uma tabela extensa já que dobramos o tamanho do banco de dados. Igualmente, tampouco serve muito indexar todos os campos em uma tabela pequena já que as seleções podem se efetuar rapidamente de qualquer forma.

Um caso em que os índices podem ser muito úteis é quando realizamos petições simultâneas sobre várias tabelas. Neste caso, o processo de seleção pode se acelerar sensivelmente se indexarmos os campos que servem de nexo entre as duas tabelas.

Os índices podem ser contraproducentes se os introduzimos sobre campos triviais a partir dos quais não se realiza nenhum tipo de petição já que, além do problema de memória já mencionado, estamos lentificando outras tarefas do banco de dados como são a edição, inserção e eliminação. É por isso que vale a pena pensar duas vezes antes de indexar um campo que não serve de critério para buscas ou que é usado com muita freqüência por razões de manutenção.

Campos a Selecionar

- * Na medida do possível há que evitar que as sentenças SQL estejam embebidas dentro do código da aplicação. É muito mais eficaz usar vistas ou procedimentos armazenados por que o gerenciador os salva compilados. Se se trata de uma sentença embebida o gerenciador deve compila-la antes de executá-la.
- * Selecionar exclusivamente aqueles que se necessitem
- * Não utilizar nunca SELECT * porque o gerenciador deve ler primeiro a estrutura da tabela antes de executar a sentença

* Se utilizar várias tabelas na consulta, especifique sempre a que tabela pertence cada campo, isso economizará tempo ao gerenciador de localizar a que tabela pertence o campo. Ao invés de SELECT Nome, Fatura FROM Clientes, Faturamento WHERE IdCliente = IdClienteFaturado, use: SELECT Clientes.Nome, Faturamento.Fatura WHERE Clientes.IdCliente = Faturamento.IdClienteFaturado.

Campos de Filtro

* Procuraremos escolher na cláusula WHERE aqueles campos que fazem parte da chave do arquivo pelo qual interroguemos. Ademais se especificarão na mesma ordem na qual estiverem definidas na chave.

* Interrogar sempre por campos que sejam chave.

* Se desejarmos interrogar por campos pertencentes a índices compostos é melhor utilizar todos os campos de todos os índices. Suponhamos que temos um índice formado pelo campo NOME e o campo SOBRENOME e outro índice formado pelo campo IDADE. A sentença WHERE NOME='Jose' AND SOBRENOME Like '%' AND IDADE = 20 seria melhor que WHERE NOME = 'Jose' AND IDADE = 20 porque o gerenciador, neste segundo caso, não pode usar o primeiro índice e ambas sentenças são equivalentes porque a condição SOBRENOME Like '%' devolveria todos os registros.

Ordem das Tabelas

Quando se utilizam várias tabelas dentro da consulta há que ter cuidado com a ordem empregada na cláusula FROM. Se desejarmos saber quantos alunos se matricularam no ano 1996 e escrevermos:

FROM Alunos, Matriculas WHERE Aluno.IdAluno = Matriculas.IdAluno AND Matriculas.Ano = 1996

o gerenciador percorrerá todos os alunos para buscar suas matrículas e devolver as correspondentes. Se escrevermos

FROM Matriculas, Alunos WHERE Matriculas.Ano = 1996 AND Matriculas.IdAluno = Alunos.IdAlunos

o gerenciador filtra as matrículas e depois seleciona os alunos, desta forma tem que percorrer menos registros.

De Cláudio em:

<http://www.criarweb.com/artigos/otimizar-consultas-sql.html>

Dicas de Desempenho em:

<http://pgdocptbr.sourceforge.net/pg80/performance-tips.html>

Diferença de Desempenho entre alguns Sistemas Operacionais

Fiz alguns teste com mesma máquina e configuração com windows 2000 server, Debian e FreeBSD 6.2, perdi um tempão instalado os tres SO (testeи um de cada vez separado) na máquina todos com configuração minima para não dizer que o debian e o FreeBSD pode ser instalado sem interface ai consumiria menos memória instalei o debian e o FreeBSD com KDE para ficar semelhante ao windows.

Uma consulta pesada que demorava 30s no Free e uns 33s no debian demora 1,5min em windows a diferença é grande.

Leandro DUTRA na lista pebr-geral

Melhor performance em instruções SQL

Artigo de Mauro Pichiliani no iMasters:

http://imasters.uol.com.br/artigo/222/sql_server/melhor_performance_em_instrucoes_sql/

Oi pessoal, nesta primeira coluna vamos falar um pouco sobre melhora de performance em instruções SQL. Para começar podemos entender instruções SQL por comandos do tipo SELECT, UPDATE e DELETE.

Cada comando possui uma série de detalhes, e no SQL Server, estes detalhes podem impactar muito na performance. No artigo desta semana, vou falar um pouco sobre talvez o mais utilizado deles: o SELECT.

Primeiro vamos à sintaxe básica do SELECT

`SELECT <lista_de_campos> [FROM <lista_de_tabelas> [WHERE <lista_de_condições>]] [GROUP BY <lista_de_campos>]`

No lugar de <lista_de_campos> devemos colocar uma expressão que seja passível de entendimento ao SQL Server. Por exemplo:

`SELECT GETDATE()` – Retorna a data atual do sistema

E na lista de tabelas os campos da tabela. Exemplo:

`SELECT CAMPO FROM TABELA1`

Com isso , já temos uma dica de performance: procura NÃO utilizar o '*' para retornar todos os campos , pois isso traz para a estação cliente dados as vezes desnecessários.Exemplo:

`SELECT * FROM TABELA1 - Procure não fazer isso.... e sim:`

`SELECT CAMPO1 , CAMPO2 , CAMPO3 FROM TABELA1`

Bom , quanto à lista de filtros (cláusula where) devemos tomar muito cuidado , pois é ai que geralmente temos problemas ou perda de performance. Para começar , evite utilizar os operadores IN() e subquery's , pois eles oneram a performance do banco.Evite também

instruções muito grandes. Procure quebrá-las em várias instruções e ligue os resultados com o comando UNION.

Cuidado com os operadores lógicos AND na cláusula WHERE pois para cada AND a mais que é colocado , todo conjunto de dados que será retornado tem que ser filtrado. Isto consome muito processamento as vezes desnecessário.

Sempre que possível procure utilizar a cláusula TOP n para indicar qual a quantidade de registro. Saber de antemão quantos registros a query tem que retornar ajuda o SQL Server a fazer um plano de execução da instrução menos e isso diminui o tempo de resposta.Por exemplo , se quisermos somente os 5 primeiros registros que atendem a uma condição:

```
SELECT TOP 5 FLAG1 FROM TABELA1 WHERE FLAG1 = 2
```

Não abuso muito do operador LIKE. Ele é otimo para consultas , mas devemos procurar não colocar % antes e depois:

- Se houver como , evite isto

```
SELECT NOME FROM TABELA1 WHERE NOME LIKE '%A%'
```

Outra dica interessante é não colocar muitos campos na cláusula ORDER BY (ordem da consulta) , pois para cada campo adicional, temos um re-ordenação interna do conjunto de dados retornado. Por exemplo :

```
SELECT CAMPO1 , CAMPO2 , CAMPO3 FROM TABELA1  
ORDER BY CAMPO1 , CAMPO2 , CAMPO3
```

procure usar (quando possível):

```
SELECT CAMPO1 , CAMPO2 , CAMPO3 FROM TABELA1  
ORDER BY CAMPO1
```

Um ponto a ser levantado é o uso de joins. A ordem que se relaciona as tabelas na instrução SELECT não importa. Isto por quê internamente o SQL Server vai escolher a ordem de acesso às tabelas. Por exemplo:

```
SELECT A.CAMPO1 , B.CAMPO2 FROM TABELA1 A , TABELA2 B  
WHERE  
A.COD = B.COD
```

Tem um performance igual se a tabela A possuir mais registros, mesmo que não satisfaçam a condição do join , do que a order inversa :

```
SELECT A.CAMPO1 , B.CAMPO2 FROM TABELA1 A , TABELA2 B  
WHERE  
B.COD = A.COD
```

Lembrando que as duas instruções anteriores retornam SEMPRE o mesmo resultado.

Otimização do PostgreSQL - Introdução em:
http://www.sqlmagazine.com.br/Artigos/Postgre/02_Otimizacao.asp

PostgreSQL Leopard em:
http://www.midstorm.org/~telles/uploads/postgresql_leopardo_conisli_2007.odp

Introdução ao Tuning do SGBD PostgreSQL em:
<http://www.lozano.eti.br/palestras/tunning-pgsql.pdf>

Tutorial completo sobre RAID 0, RAID 1, RAID 0+1 e RAID 5 em:
<http://www.guiadohardware.net/comunidade/raid-tutorial/665151/>

Construção de Comandos SQL com boa performance no PostgreSQL

Em **bancos de dados relacionais** as informações são guardadas em **tabelas**. Para recuperar uma informação necessária ao usuário, deve-se buscá-la em várias tabelas diferentes, estabelecendo-se um **relacionamento** entre elas. Esta é a origem do nome deste paradigma de banco de dados.

Tabelas são na verdade conjuntos. Por exemplo, quando em um sistema existe uma tabela de vendas, esta tabela corresponde ao conjunto de todas as vendas feitas por uma empresa. A tabela de vendedores corresponde ao conjunto de vendedores que trabalham em uma empresa. **Cada linha ou registro da tabela corresponde a um elemento do conjunto.**

Consultas e alterações na base de dados correspondem a operações realizadas sobre conjuntos. Estas operações são definidas pela álgebra relacional.

Por exemplo, **determinar quais os vendedores com tempo de casa maior que um determinado patamar**, significa determinar um subconjunto do conjunto de vendedores, onde todos os elementos possuam uma determinada propriedade. Para determinar as vendas destes vendedores, é necessário realizar a operação já citada e depois, para cada elemento do subconjunto "vendedores veteranos" é necessário determinar o subconjunto do conjunto de vendas, contendo a propriedade de ter sido realizada pelo vendedor em questão. O resultado final da consulta será a união de todos estes subconjuntos.

O problema apresentado possuí também outra forma de solução. Podemos, em um primeiro momento, determinar para cada vendedor, quais as suas vendas. Teríamos então vários subconjuntos, cada um contendo as vendas de um vendedor. Feito isto, podemos verificar quais os vendedores são veteranos, formando o resultado final a partir da união dos subconjuntos associados a cada um.

Consultas em banco de dados não passam de problemas de álgebra relacional. Assim como acontece com a álgebra "tradicional", os operadores possuem algumas propriedades. Sabemos que $2 \times 3 = 3 \times 2$. Isto significa que, quando precisamos contar uma expressão de álgebra relacional para chegar a um determinado resultado, podemos fazê-lo de mais de uma forma, pois várias expressões

levam ao mesmo resultado. Em outras palavras, quando o banco de dados precisa montar uma expressão algébrica para encontrar um resultado, ele deve escolher uma entre várias. Apesar de apresentarem o mesmo resultado, as expressões são diferentes, **e a diferença fará com que o banco de dados adote um diferente caminho para resolver cada uma. Escolher o caminho mais curto é uma das grandes atribuições do banco de dados.** Está é a missão do **otimizador**, um subsistema do banco da dados, responsável por determinar o **plano de execução** para uma consulta.

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

Vamos comparar:

descrição declarativa:

"quero saber todas as vendas feitas por vendedores com mais de 10 anos de casa."

descrição procedural:

"Para cada um dos vendedores, da tabela *vendedores*, com mais de 10 anos de casa, determine na tabela de vendas todas as v @ destes vendedores. A união de todas estas vendas será o resultado final do problema."

Na verdade, fui bem pouco honesto na comparação. Minha declaração procedural tem muito de declarativa, o que a torna mais simples. Se ela fosse realmente procedural, seria ainda mais complicada.

O fato é que, ter que informar o como fazer, torna as coisas mais difíceis. Neste sentido, SQL facilitou muito as coisas.

Porém, a partir do nosso "o que queremos", o banco de dados vai determinar o "como fazer". No problema das "vendas dos veteranos", descrevi duas formas de solucionar o problema, a primeira certamente melhor que a segunda. O objetivo deste texto é apresentar formas de dizer para o banco de dados o que queremos, ajudando-o a determinar uma forma de fazer que tenha esforço mínimo. Para chegarmos a este objetivo, lamentavelmente, teremos que nos preocupar com o "como fazer". É fato que parte da necessidade da nossa preocupação com o "como fazer" é decorrência do estágio atual da tecnologia, que ainda pode evoluir. Porém, por melhor que seja o otimizador de um banco de dados, ele poderá trocar a consulta fornecida pelo usuário por outra equivalente segundo a álgebra relacional. Em algumas situações uma consulta é equivalente à outra apenas considerando-se a semântica dos dados. Neste caso, se nós não nos preocuparmos com o "como fazer", teremos uma performance pior.

Uso de índices

Quando fazemos consultas em uma tabela estamos selecionando registros com determinadas propriedades. Dentro do conceito de álgebra relacional, estamos fazendo uma simples operação de determinar um subconjunto de um conjunto. A forma trivial de realizar esta operação é avaliar cada um dos elementos do conjunto para determinar se ele possui ou não as propriedades desejadas. Ou seja, avaliar, um a um, todos os seus registros.

Em tabelas grandes, a operação descrita acima pode ser muito custosa. Imaginemos que se deseje relacionar todas as apólices vendidas para um determinado cliente, para saber seu histórico. Se for necessário varrer toda a tabela de apólices para responder esta questão o processo certamente levará muito tempo.

A forma de resolver este problema é o uso de índices. Índices possibilitam ao banco de dados o acesso direto às informações desejadas.

Fisicamente, a tabela não está organizada em nenhuma ordem. Os registros são colocados na tabela na ordem cronológica de inserção. Deleções ainda causam mudanças nesta ordem. **Um índice é uma estrutura onde todos os elementos de uma tabela estão organizados, em uma estrutura de dados eficiente, ordenados segundo algum critério.** Um registro no índice é composto pelo conjunto de valores dos campos que compõem o índice e pelo endereço físico do registro na tabela. Ao escrever uma consulta SQL, não informamos especificamente qual índice será usado pela consulta. **Esta decisão é tomada pelo banco de dados.** Cabe a nós escrever a consulta de forma que o uso do índice seja possível. É preciso que nossa consulta *disponibilize* o índice.

Possibilitando uso de colunas para acesso indexado

Na verdade, a consulta disponibiliza colunas que podem ser usadas em acesso à índices. O banco de dados, a partir das colunas disponíveis e dos índices existentes, determina a conveniência de se usar determinado índice.

Para permitir que uma coluna seja usada em acesso à índice, **ela deve aparecer na cláusula *where* de um *select*.**

Por exemplo, a consulta:

```
select campol
from tabela
where campol = 3
      and campo2 > 4
      and campo3 <> 7
      and campo4 between 10 and 20
      and campo5 + 10 = 15
      and to_number(campo6) = 0
      and nvl(campo7, 2) = 2
      and campo8 like 'GENERAL%'
      and campo9 like '%ACCIDENT'
```

Disponibiliza o uso de índices nos campos campo1, campo2 e campo4. Nos casos dos campos campo2 e campo4, o acesso a índice será para buscar registros que possuam valor numa determinada faixa.

A condição *campo3* $<> 7$ não disporúbieza índice por ser uma relação de desigualdade.

De fato, se a tabela possuir n registros, um dos quais com *campo3* = 7, não parece razoável um índice para recuperar N - 1 elementos.

A condição *campo5* + 10 = 15 não permite uso de índice pela coluna *campo5* por igualar ao valor 15 **uma expressão envolvendo a coluna, e não a coluna isolada.** De fato, uma técnica para se inibir o uso de índice em uma coluna, quando desejado, é usar expressões tais como:

- nome-coluna + 0 = 15, para campos *number ou date*, ou
- nome-coluna || ‘ = 'ABCD', para campos *char*.

As expressões envolvendo as colunas *campo6* e *campo7* também não disponibilizam índice pelo mesmo motivo. Foram incluídas aqui por se tratarem de casos em que, freqüentemente, as pessoas acham que o uso de índice seria possível.

A expressão envolvendo *campo8* disponibiliza índice, pois ela é na verdade como se escrevêssemos: *campo8* $\geq 'GE\ cccc\dots\ and\ campo8 \leq 'GEddd\dots' onde **c** é o menor caracter na ordem da tabela ASCII, dentro do domínio possível de valores para o campo é **d o maior**. Já a expressão envolvendo *campo9* não permite uso de índice.$

Se houver um índice único na tabela pelo campo 1, o acesso disponibilizado por este índice é um **unique scan**: o banco de dados faz um acesso ao índice para buscar um único registro.

Mesmo que haja um índice único pelo campo2, será feito um **range scan** na tabela, ou seja, uma busca para recuperar vários registros. O mesmo ocorre para o campo4.

Escolhendo um índice

Dadas as colunas que podem ser usadas para acesso indexado, o banco de dados passa a decisão sobre qual índice será usado. Para isto, ele determinará os índices disponíveis para então escolher um.

Um índice estará disponível se um **prefixo** dos campos que o compõem estiver disponível. Por exemplo, se o índice for composto pelas colunas *campo1*, *campo2*, *campo3* e *campo4*, o índice estará disponível se estiverem disponíveis as colunas:

campo1 ou

campo1 e campo2 ou

campo1, campo2, e campo3 ou

campo1, campo2, campo3 e campo4.

Neste último caso, o uso do índice será completo, se ele for usado.

A seleção entre os índices para determinar qual será realmente usado é feita a partir de heurísticas, como por exemplo, é **melhor usar um índice único que um índice não único**. Esta seleção pode considerar também informações sobre os dados armazenadas na tabela, dependendo da configuração do servidor de banco de dados.

Qual o melhor índice?

O critério básico para escolha de índices é a **seletividade**. Quando o banco de dados resolve uma consulta, freqüentemente, ele precisa percorrer mais registros do que aqueles realmente retomados pela consulta. Os registros percorridos que forem rejeitados representam o trabalho perdido. Quanto menor for o trabalho perdido, mais perto estaremos da performance ótima para resolver a consulta. Portanto, o melhor índice para uma consulta é aquele que apresenta a maior seletividade. Vejamos a consulta abaixo:

```
select campo1
from tabela
where campo2 = 2 and campo3 = 1 and campo4 = 3;
```

tabela possui os índices:

índice 1:	campo2, campo5
índice 2:	campo1
índice 3:	campo3, campo1
índice 4:	campo4
índice 5:	campo5, campo4

Neste caso, estão disponíveis para consultas indexadas os campos *campo2*, *campo3* e *campo4* o que permite o uso dos índices 1, 3 e 4. O índice mais seletivo será aquele que recuperar o mínimo número de registros.

Se houver 10 registros com *campo2* = 2, 2000 registros com *campo3* = 1 e 50 registros com *campo4* = 3, o índice 1 será o mais seletivo. Nossa melhor alternativa é portanto um **range scan** no índice 1. Vale a pena ressaltar que o fato do índice 1 possuir também a coluna *campo5* prejudica um pouco a consulta. Ou seja, seria melhor, para esta consulta, que o índice 1 possuísse apenas o *campo2*.

Para resolver a consulta, o banco de dados fará o acesso ao índice, onde irá recuperar o endereço físico, na tabela, dos registros candidatos a compor o resultado. Com este endereço, ele verifica **cada registro quanto às outras condições**. Os que satisfizerem as outras condições comporão o resultado.

Em alguns casos, este cantinho pode ser mais simples. Veja o exemplo abaixo:

```
Select campo1 from tabela where campo2 = 2;
```

Tabela possui os índices:

índice 1: campo1 , campo3

índice2: campo2, campo1, campo3

índice3: campo1, campo2

Neste caso, o banco de dados apenas pode usar o índice 2. A consulta pode ser resolvida sem acesso à tabela, usando apenas o índice. Uma vez que o índice também possui os valores para *campo1* de cada registro, não há necessidade de se recuperar este valor da tabela.

Campos nulos não entram

Toda vez que um registro possuir valores nulos para todos os campos que compõem um índice, este registro não será colocado no índice.

Isto causa problemas de performance em sistemas mal projetados. Suponha que a modelagem de dados de um sistema de contas a pagar tenha resultado em um projeto onde existe uma tabela de contas (ou compromissos, ou títulos) a pagar, contendo, entre outros, dois campos: *data de vencimento* e *data de pagamento*. A primeira seria a data em que o título deve ser pago. A segunda a data em que o título foi efetivamente pago. Suponha ainda que a dinâmica do sistema determine que todo título, ao ser inserido no sistema, tenha valor nulo para o campo *data de vencimento*. Quando o pagamento vier a ser efetuado, o campo será atualizado. É bastante possível que seja construída uma consulta para recuperar todos os títulos não pagos. Neste caso, não será possível o uso de índices, pois estamos procurando campos com valor nulo. Se tivermos, nesta tabela, 200000 títulos, dos quais 500 não pagos, esta consulta terá desempenho bastante aquém do possível. Uma solução melhor, seria inicializar o campo *data de vencimento* com o valor 01/01/1998, significando conta não paga.

Tabelas pequenas

Como última consideração sobre consultas em uma tabela, vale lembrar que quando fazemos uma consulta a uma tabela bastante pequena, não compensa usar índices. O trabalho envolvido em acessar o índice pegar o endereço e, depois, acessar a tabela é maior que o esforço de ler a tabela inteira.

Consultas em várias Tabelas

Consultas envolvendo várias tabelas são feitas através de *joins*. Na teoria da álgebra relacional, estas consultas são produtos cartesianos entre os conjuntos (tabelas) envolvidos. Para cada elemento do conjunto resultado do produto cartesiano, será verificado se ele possui ou não um determinado conjunto de condições, imposto ao resultado da consulta.

O banco de dados irá tirar proveito de propriedades matemáticas para otimizar estas consultas. Imagine que tenhamos dois conjuntos, um de retângulos e um de círculos. Tome como objetivo obter o subconjunto do produto cartesiano destes dois conjuntos que apenas contenham círculos amarelos e retângulos azuis. Há duas formas de resolver isto. Uma é fazer o produto cartesiano, e, a partir do resultado, excluir os elementos que não atendem a premissa. Outra é determinar o subconjunto dos círculos amarelos e o subconjunto dos retângulos azuis, fazendo um produto cartesiano dos dois subconjuntos. Apesar de equivalentes quanto ao resultado, o segundo método é bastante mais eficiente em termos de performance.

Normalmente, quando se faz uma consulta em duas ou mais tabelas, existe alguma informação que as une. Por exemplo, você quer relacionar registros de um determinado empregado apenas ao registro do departamento onde este empregado trabalha. Esta condição é chamada de **condição de join**. Normalmente, esta condição evita que seja necessária a realização de um produto cartesiano de fato. Vejamos o exemplo:

```
select depto.nome, emp.nome
from   empregados emp, departamentos depto
where  emp.data_admissao < '01-jan-92'
and    depto.id = departamento = emp.departamento and depto.area-de-negocio = 'FAST-FOOD';
Neste caso, estamos trabalhando sobre dois conjuntos: empregados e departamentos. Possuímos restrições sobre estes dois conjuntos e uma restrição que serve para juntá-/os.
```

O banco de dados pode resolver a consulta acima de duas formas. A primeira envolve determinar o subconjunto dos empregados e departamentos que obedecem as restrições nestes conjuntos. Posteriormente, geramos o resultado a partir dos dois subconjuntos, respeitando a **condição de join**. A segunda forma envolve escolher um conjunto para iniciar a solução, por exemplo o de departamentos. Deterrminamos o subconjunto deste que possua a propriedade apresentada (determinada área de negócio). Para cada elemento deste subconjunto, faremos sua associação ao elemento correspondente no outro conjunto, segundo a condição de join. Finalmente, verificamos se o registro formado possui a restrição no outro conjunto. A primeira forma de solução é chamada de **sort-merge join**. A segunda é chamada de **nested-loops**. O banco de dados sempre usa uma destas duas técnicas para resolver consultas em múltiplas tabelas. Ele pode mesmo, combinar as duas.

Na maioria dos casos, o método de **nested-loops** apresenta melhores resultados. Em alguns casos complexos, o **sort-merge** é indicado.

Outer join

O.outer join(ou junção extrema) é um caso particular do join. Quando se faz um join simples, a impossibilidade de se encontrar em alguma tabela um registro associado ao resultado intermediário anterior, determina que o join não retomará nenhuma informação.

Quando uma tabela entra no **join** através de um **outer join**, quando não houver registros na tabela associados ao resultado intermediário, uma linha imaginária será adicionada à tabela em questão, contendo valor nulo para todos os seus campos. Esta linha será juntada aos registros de resultado intermediário, compondo o novo resultado intermediário.

Um Outer join não causa uma lentidão de processamento muito maior que a de um join.

Sub Select

Outro mito que ronda a construção de SQL diz que é melhor usar Sub Select do que usar um join. Dependendo de quem reproduz o mito, a situação se inverte, e o join passa a ser melhor que o Sub Select. Na verdade, deve-se fazer uma análise caso a caso. As

duas técnicas podem inclusive, serem equivalentes.

Vejamos o exemplo:

```
select campo1
from tabela1
where campo2 in (Select campo2
                  from tabela2 where campo4 = constante)
```

e

```
select campol
from tabelal, tabela2
where campo4 = constante and
tabela1.campo2 = tabela2.campo2
```

Neste caso, primeiro devemos notar que as duas consultas só são equivalentes se houver unicidade nos valores do campo3. Caso contrário, pode haver diferença de resultados, quanto ao número de registros retornados. Imagine que as duas tabelas tenham três registros cada uma, todos com o mesmo valor para campo2 (na tabelal) e o mesmo valor para campo3 (na tabela2). Imagine também que todas as linhas da tabelal tenham o mesmo valor para campo 1. No caso da sub select, a consulta retorna três linhas como resultado. No caso do join, retorna 9 linhas. Portanto, para serem equivalentes de fato, é necessário que exista unicidade nos valores de campo3).

Se houver a unicidade, o uso do sub select . ou join (com relação à performance) é absolutamente equivalente. De fato, para resolver esta consulta, em qualquer dos dois casos, o banco de dados irá determinar os registros da tabela2 que possuem "campo4 = constante". Irá recuperar então o valor de campo3 para estes registros. Para cada valor, obterá os registros da tabelal com campo2 igual a este valor. Os valores de campo1 nestes registros comporão os conjuntos resultados.

Vejamos unia situação onde precisamos saber todas as notas fiscais que contenham itens fabricados por um determinado fornecedor, em um determinado dia.

Sub Select:

```
select num nota
from notas Fiscais
where data-emissao = constante and exists (select 'x'
                                              from produtos , itens-nota
                                              where produtos.num-nota = itens-nota.num-nota and
produtos.produto = itens-nota.produto and
fornecedor = constante)
```

Join:

```

select distinct num - nota
from   produtos , itens - nota , notas-fiscais
where  notas-fiscais.data-emissao = constante and
       itens-notas.num-nota = notas-fiscais.num-nota and
       produtos.produto = itens-nota.produto and
       produto.fornecedor = constante

```

Existe uma diferença apreciável de performance entre as duas consultas. Note que, o importante é saber se existe pelo menos um item de uma nota fiscal associado a um determinado fornecedor. Não é necessário determinar **todos** os itens deste fornecedor. Este é o ponto onde a consulta com join torna-se pior, em termos de performance que a consulta com sub select. Note que a data da nota foi incluída para ser um fato seletivo e justificar o exemplo. Se não houvesse tal restrição, o jeito certo de construir a consulta seria determinar todos os produtos do fornecedor, depois os itens de nota com este produto e finalmente as notas correspondentes. Desde, é claro, que um fornecedor seja razoavelmente seletivo. Neste caso, teríamos um exemplo onde join é melhor do que sub select. Apenas por uma diferença sutil de existência de um critério de data.

Dicas de tuning e desempenho para PostgreSQL

<http://www.techforce.com.br/content/postgresql-como-fazer-um-elefante-voar>

Usando o Vacuum e Analyze

Otimizando o Desempenho com Vacuum

Gerenciar e otimizar o espaço em disco. No PostgreSQL as alterações e remoções de registros são apenas marcadas mas o espaço continua ocupado. Para a remoção de fato precisamos executar o comando vacuum. Existe também o comando vacuumdb para a linha de comando.

Sintaxe

```

vacuum [full | freeze | verbose] [tabela]
vacuum [full | freeze |verbose] analyze [tabela]{
campo[,...])]]
```

```
\c intranet
vacuum verbose lotes;
```

O vacuum atualiza a tabela de estatísticas.

```
vacuum analyze perímetros;
```

ANALYZE - Atualiza as estatísticas do banco de dados no catálogo (na tabela pg_statistic).

Após o seu uso o PostgreSQL melhora seu desempenho, tomando decisões mais adequadas para atender às solicitações dos usuários.

Sintaxe

```
analyze [verbose [[tabela[(campo[,...])]]]
```

```
\c intranet
```

```
analyze lotes;
analyze verbose lotes;
```

Autovacuum:

<https://www.postgresql.org/docs/9.6/static/runtime-config-autovacuum.html>

<https://www.postgresql.org/docs/9.6/static/routine-vacuuming.html#AUTOVACUUM>

Atualmente o autovacuum roda como um processo, veja:

```
ps ax | grep postgres
```

```
1325 ? Ss 0:00 postgres: autovacuum launcher process
```

Otimizando o Banco

```
vacuum;
vacuum -a
      -a todos os databases
vacuumdb -f -a -z
      -f
      -a todos os databases
      -z analize
```

Uso do Vacuum**VACUUM**

Vacuum - limpa e opcionalmente analisa um banco de dados. Recupera a área de armazenamento ocupada pelos registros excluídas. Na operação normal do PostgreSQL os registros excluídos, ou tornados obsoletos por causa de uma atualização, não são fisicamente removidos da tabela; permanecem presentes até o comando VACUUM ser executado. Portanto, é necessário executar o comando VACUUM periodicamente, especialmente em tabelas freqüentemente atualizadas.

Sem nenhum parâmetro, o comando VACUUM processa todas as tabelas do banco de dados corrente. Com um parâmetro, o comando VACUUM processa somente esta tabela. O comando VACUUM ANALYZE executa o VACUUM e depois o ANALYZE para cada tabela selecionada. Esta é uma forma de combinação útil para scripts de rotinas de manutenção. Para obter mais detalhes sobre o seu processamento deve ser consultado o comando ANALYZE.

O comando VACUUM simples (sem o FULL) apenas recupera o espaço, tornando-o disponível para ser reutilizado. Esta forma do comando pode operar em paralelo com a leitura e escrita normal da tabela, porque não é obtido um bloqueio exclusivo. O VACUUM FULL executa um processamento mais extenso, incluindo a movimentação das tuplas entre blocos para tentar compactar a tabela no menor número de blocos de disco possível. Esta forma é muito mais lenta, e requer o bloqueio exclusivo de cada tabela enquanto está sendo processada.

Parâmetros

FULL

Seleciona uma limpeza "completa", que pode recuperar mais espaço, mas é muito mais demorada e bloqueia a tabela no modo exclusivo.

FREEZE

Seleciona um "congelamento" agressivo das tuplas. Especificar FREEZE é equivalente a realizar o VACUUM com o parâmetro vacuum_freeze_min_age definido como zero. A opção FREEZE está em obsolescência e será removida em uma versão futura; em vez de utilizar esta opção deve ser definido o parâmetro vacuum_freeze_min_age. (adicionar ao postgresql.conf).

vacuum_freeze_min_age (integer)

Specifies the cutoff age (in transactions) that VACUUM should use to decide whether to replace transaction IDs with FrozenXID while scanning a table. The default is 100000000 (100 million). Although users can set this value anywhere from zero to 1000000000, VACUUM will silently limit the effective value to half the value of autovacuum_freeze_max_age, so that there is not an unreasonably short time between forced autovacuums. For more information see Seção 22.1.3.

A convenient way to examine this information is to execute queries such as SELECT relname, age(relfrozenxid) FROM pg_class WHERE relkind = 'r';
SELECT datname, age(datfrozenxid) FROM pg_database;

VERBOSE

Mostra, para cada tabela, um relatório detalhado da atividade de limpeza.

ANALYZE

Atualiza as estatísticas utilizadas pelo planejador para determinar o modo mais eficiente de executar um comando.

tabela

O nome (opcionalmente qualificado pelo esquema) da tabela específica a ser limpa. Por padrão todas as tabelas do banco de dados corrente.

coluna

O nome da coluna específica a ser analisada. Por padrão todas as colunas.

Executando o Vacuum Manualmente

Para uma tabela

VACUUM ANALYZE tabela;
VACUUM VERBOSE ANALYZE clientes;

```
Para todo um banco
\c nomebanco
VACUUM FULL ANALYZE;
```

Encontrar as 5 maiores tabelas e índices
`SELECT relname, relpages FROM pg_class ORDER BY relpages DESC LIMIT 5;`

Ativando o daemon do auto-vacuum

Iniciando na versão 8.1 é um processo opcional do servidor, chamado de autovacuum daemon, cujo uso é para automatizar a execução dos comandos VACUUM e ANALYZE.

Roda periodicamente e checa o uso em baixo nível do coletor de estatísticas.
 Só pode ser usado se stats_start_collector e stats_row_level forem alterados para true.

Veja detalhes na aula 5 do módulo 2 (Monitorando as Atividades do Servidor do PostgreSQL).

Por default será executado a casa 60 segundos. Para alterar descomente e mude a linha:
`autovacuum_naptime = 60`

Autovacuum

Assim que você entra em produção no 8.0, você vai querer fazer um plano de manutenção incluindo VACUUMs e ANALYZEs. Se seus bancos de dados envolvem um fluxo contínuo de escrita de dados, mas não requer a maciças cargas e apagamentos de dados ou freqüentes reinícios, isto significa que você deve configurar o pg_autovacuum. Isto é melhor que agendar vaccuns porque:

- * Tabelas sofrem o vaccum baseados nas suas atividades, excluindo tabelas que apenas sofrem leituras.

- * A freqüência dos vaccums cresce automaticamente com o crescimento da atividade no banco de dados.

- * É mais fácil calcular o mapa de espaço livre e evitar o inchaço do banco de dados.

Configurando o autovacuum requer a fácil compilação de um módulo do diretório contrib/pg_autovacuum da fonte do seu PostgreSQL (usuários Windows devem procurar o autovacuum incluído no pacote do instalador). Você liga as estatísticas de configuração detalhadas no README. Então você inicia o autovacuum depois de o PostgreSQL ser iniciado como um processo separado; ele será desligado automaticamente quando o PostgreSQL desligar.

As configurações padrões do autovacuum são muito conservadores, imagino, e são mais indicadas para bancos de dados muito pequenos. Eu geralmente uso algo mais agressivo como:

-D -v 400 -V 0.4 -a 100 -A 0.3

Isto irá rodar o vacuum nas tabelas após 400 linhas + 40% da tabela ser atualizada ou apagada e irá rodar o analyze após 100 linhas + 30% da tabelas sofrer inserções, atualizações ou ser apagada. As configurações acima também me permitem configurar o meu max_fsm_pages para 50% das páginas de dados com a confiança de que este número não será subestimado gerando um inchaço no banco de dados. Nós atualmente estamos testando várias configurações na OSDL e teremos mais informações em breve.

Note que você também pode usar o autovacuum para configurar opções de atraso ao invés de configura-lo no PostgreSQL.conf. O atraso no Vacuum pode ser de vital importância em sistemas que tem tabelas e índices grandes; em último caso pode parar uma operação importante.

Existem infelizmente um par de limitações sérias para o autovacuum no 8.0 que serão eliminadas em versões futuras:

* Não tem memória de longa duração: autovacuum esquece toda a sua atividade quando você reinicia o banco de dados. Então se você reinicia regularmente, você deve realizar um VACUUM ANALYZE em todo o banco de dados imediatamente antes ou depois.

* Preste atenção em quanto o servidor está ocupado: há planos de checar a carga do servidor antes de realizar o vacuum, mas não é uma facilidade corrente. Então se você tem picos de carga extremos, o autovacuum não é para você.

Fábio Telles em - <http://www.midstorm.org/~telles/2006/10/>

42 - Gerenciando Bancos de Dados no PostgreSQL

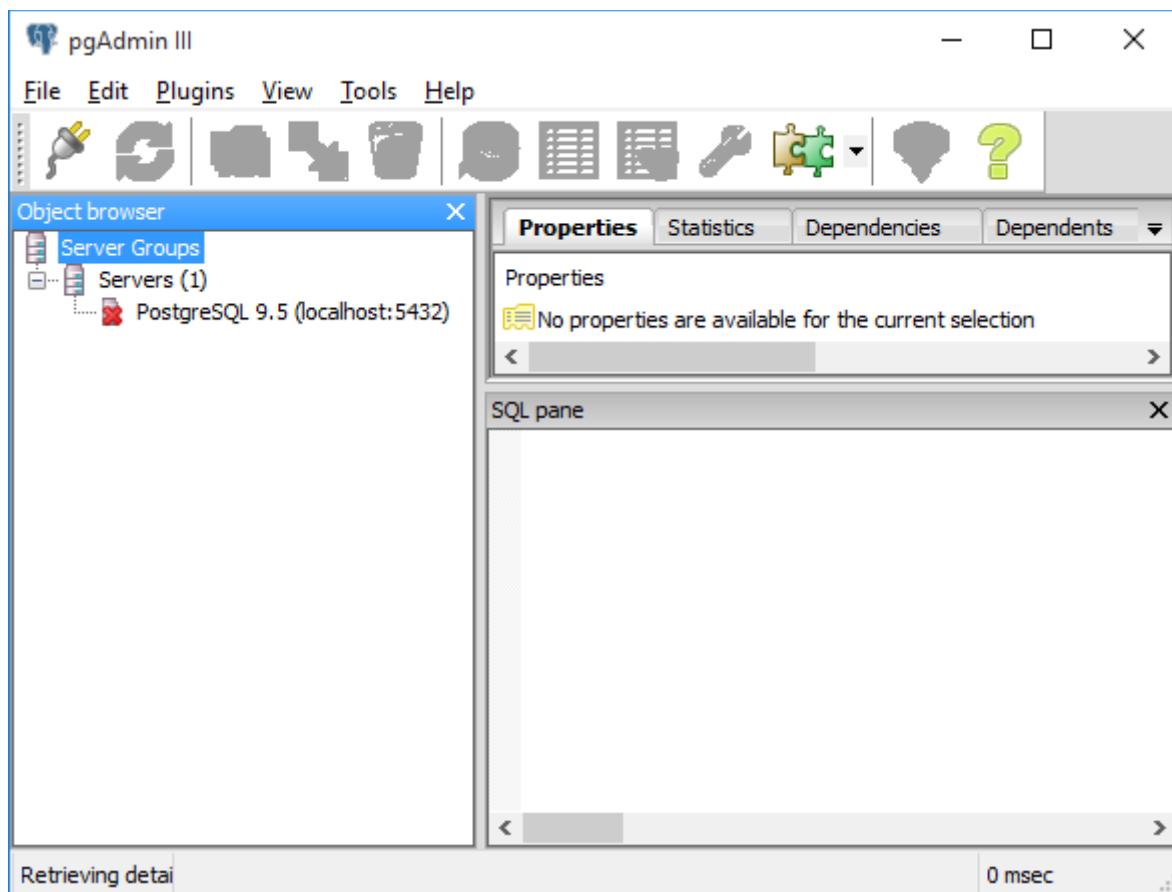
Usaremos os clientes do PostgreSQL: PGAdmin, psql e adminer

PGAdmin – que é um cliente em modo gráfico com bons recursos

psql – que é um cliente para a linha de comando. Este requer que saibamos os comandos e os executemos na linha de comando mas nos oferece mais recursos que as demais ferramentas.

PGAdmin

Se no Windows 10 clique no botão Iniciar e digite PG (já aparece o pgAdmin III). Clique nele para abrir. Será aberta a janela abaixo:



Observe se a porta está de acordo com a que escolhemos na instalação (5432), caso não esteja, clique com o botão direito do mouse sobre PostgreSQL 9.6 ... - Properties e mude a porta.

Agora apenas efetue um duplo clique sobre a conexão criada por default PostgreSQL 9.6 (Localhost:5432)

Para que solicite a senha e abra a conexão.

Então veremos uma tela parecida com esta:

The screenshot shows the pgAdmin III interface. The title bar says "pgAdmin III". The menu bar includes File, Edit, Plugins, View, Tools, and Help. Below the menu is a toolbar with various icons. The main window has a "Object browser" tab selected. On the left, a tree view shows "Server Groups", "Servers (1)", and "PostgreSQL 9.5 (localhost:5432)". Under this server, there are "Databases (1)" containing "postgres", "Tablespaces (2)", "Group Roles (0)", and "Login Roles (2)". On the right, a "Properties" tab is active, displaying the following properties:

Property	Value
Description	PostgreSQL 9.5
Service	
Hostname	localhost
Host Address	
Port	5432
Encryption	not encrypted
SSL Certificate File	
SSL Key File	
SSL Root Certificate File	
SSL Certificate Revocation List	
SSL Compression?	yes
Service ID	postgresql-x64-9.5
Maintenance database	postgres
Username	postgres

Para quem está iniciando no PostgreSQL o PGAdmin ajuda muito, visto que basta ler e clicar. Com o tempo, quando for entendendo mais o psql pode ser mais produtivo, mas é sempre bom usar os dois.

Veja que já temos um banco de dados, o `postgres`, que é um banco criado por default. Praticamente este banco não tem nenhum objeto, especialmente nenhuma tabela. É melhor evitar seu uso. Criemos os nossos bancos ao invés.

Veja

This screenshot shows a detailed view of the "postgres" database structure under the "PostgreSQL 9.5 (localhost:5432)" server in pgAdmin III. The tree view shows:

- Databases (1)**: Contains **postgres**.
- postgres**: Contains:
 - Catalogs (2)**: Contains **Event Triggers (0)** and **Extensions (1)**.
 - Schemas (1)**: Contains **public**.
 - Tables (0)**: Contains **Collations (0)**, **Domains (0)**, **FTS Configuration (0)**, **FTS Dictionaries (0)**, **FTS Parsers (0)**, **FTS Templates (0)**, **Functions (0)**, **Sequences (0)**, and **Tables (0)**.

Selecione o banco postgres – expanda Schemas – expanda public – veja 0 Tables.

Projeto

Vamos organizar as coisas e definir em linhas gerais como será nosso banco de dados, usuários e tabelas, ou seja, elaborar um pequeno projeto.

Um conhecimento muito importante e básico para trabalhar com bancos de dados é a normalização e a modelagem do banco. Existem algumas regras que nos ajudam a ser mais eficientes na construção dos bancos de dados.

Vamos começar com um cadastro de pessoal de uma empresa.

Criamos o banco - rh

Sem pensar muito estaríamos criando a tabela pessoal assim:

```
create table pessoal(
    id serial primary key,
    nome varchar(50) not null,
    cpf varchar(11) not null,
    endereco varchar(100) not null,
    telefone varchar(11),
);
```

E estariamos cadastrando os funcionários assim:

```
id – 1
nome – João da Silva
cpf – 11122233323
endereco – Rua dos Anzóis, 315, Centro, Fortaleza,Ce
telefone – 85 98342 3454
```

Assim existe muita ineficiência. Veja as regras da Terceira Normal para melhorar isso:

Se a empresa tem todos os funcionários morando no ceará, o campo endereço está redundante se repetindo na parte do estado para todos. Idealmente criamos uma tabela para estados, uma para municípios e quem sabe uma para os ceps.

Segunda Etapa

Numa segunda etapa estaremos criando um banco de dados, contendo alguns usuários, esquemas e tabelas nos mesmos.

Banco
pessoal

Usuários

```

diretoria
  diretoria_gab
  diretoria_sec
rh
  rh_admissao
  dp_compensasao
  dp_demissao
ti
  ti_suporte
  ti_redes
  ti_programacao

```

Esquemas

```

diretoria
  diretoria_gab
  diretoria_sec
rh
  rh_admissao
  dp_compensasao
  dp_demissao
ti
  ti_suporte
  ti_redes
  ti_programacao

```

Algumas Tabelas

```

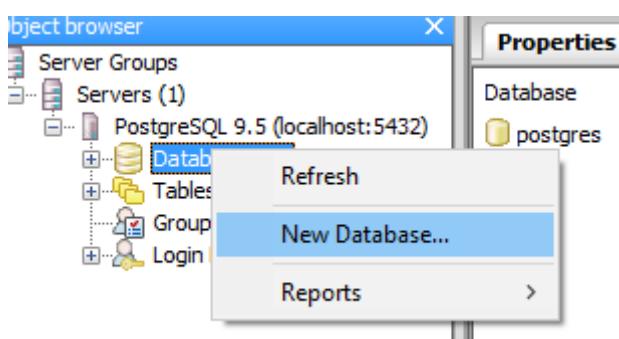
dp_k1
  irrigantes
  lotes

```

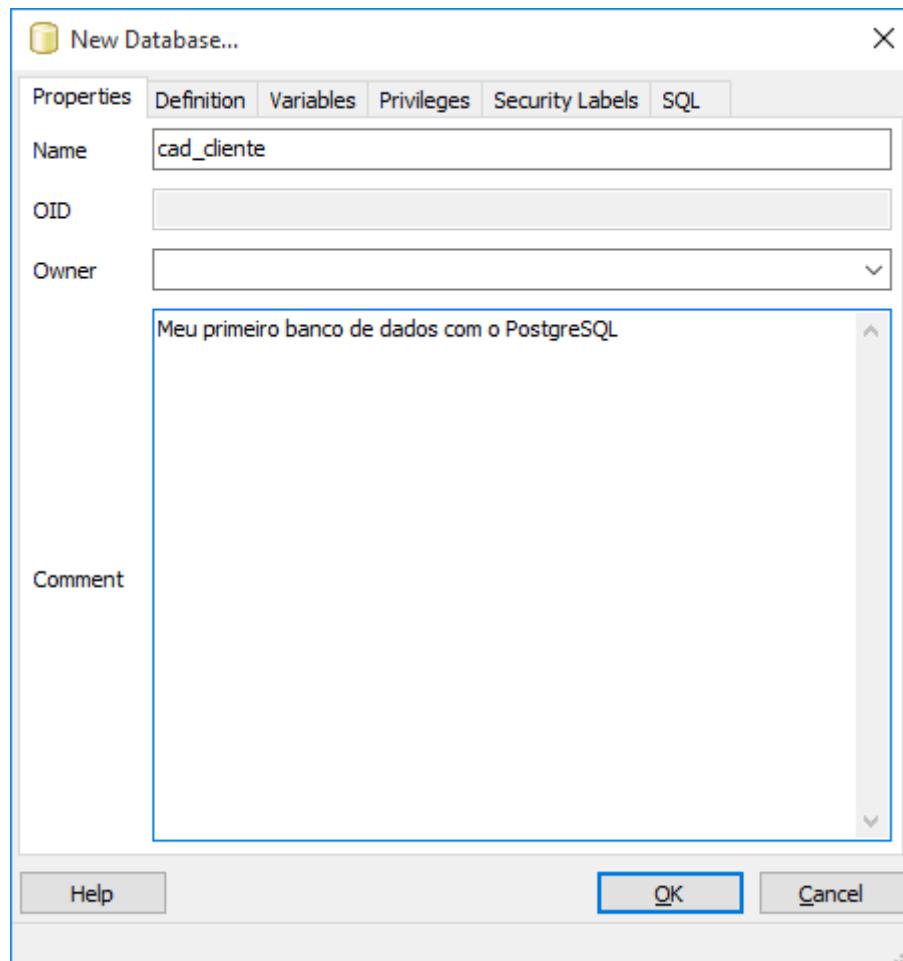
Criando Novo Banco de Dados

Antes de começar a criar bancos de dados devemos fazer um planejamento do que desejamos e precisamos fazer: que bancos criar, usuários, privilégios, tabelas, relacionamentos. É fato que iremos aprendendo tudo isso com o tempo, mas é bom já planejar no início. Não podemos planejar sem ainda conhecer como fazer, mas fique atento para isso.

Uma boa providência é ler o material sobre normalização.
Clique sobre Databases com o botão direito – New Database



Agora basta entrar com o nome do banco
cad_cliente



E clicar em OK

Agora podemos criar esquemas, tabelas e outros objetos no nosso banco de dados, para então adicionar registros e manipular os mesmos.

A quantidade de Recursos Mostrados pelo PGAdmin

Object browser

Catalog Object	Owner
_pg_foreign_dat...	postgres
_pg_foreign_ser...	postgres
_pg_foreign_tabl...	postgres
_pg_foreign_tables	postgres
_pg_user_mappings	postgres
administrable_rol...	postgres
applicable_roles	postgres
attributes	postgres
character_sets	postgres
check_constraint...	postgres
check_constraints	postgres
collation_charact...	postgres
collations	postgres
column_domain_...	postgres
column_options	postgres
column_privileges	postgres
column_udt_usage	postgres
columns	postgres
constraint_colum...	postgres
constraint_table_...	postgres
data_type_privile...	postgres
domain_constraints	postgres
domain_udt_usage	postgres
domains	postgres

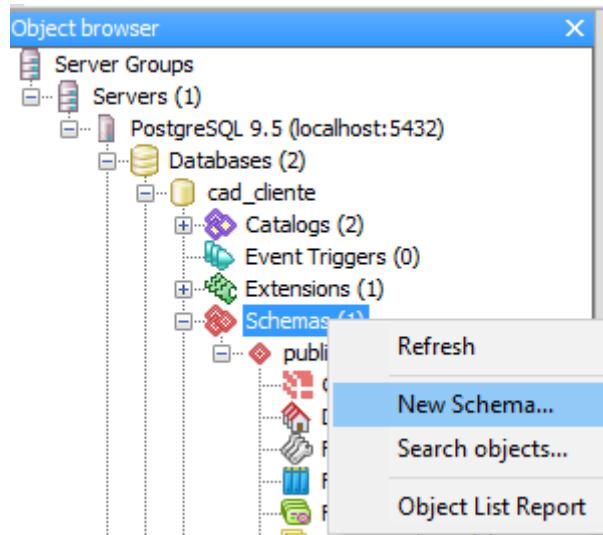
SQL pane

Veja que temos funções (2653), tabelas (54) e views (52) de catálogo, a estorage procedure PI/PgSQL, esquemas, funções, tabelas, etc.

Criar um Novo Esquema

Esquemas são uma forma lógica de organizar as tabelas em um banco de dados, como também de melhorar o controle de acesso de usuários.

Expandir Databases – cad_cliente – Clicar com o botão direito sobre Schemas



Criando uma Tabela

As tabelas são os principais repositórios de dados dos nossos bancos de dados. Nossos registros ficam armazenados nas tabelas.

Expandir Databases – cad_cliente – Schemas – public

43 - Administrando o PostgreSQL pela linha de comando (psql)

Windows:

Conectar ao SGBD

```
cd c:\Arquivos de programas\PostgreSQL\8.2\bin  
psql -U postgres
```

Conectar ao banco cliente

```
\c cliente
```

Consultar registros da tabela cad_clientes

```
select * from cad_clientes;
```

Adicionar registros

```
inser into cad_clientes (codigo, nome, email) values (1, 'João Brito', 'joao@brito.com');
```

Atualizar registros

```
update cad_clientes set nome='Pedro queiroz' where codigo=1;
```

Excluir registros

```
delete from cad_clientes where codigo=1;
```

Linux Ubuntu

A diferença é apenas na conexão ao SGBD, que no caso é assim:

```
su - postgres
```

Obs.: Caso seja a primeira vez que vá conectar então use:

```
sudo passwd postgres
```

Dê uma senha para o usuário postgres e então execute o passo "su - postgres".

Os demais comandos são semelhantes aos executados para o Windows.

Tutorial Básico para Administração do PostgreSQL via PGAdmin da Andréia Duarte Bilenkij:

<http://bilenkij.sites.uol.com.br/TutorialPostgreSQLpgAdmin.pdf>

SHOW

Diversos comandos e configurações podem ser monitoradas com o comando SHOW.
É uma extensão do PostgreSQL muito útil ao administrador.

Deve ser executado com um parâmetro de cada vez.
Não permite o uso combinado com comandos SQL.

Sintaxe:

SHOW nome_parametro;

SHOW ALL;

Exemplos:

Exibindo todos os parâmetros de configuração:

SHOW ALL;

Exibindo o estilo de datas:

SHOW datestyle;

RESET

Este comando devolve o valor original para um parâmetro.

Sintaxe:

RESET nome_parametro;

RESET ALL;

Cuidado com o uso do RESET ALL. Só use com extrema segurança do que está fazendo, desde que pode levar o SGBD a um comportamento inesperado.

SHOW datestyle;
SET datestyle to SQL, DMY;
SHOW datestyle;
RESET datestyle;
SHOW datestyle;

Criando Variáveis de Ambiente no Windows:

SET PGDATA=C:/Program Files/PostgreSQL/8.2/data

Chamando:
echo %PGDATA%

Usando com o comando pg_controldata (devolve diversas informações sobre o PostgreSQL):
pg_controldata

Chamando sem criar a variável:
pg_controldata "C:/Program Files/PostgreSQL/8.2/data"

Executar comandos do sistema operacional:

! ls

44 - Administração com o cliente web Adminer

<http://adminer.org>

Criação de diagramas DER com o dbVisualizer

<http://www.dbvis.com/>

45 - Instalação do PostgreSQL através dos Fontes

Conhecimentos requeridos:

Para compilar algo no linux subentende-se que você tenha pelo menos alguns conhecimentos anteriores: uso do terminal do Linux, compactar e descompactar arquivos, uso de shell script e da compilação de programas, além da instalação de pacotes da distribuição.

Compilação do PostgreSQL 9.5.3

A compilação permite interferir nas características do postgresql instalado e ter mais controle do que temos ao final do que instalando pelos pacotes. Na configuração podemos indicar em que diretório ficarão os binários do nosso SGBD, em que diretório ficará o cluster com nossos bancos de dados.

Usar os fontes permite muita flexibilidade. Permite até que usemos uma distribuição para a qual ainda não existam pacotes. Não precisamos nos preocupar com qual distribuição usar, a arquitetura, etc.

Para uma experiência mais simples iremos começar:

- Parando o postgresql existente
- Dexinstando completamente

```
sudo service postgresql stop  
sudo apt-get remove --purge postgresql postgresql-contrib
```

Plataformas suportadas:

<https://www.postgresql.org/docs/9.5/static/supported-platforms.html>

Notas para algumas plataformas:

<https://www.postgresql.org/docs/9.5/static/installation-platform-notes.html>

Resumo dos procedimentos de compilação para quem já sabe o que fazer:

<https://www.postgresql.org/docs/9.5/static/install-short.html>

Pré-requisitos

<https://www.postgresql.org/docs/9.5/static/install-requirements.html>

Download

<https://www.postgresql.org/ftp/source/>

Na página acima temos as versões da 1.08 até a 9.6.3. (Hoje)

A recomendação é usar a versão estável mais recente. No caso baixarei a 9.5.3.

Baixar o bz2, que é menor

<https://ftp.postgresql.org/pub/source/v9.5.3/postgresql-9.5.3.tar.bz2>

<https://ftp.postgresql.org/pub/source/v9.5.3/postgresql-9.5.3.tar.bz2.md5>

Uma boa sugestão é usar o wget para fazer o download:

wget -c <https://ftp.postgresql.org/pub/source/v9.5.3/postgresql-9.5.3.tar.bz2>

Checar autenticidade do pacote baixado:

md5sum -c postgresql-9.5.3.tar.bz2

SUCESSO

Descompactar

bunzip2 postgresql-9.5.3.tar.bz2 (criará o arquivo postgresql-9.5.3.tar)

sudo su

tar xpvf postgresql-9.5.3.tar -C /usr/local/src

cd /usr/local/src/

mv postgresql-9.5.3 psql

Veja o conteúdo dos fontes:

cd psql

Listar somente os diretórios do primeiro nível

find ./ -maxdepth 1 -type d

/config - configurações

/contrib - diversos pacotes recebidos de contribuições que não fazem parte do core

/doc - documentação

/src - core com muita coisa

Instalar dependências:

Header do kernel

apt-get install build-essential linux-headers-\$(uname -r)

Outras dependências

apt-get install ncurses-dev bison gawk python-setproctitle zlib1g-dev

apt-get install libreadline6 libreadline6-dev ledit gcc g++ zlibc gettext

readline - importante para o psql

zlib - importante para backups com pg_dump

No Ubuntu o make é o gmake. Caso queira pode criar um link simbólico:

sudo ln -s /usr/bin/make /usr/bin/gmake

Guardar o help da configuração

./configure --prefix=/usr/local/pgsql > configure.txt

Configurar, indicando o diretório do PostgreSQL

No configure podemos passar vários parâmetros que customizam a instalação. Eles produzirão um makefile que será usado pelo make na instalação. Podemos passar o diretório de destino, número da porta padrão, habilitar e desabilitar várias características e diretórios.

```
./configure --prefix=/usr/local/pgsql
```

Guardar o make (esta etapa talvez não valha a pena. A saída não deve ser muito interessante):

```
make world > make_world.txt
```

Compilar tudo: core, docs e contribs:

```
make world
```

Caso receba a mensagem:

PostgreSQL, contrib, and documentation successfully made. Ready to install.

Significa que foi bem sucedido na compilação e pode agora instalar, como a seguir.

Antes vamos fazer os Testes de Regressão

Os testes de regressão são uma forma de testar pra valer o suporte do seu servidor ao PostgreSQL.

Mas para ser executado, você deve deixar os fontes na pasta de um usuário comum e executar os mesmos como usuário comum e não como root. Lembre que toda a pasta com os fontes deve ter o usuário como dono e dar permissão de escrita para ele.

```
sudo cp -ra /usr/local/src/pgsql /home/ribafs
sudo chmod -R 755 /home/ribafs/pgsql
sudo chown -R ribafs:ribafs /home/ribafs/pgsql
```

Para executar os testes de regressão, estando na home do usuário, execute:

```
cd /home/ribafs/pgsql
```

```
make check
```

Aqui ao final recebi:

```
All 157 tests passed.
```

Mais detalhes:

<https://www.postgresql.org/docs/9.5/static/install-procedure.html>

Guardar o resultado da instalação:

```
make install-world > make_install_world.txt
```

Instalar

```
make install-world
```

Aguarde...

Como a última mensagem foi:

PostgreSQL, contrib, and documentation installation complete.

Nossa instalação foi bem sucedida.

Prontinho os fontes do PostgreSQL estão instalados em sua máquina. Mas lembre que apenas instalamos, seu processo não está em execução, nem sua porta está aberta nem o cluster foi criado ainda.

Criar o super usuário do PostgreSQL:

```
groupadd postgres
useradd -g postgres -d /usr/local/pgsql postgres
chown -R postgres:postgres /usr/local/pgsql
passwd postgres
```

Criar o diretório data. Como instalamos em /usr/local/pgsql, criaremos o data em:
mkdir /usr/local/pgsql/data

Mas é bom lembrar que este diretório, que guardará todo o cluster, pode ficar em qualquer diretório. Idealmente em um disco mais rápido para os bancos de dados.

Tornar o postgres seu dono:

```
chown -R postgres:postgres /usr/local/pgsql/data
```

Criar o novo cluster:

```
su - postgres
```

Adicionar /usr/local/pgsql/bin ao PATH:

```
nano ~/.bash_profile
```

Adicione isto:

```
if [ -d "$HOME/bin" ] ; then
    PATH="$PATH:$HOME/bin"
fi
```

```
PATH=$PATH:/usr/local/pgsql/bin
PGDATA=/usr/local/pgsql/data
LD_LIBRARY_PATH=/usr/local/pgsql/lib
export PATH
export PGDATA
export LD_LIBRARY_PATH
```

Atualizar ambiente:

```
exit
```

```
su - postgres
```

Criar o novo cluster

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

Se aparecer a mensagem:

Success. You can now start the database server using:

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

Foi bem sucedido.

Detalhes sobre o initdb:

<https://www.postgresql.org/docs/current/static/app-initdb.html>

Veja os diretórios do novo cluster:

```
cd /usr/local/pgsql/data
find ./ -maxdepth 1 -type d
./global
./pg_stat
./pg_replslot
./base
./pg_logical
./pg_twophase
./pg_clog
./pg_notify
./pg_xlog
./pg_dynshmem
./pg_stat_tmp
./pg_multixact
./pg_serial
./pg_tblspc
./pg_subtrans
./pg_snapshots
./pg_commit_ts
```

Caso queira mudar a porta default (5432) para outra:

nano /usr/local/pgsql/data/postgresql.conf

Adicionar a variável PGDATA

Iniciar o novo postgresql instalado

pg_ctl start

Detalhes sobre o pg_ctl

<https://www.postgresql.org/docs/current/static/app-pg-ctl.html>

Para verificar se processo postgres foi ativado:

ps -A | grep postgres

Caso haja instalado e desinstalado o postgresql através dos pacotes, ele pode ter deixado alguns binários instalados, como é o caso do psql. Para se certificar execute:

which psql

Se retornar /usr/bin/psql então é o binário deixado pelos pacotes, visto que o nosso está em /usr/local/pgsql/bin. Então execute assim.
 Exclua o /usr/bin/psql, feche todas as janelas do terminal abertas e abra novamente que agora o psql encontrado será este. Não só o psql, mas o createdb e outros precisará excluir para que seja encontrado o da compilação.

Aparecerão 6 processos. Caso apareçam 12, os outros 6 são do outro postgresql.

Criar um banco teste no nosso novo postgresql:
 createdb teste

Entrar na console

```
psql
\l
```

Executando comandos do sistema operacional dentro do psql
 ! comando

```
! ls
```

Vamos adicionar ao PATH

Se no Ubuntu ou outro Debian:

Para todos os usuários

```
sudo nano /etc/profile (e adicione a linha):
PATH=$PATH:/usr/local/pgsql/bin
export PATH
```

Depois execute:

```
exit
```

login novamente

Para apenas um usuário

```
nano ~/.bash_profile
```

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

Depois execute:

```
exit
```

login novamente

initdb – inicializa o cluster, cria os scripts de configuração default.

postmaster – inicia o processo do servidor responsável por escutar por pedidos de conexão.

Caso em alguma etapa se perca e queira apagar tudo e recomeçar, precisará destruir os processos:
 killall postgres

Automatizar o Postgresql

Antes de tudo, vamos parar os serviços postgresql que estejam rodando:

```
killall postgres
```

Agora, aos procedimentos de automatização:

```
cp /usr/local/src/pgsql/contrib/start-scripts/linux /etc/init.d/postgresql
```

```
chmod +x /etc/init.d/postgresql
```

Agora o administrador já poderá utilizar os comandos:

```
/etc/init.d/postgresql stop  
/etc/init.d/postgresql start  
/etc/init.d/postgresql restart
```

Agora já podemos proceder da mesma forma como procediamos com os pacotes instalados:

```
service postgresql stop  
service postgresql start
```

```
su - postgres  
psql
```

```
\l
```

Referências:

<https://www.postgresql.org/docs/9.5/static/installation.html>

https://pt.wikibooks.org/wiki/PostgreSQL_Pr%C3%A1tico/Instala%C3%A7%C3%A3o>No_Linux

<https://concani3.wordpress.com/2012/09/04/compilar-postgresql-e-fazer-a-instalacao-manual-em-ambiente-linux-debian/>

Contribs

Alguns programadores desenvolvem ferramentas, módulos e exemplos que são úteis a quem trabalha com PostgreSQL.

Como sua utilidade é restrita e também a equipe pretende manter o core do PostgreSQL o menos possível, as contribs não são incorporados ao PostgreSQL. Com o tempo, quando alguma destas contribuições tornam-se muito importantes ela acaba por ser incorporada ao core, como foi o caso da T-Search agora na versão 8.3.

Relação de contribs do 9.5.3:

Compilando e instalando:

```
cd contrib
make
make install

adminpack
auth_delay
auto_explain
btree_gin
btree_gist
chkpass
citext
cube
dblink
dict_int
dict_xsyn
earthdistance
file_fdw
fuzzystrmatch
hstore
hstore_plperl
hstore_plpython
intagg
intarray
isn
lo
ltree
ltree_plpython
oid2name
pageinspect
passwordcheck
pg_buffercache
pgcrypto
pg_freespacemap
pg_prewarm
pgrowlocks
```

```

pg_standby
pg_stat_statements
pgstattuple
pg_trgm
postgres_fdw
seg
sepgsql
spi
sslinfo
start-scripts
tablefunc
tcn
test_decoding
tsearch2
tsm_system_rows
tsm_system_time
unaccent
uuid-ossp
vacuumlo
xml2

```

Importar uma contrib no Windows:

```
C:\Program Files\PostgreSQL\8.2\bin>psql -U postgres -d dnocs < ..\share\contrib\cube.sql
```

No Linux ou instalamos o pacote dos contribs ou compilamos o diretório contrib nos fontes, como também podemos compilar somente o diretório do cube.

Algumas Contribs

cube

Traz um tipo de dados cubo, que pode ser tridimensional ou com cinco ou seis dimensões.

\c dnocs

-- Definindo um cubo:

```
SELECT '4'::cube AS cube;
```

-- Definindo um ponto no espaço:

```
SELECT '4, 5, 6'::cube AS cube;
```

-- Definindo uma caixa n-dimensional representada por um par de pontos opostos:

```
SELECT '(0,0,0,0),(1,-2,3,-4)'::cube AS cube;
```

-- O cubo inicia em (0,0,0,0) e termina em (1,-2,3,-4).

-- Ao definir um cubo devemos ficar atento para que os pontos tenham a mesma dimensionalidade:

-- Calcular a interseção de dois cubos:

```
SELECT cube_inter('1,2,3,4),(0,0,0,0)', '(0,0,0,0),(-1,-2,-3,-4)');
```

-- Mostrará o ponto em comum entre os cubos.

-- União entre cubos

```
SELECT cube_union('(1,2,3,4),(0,0,0,0)', '(0,0,0,0),(-1,-2,-3,-4)');
```

```
SELECT cube_union(cube_union('(1,2,3,4),(0,0,0,0)', '(0,0,0,0),(-1,-2,-3,-4)'), '(0,0,0,0),(9,-10,11,-12)');
```

-- Localizando certo ponto em um cubo

```
SELECT cube_contains('(1,-2,3,-4),(0,0,0,0)', '(0,-1,1,-2)');
```

-- Podemos usar os operadores < e > em cubos:

```
SELECT '(0,0,0,0),(1,2,3,4)::cube < '(0,0,0,0),(2,2,3,4)::cube;
```

-- O operador << procurar um cubo à esquerda de outro cubo

```
SELECT '(-2,-3),(-1,-2)::cube << '(0,0),(2,2)::cube;
```

-- O operador >> procurar um cubo à direita de outro cubo

Cubos e Índices

```
CREATE TABLE mycubes(a cube DEFAULT '0,0'::cube);
```

Definindo um índice do tipo GiST:

```
CREATE INDEX cube_idx ON mycubes USING gist (a);
```

```
create table mytexts(a varchar primary key);
```

```
insert into mytexts values ('Joao');
insert into mytexts values ('Pedro');
insert into mytexts values ('Ribamar');
insert into mytexts values ('Manoel');
```

Para garantir que o SGBD não executará uma varredura sequencial (sequential scan), então executaremos:

```
SET enable_seqscan TO off;
```

```
SELECT * FROM mytexts WHERE a='Pedro';
```

Caso enable_seqscan tivesse como on o PostgreSQL executaria uma varredura sequencial, pois a tabela é muito pequena em termos de registros.

A situação muda se ao invés de usarmos = usarmos ~.

```
SELECT * FROM mytexts WHERE a ~ 'Pedro';
```

```
EXPLAIN SELECT * FROM mytexts WHERE a ~ 'Pedro';
```

Veja que agora o PostgreSQL voltou a usar o sequential scan, mesmo desabilitado.

Trabalhando com ISBN e ISSN

A contrib destes está no arquivo isn.sql.

International Standard Book Number (ISBN) e International Standard Serial Number (ISSN).

```
CREATE TABLE myisbn(name text, number isbn);
```

```
INSERT INTO myisbn VALUES('Apache Administration', '3-8266-0554-3');
```

```
SELECT * FROM myisbn;
```

Testando um ISBN inválido:

```
INSERT INTO myisbn VALUES('no book', '324324324324234');
```

```
SELECT * FROM myisbn WHERE number>'3-8266-0506-3'::isbn;
```

Instalando PostgreSQL no Slackware Linux

Autor: Diego Pereira Grassato <diego_mldo@hotmail.com>

Data: 08/06/2009

Instalando PostgreSQL no Slackware Linux

Por padrão o Slackware Linux não disponibiliza o PostgreSQL em sua distribuição.

Baixaremos a versão já compilada para o Slackware:

```
# wget http://repository.slacky.eu/slackware-12.2/database/postgresql/8.3.7/postgresql-8.3.7-i486-2sl.tgz
```

Criaremos o diretório onde ficarão os arquivos do banco dados:

```
# mkdir -p /var/lib/pgsql
```

Daremos agora permissões:

```
# chown pgsql:pgsql /var/lib/pgsql -R
```

Daremos permissão ao arquivo de execução do PostgreSQL.

```
# chmod +x /etc/rc.d/rc.pgsql
```

Iniciaremos agora o banco de dados:

```
# su - pgsql
$ initdb -D /var/lib/pgsql/data
$ exit
```

Pronto, seu PostgreSQL está devidamente configurado. Feito isto basta iniciar o banco de dados:

```
# /etc/rc.d/rc.pgsql start
```

```
Starting PostgreSQL...
server starting
```

Configurações do SGDB

Neste momento o banco de dados está plenamente funcional, é necessário no entanto configurar o SGDB para torná-lo seguro. O mínimo a ser efetuado seria algo do tipo:

```
# su postgres
$ psql postgres
postgres=# alter user postgres with password 'senha';
```

O comando acima foi para definir uma senha para o superusuário postgres.

Criaremos um usuário para acessar o Postgres:

```
postgres=# create user diego;
postgres=# alter user diego with password 'senha';
```

E editar o arquivo /var/lib/pgsql/pg_hba.conf para restringir quem pode acessar o banco de dados e como. O ideal seria comentar todas as três entradas default e acrescentar algo como:

```
# cat pg_hba.conf | grep -Ev '^#|^$'
local  postgres      postgres          password
host   postgres      postgres          127.0.0.1/32    password
host   banco        conta            a.b.c.d/32     password
```

Conexões internas e externas

Se não quiser apenas conexões locais (via socket), adicione as variáveis no arquivo de inicialização do Postgres BIND e HOST. HOST é a variável que guardará o endereço onde o servidor irá aguardar novas conexões, editando o arquivo:

```
# vim /etc/rc.d/rc.pgsql
```

Adicionar as variáveis (no lugar de 192.168.2.10 o IP da máquina):

```
HOST=192.168.2.10
BIND="-o '-i -h $HOST"
```

Na linha onde contém (su pgsql -c "pg_ctl start -D \$DBCLUSTER -l \$SERVERLOG), altere para:

```
su pgsql -c "pg_ctl start -D $DBCLUSTER -l $SERVERLOG $BIND"
```

Para acessar pela rede será desta forma:

```
# psql -h 192.168.2.10 banco -U conta
banco=>
```

Se você estiver no Windows e estiver usando algum cliente PostgreSQL como PGADMIN III, conseguirá acessar normalmente.

<http://www.vivaolinux.com.br/dica/Instalando-PostgreSQL-no-Slackware-Linux>

Instalação do PostgreSQL no CentOS

Instalando PostgreSQL no Slackware Linux

Autor: Diego Pereira Grassato <diego_mldo@hotmail.com>

Data: 08/06/2009

Instalando PostgreSQL no Slackware Linux

Por padrão o Slackware Linux não disponibiliza o PostgreSQL em sua distribuição.

Baixaremos a versão já compilada para o Slackware:

```
# wget http://repository.slacky.eu/slackware-12.2/database/postgresql/8.3.7/postgresql-8.3.7-i486-2sl.tgz
```

Criaremos o diretório onde ficarão os arquivos do banco dados:

```
# mkdir -p /var/lib/pgsql
```

Daremos agora permissões:

```
# chown pgsql:pgsql /var/lib/pgsql -R
```

Daremos permissão ao arquivo de execução do PostgreSQL.

```
# chmod +x /etc/rc.d/rc.pgsql
```

Iniciaremos agora o banco de dados:

```
# su - pgsql
$ initdb -D /var/lib/pgsql/data
$ exit
```

Pronto, seu PostgreSQL está devidamente configurado. Feito isto basta iniciar o banco de dados:

```
# /etc/rc.d/rc.pgsql start
```

```
Starting PostgreSQL...
server starting
```

Configurações do SGDB

Neste momento o banco de dados está plenamente funcional, é necessário no entanto configurar o SGDB para torná-lo seguro. O mínimo a ser efetuado seria algo do tipo:

```
# su postgres
$ psql postgres
postgres=# alter user postgres with password 'senha';
```

O comando acima foi para definir uma senha para o superusuário postgres.

Criaremos um usuário para acessar o Postgres:

```
postgres=# create user diego;
postgres=# alter user diego with password 'senha';
```

E editar o arquivo /var/lib/pgsql/pg_hba.conf para restringir quem pode acessar o banco de dados e como. O ideal seria comentar todas as três entradas default e acrescentar algo como:

```
# cat pg_hba.conf | grep -Ev '^#|^$'
local  postgres      postgres          password
host   postgres      postgres      127.0.0.1/32      password
host   banco        conta       a.b.c.d/32      password
```

Conexões internas e externas

Se não quiser apenas conexões locais (via socket), adicione as variáveis no arquivo de inicialização do Postgres BIND e HOST, HOST é a variável que guardará o endereço onde o servidor irá aguardar novas conexões, editando o arquivo:

```
# vim /etc/rc.d/rc.pgsql
```

Adicionar as variáveis (no lugar de 192.168.2.10 o IP da máquina):

```
HOST=192.168.2.10
BIND="-o '-i -h $HOST"
```

Na linha onde contem (su pgsql -c "pg_ctl start -D \$DBCLUSTER -l \$SERVERLOG), altere para:

```
su pgsql -c "pg_ctl start -D $DBCLUSTER -l $SERVERLOG $BIND"
```

Para acessar pela rede será desta forma:

```
# psql -h 192.168.2.10 banco -U conta
banco=>
```

Se você estiver no Windows e estiver usando algum cliente PostgreSQL como PGADMIN III, conseguirá acessar normalmente.

<http://www.vivaolinux.com.br/dica/Instalando-PostgreSQL-no-Slackware-Linux>

Instalação do PostgreSQL no OpenBSD

OpenBSD

The postgresql package is available from [ftp.openbsd.org](ftp://ftp.openbsd.org) or on CD's that came with your distribution. You can install it using `pkg_add`. Once the package is installed, you should run the following commands:

```
useradd -m postgres
passwd postgres (assign the new user a password)
mkdir /usr/local/pgsql /usr/local/pgsql/var
chown -R postgres:users /usr/local/pgsql
su postgres
initdb -D /usr/local/pgsql/var
exit
```

`/var/lib/pgsql`

`/usr/local/pgsql`

`/usr/local/etc/rc.d/010.pgsql.sh start`

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

Testes de regressão

Os testes de regressão compõem um conjunto detalhado de testes da implementação da linguagem SQL no PostgreSQL. São testadas as operações SQL padrão, assim como as funcionalidades estendidas do PostgreSQL.

<http://pgdocptbr.sourceforge.net/pg80/regress.html>

1. Execução dos testes

Os testes de regressão podem ser executados em um servidor já instalado e em funcionamento, ou utilizando uma instalação temporária dentro da árvore de construção. Além disso, existem os modos "paralelo" e "seqüencial" para execução dos testes. O modo seqüencial executa cada um dos scripts de teste por vez, enquanto o modo paralelo inicia vários processos servidor para executar grupos de teste em paralelo. Os testes em paralelo dão a confiança de que a comunicação entre processos e os bloqueios estão funcionando de forma correta. Por motivos históricos, os testes seqüenciais são geralmente executados em uma instalação existente, e o modo paralelo em uma instalação temporária, mas não há motivo técnico para ser feito assim.

Para executar os testes de regressão após construir, mas antes de instalar, deve ser digitado

```
gmake check
```

no diretório de nível mais alto (Ou o diretório src/test/regress pode ser tornado o diretório corrente e o comando executado neste diretório). Primeiro serão construídos vários arquivos auxiliares, como algumas funções de gatilho de exemplo definidas pelo usuário e, depois, executado o script condutor do teste. Ao final deve ser visto algo parecido com

```
=====
All 96 tests passed.
=====
```

ou, senão, uma nota sobre quais testes falharam. Deve ser vista a [Seção 2](#) abaixo antes de assumir que "failure" representa um problema sério.

Uma vez que este método executa um servidor temporário, não funciona quando se está logado como o usuário root (uma vez que o servidor não inicializa sob o root). Se foi feita a construção como root, não será necessário começar tudo novamente. Em vez disto, deve ser feito com que o diretório do teste de regressão possa ser escrito por algum outro usuário, se conectar como este usuário, e recomeçar os testes. Por exemplo:

```
root# chmod -R a+w src/test/regress
root# chmod -R a+w contrib/spi
root# su - joeuser
joeuser$ cd diretório_de_construção_de_nível_mais_alto
joeuser$ gmake check
```

(O único "risco de segurança" possível neste caso é a possibilidade de outro usuário alterar os resultados do teste de regressão sem que se saiba. Use o bom senso ao gerenciar permissões para usuários).

Como alternativa, os testes podem ser executados após a instalação.

Se o PostgreSQL for configurado para ser instalado em um local onde existe uma instalação mais antiga do PostgreSQL, e for executado gmake check antes de instalar a nova versão, pode ser que os testes falhem devido aos novos programas tentarem utilizar as bibliotecas compartilhadas já instaladas (O sintoma típico é a reclamação sobre símbolos não definidos). Se for desejado executar os testes antes de sobrescrever a instalação antiga, será necessário fazer a construção utilizando configure --disable-rpath. Entretanto, não se recomenda que esta opção seja utilizada na instalação final.

O teste de regressão paralelo inicia alguns processos sob o ID do usuário. Atualmente, a simultaneidade máxima são vinte scripts de teste em paralelo, o que significa sessenta processos: para cada script de teste existe um processo servidor, o psql e, geralmente, o processo ancestral do interpretador de comandos que chamou o psql. Portanto, se o sistema impõe limite para o número de processos por usuário, certifique-se que este limite

é de setenta e cinco ou mais, senão podem acontecer falhas aleatórias no teste paralelo. Se não houver condição de aumentar este limite, pode ser diminuído o grau de paralelismo definindo o parâmetro `MAX_CONNECTIONS`. Por exemplo,

```
gmake MAX_CONNECTIONS=10 check
```

não executa mais de dez testes ao mesmo tempo.

Em alguns sistemas o interpretador de comandos padrão compatível com o Bourne (`/bin/sh`) se confunde ao gerenciar tantos processos descendentes em paralelo, podendo fazer com que o teste paralelo trave ou falhe. Neste caso, deve ser especificado na linha de comandos um interpretador de comandos compatível com o Bourne diferente como, por exemplo:

```
gmake SHELL=/bin/ksh check
```

Se não estiver disponível nenhum interpretador de comandos que não apresente este problema, então este problema poderá ser contornado limitando o número de conexões, conforme mostrado acima.

Para executar os testes após a instalação (consulte o [Capítulo 14](#)), deve ser inicializada a área de dados e ativado o servidor e depois, conforme explicado no [Capítulo 16](#), deve ser digitado:

```
gmake installcheck
```

ou para o teste em paralelo

```
gmake installcheck-parallel
```

Os testes esperam fazer contato com o servidor no hospedeiro local no número de porta padrão, a menos que esteja especificado o contrário nas variáveis de ambiente PGHOST e PGPORT.

2. Avaliação dos testes

Algumas instalações do PostgreSQL, devidamente instaladas e inteiramente funcionais, podem "falhar" em alguns testes de regressão por causa de algumas particularidades específicas da plataforma, como a representação diferente do ponto flutuante ou do suporte à zona horária. Atualmente a avaliação dos testes é feita simplesmente usando o programa `diff`, para comparar os resultados obtidos pelos testes com os resultados produzidos no sistema de referência e, portanto, os testes são sensíveis a pequenas diferenças entre sistemas. Quando for relatado que o teste "falhou", sempre deve ser examinada a diferença entre o resultado esperado e o resultado obtido; pode-se descobrir que as diferenças não são significativas. Ainda assim, são feitos esforços para manter os arquivos de referência idênticos entre todas as plataformas suportadas, portanto deve-se esperar que todos os testes sejam bem-sucedidos.

As saídas produzidas pelos testes de regressão ficam nos arquivos do diretório `src/test/regress/results`. Os scripts dos testes utilizam o programa `diff` para comparar cada arquivo de saída produzido com a saída de referência armazenada no diretório `src/test/regress/expected`. Todas as diferenças são salvas em `src/test/regress/regression.diffs` para poderem ser inspecionadas (ou pode ser executado o programa `diff` diretamente, se for preferido).

2.1. Diferenças nas mensagens de erro

Alguns testes de regressão envolvem, intencionalmente, valores de entrada inválidos. As mensagens de erro podem ser originadas pelo código do PostgreSQL, ou pelas rotinas do sistema da plataforma hospedeira. No último caso, as mensagens podem variar entre plataformas, mas devem conter informações semelhantes. Estas diferenças nas mensagens resultam em testes de regressão que "falham", mas que podem ser validados por inspeção.

2.2. Diferenças no idioma

Se os testes forem executados em um servidor já instalado, que foi inicializado com o idioma da ordem de intercalação (*collation-order*) [\[1\]](#) [\[2\]](#) [\[3\]](#) [\[4\]](#) diferente de C, então poderão haver diferenças por causa da ordem de classificação e falhas de continuação. O conjunto de testes de regressão é configurado para tratar este problema mediante arquivos de resultado alternativos, que juntos podem tratar um grande número de idiomas. Por exemplo, para o teste `char` o arquivo esperado `char.out` trata os idiomas C e POSIX, e o arquivo `char_1.out` trata vários outros idiomas. O condutor do teste de regressão pega, automaticamente, o melhor arquivo para fazer a comparação quando verifica se foi bem-sucedido, e para computar as diferenças da falha (Isto significa que os testes de regressão não conseguem detectar se os resultados são apropriados para o idioma configurado. Os testes simplesmente pegam o arquivo de resultado que melhor se adequar).

Se por alguma razão os arquivos esperados existentes não incluírem algum idioma, é possível adicionar um novo arquivo. O esquema de nomes é `nomedoteste_dígito.out`. O dígito utilizado não tem importância. Lembre-se que o condutor do teste de regressão considera todos os arquivos deste tipo como sendo resultados de teste igualmente válidos. Se os resultados do teste forem específicos para alguma plataforma, em vez dessa abordagem deve ser utilizada a técnica descrita na [Seção 26.3](#).

2.3. Diferenças na data e hora

Umas poucas consultas do teste horology [5] falham se forem executadas no dia de início ou de fim do horário de verão, ou no dia seguinte aos mesmos. Estas consultas esperam que os intervalos entre a meia-noite do dia anterior, a meia noite do dia, e a meia-noite do dia seguinte, sejam de exatamente vinte e quatro horas — o que não acontece se o início ou o fim do horário de verão ocorrer neste intervalo.

Nota: Uma vez que são utilizadas as regras de horário de verão (*daylight-saving time*) dos EUA, este problema sempre ocorre no primeiro domingo de abril, no último domingo de outubro, e nas segundas-feiras seguintes, não importando quando o horário de verão começa ou termina onde se vive. Deve ser observado, também, que este problema aparece ou desaparece à meia-noite do Horário do Pacífico (UTC-7 ou UTC-8), e não à meia-noite do horário local. Portanto, a falha pode ocorrer no sábado ou durar até terça-feira dependendo de onde se vive.

A maior parte dos resultados de data e hora são dependentes da zona horária do ambiente. Os arquivos de referência são gerados para a zona horária PST8PDT (Berkeley, Califórnia), havendo falhas aparentes se os testes não forem executados com esta definição de zona horária. O condutor do teste de regressão define a variável de ambiente PGTZ como PST8PDT, o que normalmente garante resultados apropriados.

2.4. Diferenças no ponto flutuante

Alguns testes incluem cálculos com números de ponto flutuante de 64 bits (double precision) a partir de colunas das tabelas. Foram observadas diferenças nos resultados quando estão envolvidas funções matemáticas aplicadas a colunas do tipo double precision. Os testes float8 e geometry são particularmente propensos a apresentar pequenas diferenças entre plataformas, ou devido a opções diferentes de otimização do compilador. São necessárias comparações utilizando o olho humano para determinar se estas diferenças, que geralmente estão dez casas à direita do ponto decimal, são significativas.

Alguns sistemas mostram menos zero como -0, enquanto outros mostram simplesmente 0.

Alguns sistemas sinalizam erros das funções pow() e exp() de forma diferente da esperada pelo código corrente do PostgreSQL.

2.5. Diferenças na ordem das linhas

Podem ser encontradas diferenças onde as mesmas linhas são mostradas em uma ordem diferente da que aparece no arquivo de comparação. Na maior parte das vezes isto não é, a rigor, um erro. Os scripts de teste de regressão, em sua maioria, não são tão refinados a

ponto de utilizar a cláusula ORDER BY em todos os comandos SELECT e, portanto, a ordem das linhas do resultado não é bem definida, de acordo com o texto de especificação do padrão SQL. Na prática, uma vez que se está examinando as mesmas consultas sendo executadas nos mesmos dados pelo mesmo programa, geralmente são obtidos resultados na mesma ordem em todas as plataformas e, por isso, a ausência do ORDER BY não se torna um problema. Entretanto, algumas consultas apresentam diferenças na ordem das linhas entre plataformas (Diferenças na ordem das linhas também podem ser ocasionadas pela definição de um idioma diferente de C).

Portanto, se houver diferença na ordem das linhas isto não é algo com que devamos nos preocupar, a menos que a consulta possua uma cláusula ORDER BY que esteja sendo violada. Mas, por favor, informe de qualquer forma para que possamos adicionar a cláusula ORDER BY a esta consulta em particular e, com isso, eliminar a falsa "falha" nas próximas versões.

Deve-se estar querendo saber porque não se ordena explicitamente todas as consultas dos testes de regressão imediatamente, para acabar com este problema de uma vez e para sempre. A razão é que isto torna os testes de regressão menos úteis, e não mais úteis, porque vão tender a utilizar tipos de plano de consulta que produzem resultados ordenados, prejudicando os planos que não o fazem.

2.6. O teste "random"

O script do teste random tem por objetivo produzir resultados aleatórios. Raramente isto faz com que o teste de regressão random falhe. Ao ser digitado

```
diff results/random.out expected/random.out
```

deve ser produzida somente uma, ou umas poucas linhas diferentes. Não é necessário se preocupar com isto, a menos que o teste falhe repetidamente.

3. Arquivos de comparação específicos de plataformas

Como alguns testes produzem resultados inerentes a uma determinada plataforma, é disponibilizada uma maneira de fornecer arquivos de comparação de resultados específicos para a plataforma. Com freqüência a mesma discrepância se aplica a várias plataformas; em vez de fornecer arquivos de comparação distintos para todas as plataformas, existe um arquivo de mapeamento que define o arquivo de comparação a ser utilizado. Portanto, para eliminar falsas "falhas" nos testes para uma determinada plataforma, deve ser escolhido ou desenvolvido um arquivo de resultado alternativo, e depois adicionada uma linha no arquivo de mapeamento, que é o src/test/regress/resultmap.

Toda linha do arquivo de mapeamento possui a forma:

nome_do_teste/padrão_de_plataforma=nome_do_arquivo_de_comparação

O nome do teste é simplesmente o nome do módulo de teste de regressão específico. O padrão de plataforma é um padrão no estilo da ferramenta Unix expr (ou seja, uma expressão regular com uma âncora ^ implícita no início). Este padrão é comparado com o nome da plataforma conforme exibido por config.guess, seguido por :gcc ou :cc, dependendo se for utilizado o compilador GNU ou o compilador nativo do sistema (nos sistemas onde há diferença). O nome do arquivo de comparação é o nome do arquivo de comparação de resultado substituto.

Por exemplo: alguns sistemas interpretam valores de ponto flutuante muito pequenos como zero, em vez de informar um erro de *underflow*. Isto causa algumas pequenas diferenças no teste de regressão float8. Por isso é fornecido um arquivo de comparação alternativo, float8-small-is-zero.out, que inclui os resultados esperados nestes sistemas. Para silenciar as mensagens falsas de "falha" nas plataformas OpenBSD, o arquivo resultmap inclui

```
float8/i.86-.*-openbsd=float8-small-is-zero
```

que dispara em toda máquina para a qual a saída de config.guess corresponde a i.86-.*-openbsd. Outras linhas no arquivo resultmap selecionam arquivos de comparação alternativos para outras plataformas conforme apropriado.

46 - Backup lógico e físico do postgresql

Backup Lógico e restauração de bancos de dados

pg_dump – faz o backup de um banco de dados do PostgreSQL em um arquivo de script (texto puro) ou de outro tipo (tar ou outro).

pg_dump opções banco

São feitas cópias de segurança consistentes, mesmo que o banco de dados esteja sendo utilizado ao mesmo tempo. O pg_dump não bloqueia os outros usuários que estão acessando o banco de dados (leitura ou escrita).

As cópias de segurança podem ser feitas no formato de *script (p)*, *customizado (c)* ou *tar (t)*.

O formato personalizado, por padrão é compactado.

Para restaurar a partir de scripts em texto devemos usar o comando psql.

Para restaurar scripts feitos com formato personalizado (-Fc) e tar (-Ft) devemos usar o pg_restore.

Por default customizado é compactado.

Algumas Opções

-a (backup somente dos dados)

-b (incluir objetos grandes no backup)

-c (incluir comandos para remover – DROP, os objetos do banco antes de criá-lo)

-C (criar o banco e conectar ao mesmo)

-d (salvar os registros usando o comando INSERT ao invés do COPY). Reduz o desempenho. Somente se necessário (migração de SGBDs por exemplo).

-D (salvar os registros usando o comando INSERT ao invés do COPY explicitando os nomes das colunas).

-E codificação (passar uma codificação no backup)

-f arquivo (gerar backup para um arquivo)

-F formato (formato de saída. p – para texto plano, que é o padrão, c – para custom e t – para tar)

-i (ignorar a diferença de versão entre o pg_dump e o servidor)

-n esquema (backup apenas do esquema citado)

-N esquema (no backup NÃO incluir o esquema citado)

-s (backup somente a estrutura do banco, sem os dados)

-t tabela (salvar somente a tabela, sequência ou visão citada)

-T tabela (NÃO realizar o backup da referida tabela)

--disable-triggers (aplica-se somente quando salvando só os dados. Para desativar as triggers enquanto os dados são carregados).

-h host (host ou IP)

-p porta (porta)

-U usuário (usuário)

-W (forçar solicitação de senha)

Mais detalhes em:

<http://pgdocptbr.sourceforge.net/pg82/app-pgdump.html>

Observações

Se o agrupamento de bancos de dados tiver alguma adição local ao banco de dados template1, deve-se ter o cuidado de restaurar a saída do pg_dump em um banco de dados totalmente vazio; senão, poderão acontecer erros devido a definições duplicadas dos objetos adicionados. Para criar um banco de dados vazio, sem nenhuma adição local, deve-se fazê-lo a partir de template0, e não de template1 como, por exemplo:

```
CREATE DATABASE banco WITH TEMPLATE template0;
```

O backup no formato tar tem limitação para cada tabela, que devem ter no máximo 8GB no formato texto puro. O total dos objetos não tem limite de tamanho mas seus objetos sim.

É aconselhável executar o ANALYZE após restaurar de uma cópia de segurança para garantir um bom desempenho.

O pg_dump também pode ler bancos de dados de um PostgreSQL mais antigo, entretanto geralmente não consegue ler bancos de dados de um PostgreSQL mais recente.

Exemplos

Para salvar o banco de dados chamado meu_bd em um arquivo de script SQL:

```
pg_dump -U postgres meu_bd > bd.sql
```

Para restaurar este script no banco de dados (recém criado) chamado novo_bd:

```
dropdb novo_bd  
createdb novo_bd  
psql -U postgres -d novo_bd -f bd.sql
```

Para efetuar backup do banco de dados no script com formato customizado:

```
pg_dump -U postgres -Fc meu_bd > bd.dump
```

Para recarregar esta cópia de segurança no banco de dados (recém criado) chamado novo_bd:

```
pg_restore -U postgres -d novo_bd bd.dump
```

Para salvar uma única tabela chamada minha_tabela:

```
pg_dump -U postgres -t minha_tabela meu_bd > bd.sql
```

Para salvar todas as tabelas cujos nomes começam por emp no esquema detroit, exceto a tabela chamada empregado_log:

```
pg_dump -U postgres -t 'detroit.emp*' -T detroit.empregado_log meu_bd > bd.sql
```

Para salvar todos os esquemas cujos nomes começam por leste ou oeste e terminam por gsm, excluindo todos os esquemas cujos nomes contenham a palavra teste:

```
pg_dump -U postgres -n 'leste*gsm' -n 'oeste*gsm' -N '*teste*' meu_bd > bd.sql
```

A mesma coisa utilizando a notação de expressão regular para unificar as chaves:

```
pg_dump -U postgres -n '(leste|oeste)*gsm' -N '*teste*' meu_bd > bd.sql
```

Para salvar todos os objetos do banco de dados, exceto as tabelas cujos nomes começam por ts_ :

```
pg_dump -U postgres -T 'ts_*' meu_bd > bd.sql
```

Backup dos usuários e grupos e depois de todos os bancos:

```
pg_dumpall -g  
pg_dump -Fc para cada banco
```

<http://www.pgadmin.org/docs/1.6/backup.html>

pg_dumpall

Salva todos os bancos de um agrupamento de bancos de dados do PostgreSQL em um único arquivo de script.

pg_dumpall opções

O pg_dumpall também salva os objetos globais, comuns a todos os bancos de dados (O pg_dump não salva estes objetos). Atualmente são incluídas informações sobre os usuários do banco de dados e grupos, e permissões de acesso aplicadas aos bancos de dados como um todo.

Necessário conectar como um superusuário para poder gerar uma cópia de segurança completa. Também serão necessários privilégios de superusuário para executar o *script* produzido, para poder adicionar usuários e grupos, e para poder criar os bancos de dados.

O pg_dumpall precisa conectar várias vezes ao servidor PostgreSQL (uma vez para cada banco de dados). Se for utilizada autenticação por senha, provavelmente será solicitada a senha cada uma destas vezes. Neste caso é conveniente existir o arquivo `~/.pgpass` ou `pgpass.conf` (Windows).

Algumas Opções

-a (realizar backup somente dos dados)

-c (insere comandos para remover os bancos de dados – DROP antes dos comandos para criá-los)

-d (realiza o backup inserindo comandos INSERT ao invés do COPY). Fica lenta.

-D (backup usando INSERT ao invés de COPY e explicitando os nomes de campos)

-g (salva somente os objetos globais do SGBD, ou seja, usuários e grupos)

e várias outras semelhante ao comando pg_dump.

Mais opções em: <http://pgdocptbr.sourceforge.net/pg82/app-pg-dumpall.html>

Exemplos

```
pg_dumpall -U postgres > todos.sql
```

Restaurando:

```
psql -U postgres -f todos.sql banco
```

Aqui o banco pode ser qualquer um, pois o script gerado pelo pg_dumpall contém os comandos para a criação dos bancos e conexão aos mesmos, de acordo com o original.

Backup Full Compactado (Linux)

```
pg_dumpall -U postgres | gzip > full.gz
```

```
cat full.gz | gunzip | psql template1
```

Descompactar e fazer o restore em um só comando:

```
gunzip -c backup.tar.gz | pg_restore -d banco
```

ou

```
cat backup.tar.gz | gunzip | pg_restore -d banco
```

(o cat envia um stream do arquivo para o gunzip que passa para o pg_restore)

Backup remoto de um banco:

```
pg_dump -h hostremoto -d nomebanco | psql -h hostlocal -d banco
```

Backup em multivolumes (volumes de 200MB):

```
pg_dump nomebanco | split -m 200 nomearquivo
```

m para 1Mega, k para 1K, b para 512bytes

Importando backup de versão anterior do PostgreSQL

Instala-se a nova versão com porta diferente (ex.: 5433) e conectar ambos

```
pg_dumpall -p 5432 | psql -d template1 -p 5433
```

No Windows

```
pg_dump -f D:\MYDB_BCP -Fc -x -h localhost -U postgres MYDB
```

===== SCRIPT BAT =====

```
@echo off
```

```
rem (Nome do Usuário do banco para realizar o backup)
```

```
SET PGUSER=postgres
```

```
rem (Senha do usuário acima)
```

```
SET PGPASSWORD=1234
```

```
rem (Indo para a raiz do disco)
```

```
C:
```

```
rem (Selecionando a pasta onde será realizada o backup)
```

```

chdir C:\Sistemas\backup_postgresql
rem (banco.sql é o nome que definir para o meu backup
rem (Deletando o backup existente, só por precaução)
del banco.sql
echo "Aguarde, realizando o backup do Banco de Dados"
pg_dump -i -U postgres -b -o -f "C:\Sistemas\backup_postgresql\banco%Date%.sql"
banco
rem (sair da tela depois do backup)
exit

```

Por default o psql continua caso encontre um erro, mas podemos configurá-lo para parar caso encontre um:

```
\set ON_ERROR_STOP
```

Script de Backup no Windows

```
=====backup.bat=Formato: banco_dd-mm-aaaa_hh-mm=====
```

```

for /f "tokens=1,2,3,4 delims=/ " %%a in ('DATE /T') do set Data=%%b-%%c-%%d
for /f "tokens=1 delims=: " %%h in ('time /T') do set hour=%%h
for /f "tokens=2 delims=: " %%m in ('time /T') do set minutes=%%m
for /f "tokens=3 delims=: " %%a in ('time /T') do set ampm=%%a
set Hora=%hour%-%minutes%-%ampm%

```

```
pg_dump -U postgres controle_estoque -p 5222 -f "controle_estoque_%Data%_%Hora%.sql"
```

pg_restore - restaura um banco de dados do PostgreSQL a partir de um arquivo criado pelo pg_dump no formato custom (-Fc) ou tar (-Ft)

Importante: no Windows o pg_dumpall pedirá a senha para cada banco existente.

No pgpass.conf colocar * no campo bancodedados ou então faça uma cópia da linha para todos os bancos, inclusive tempalte0, template1 e postgres.
Ou não use pg_dumpall mas pg_dump apenas para os bancos desejados.

Na lista pgsql-general - <http://archives.postgresql.org/pgsql-general/2005-06/msg01279.php>

pg_restore

Objetivo - reconstruir o banco de dados no estado em que este se encontrava quando foi feito o backup. Para garantir a integridade dos dados, o banco a ser restaurado deve estar limpo, um banco criado recentemente e sem nenhum objeto ou registro.

-a (restaura somente os dados)

-c (exclui os objetos dos bancos antes de criar para restaurar)

-C (cria o banco antes de restaurar)

-d banco (restaura diretamente neste banco indicado). Caso esta opção não seja explicitada, a restauração será para o banco original do backup.

-e (encerra caso encontre erro. exit_on_error)

-f arquivo (restaura deste arquivo)

-i (ignorar diferença de versões entre o pg_restore e o servidor)

-L lista (restaura somente os objetos presentes no arquivo lista e na ordem)

-n esquema (restaura somente o esquema especificado)

-P função (restaura somente esta função)

-s (restaura somente o esquema do banco, não os dados)

-t tabela (restaura somente a definição e/ou dados da tabela)

-T gatilho (restaura somente este gatilho)

-W (força a solicitação da senha)

-1 (a restauração ocorre em uma única transação. Tudo ocorrerá bem ou nada será salvo)

-h host

-p porta

-U usuário

Mais opções em: <http://pgdocptbr.sourceforge.net/pg82/app-pgrestore.html>

Para criar um banco de dados vazio, sem nenhuma adição local, deve-se fazê-lo partir de template0, e não de template1 como, por exemplo:

```
CREATE DATABASE banco WITH TEMPLATE template0;
```

Uma vez restaurado, é aconselhável executar o comando ANALYZE em todas as tabelas restauradas para que o otimizador possua estatísticas úteis.

Exemplos

Banco de dados meu_bd em um arquivo de cópia de segurança no formato personalizado:

```
pg_dump -U postgres -Fc meu_bd > db.dump
```

Para remover o banco de dados e recriá-lo a partir da cópia de segurança:

```
dropdb -U postgres meu_bd
pg_restore -U postgres -C -d postgres db.dump
```

O banco de dados especificado na chave -d pode ser qualquer banco de dados existente no agrupamento; o pg_restore somente utiliza este banco de dados para emitir o comando CREATE DATABASE para meu_bd. Com a chave -C, os dados são sempre restaurados no banco de dados cujo nome aparece no arquivo de cópia de segurança.

Para recarregar a cópia de segurança em um banco de dados novo chamado bd_novo:

```
createdb -U postgres -T template0 bd_novo
pg_restore -U postgres -d bd_novo db.dump
```

Deve ser observado que não foi utilizada a chave -C, e sim conectado diretamente ao banco de dados a ser restaurado. Também deve ser observado que o novo banco de dados foi clonado a partir de template0 e não de template1, para garantir que esteja inicialmente vazio.

pg_dumpall has lot of disadvantages compared to pg_dump -Fc, I don't see the point why one would want that.

What I'd recommend is to use pg_dumpall -g and pg_dump -Fc on each DB. Then get a copy of rdiff-backup for windows to create nice incremental backups. Wrap those 3 things in a cmd script and use the task scheduler to run it in a given interval.

Mais detalhes em:

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

<http://pgdocptbr.sourceforge.net/pg80/backup.html>

5.4) Backup Físico offline (sistema de arquivos)

Uma estratégia alternativa para fazer cópia de segurança, é copiar diretamente os arquivos que o PostgreSQL usa para armazenar os dados dos bancos de dados.

Pode ser utilizada a forma preferida para fazer as cópias de segurança usuais dos arquivos do sistema como, por exemplo:

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

Entretanto, existem duas restrições que fazem com que este método seja impraticável ou, pelo menos, inferior ao pg_dump:

1. O servidor de banco de dados *deve* estar parado para que se possa obter uma cópia de segurança utilizável. Meias medidas, como impedir todas as conexões, não funcionam (principalmente porque o tar, e as ferramentas semelhantes, não capturam um instantâneo atômico do estado do sistema de arquivos em um determinado ponto no tempo). As informações sobre como parar o servidor podem ser encontradas na [Seção 16.6](#). É desnecessário dizer que também é necessário parar o servidor antes de restaurar os dados.
2. Caso tenha se aprofundado nos detalhes da organização do sistema de arquivos do banco de dados, poderá estar tentado a fazer cópias de segurança ou restauração de apenas algumas determinadas tabelas ou bancos de dados a partir de seus respectivos arquivos ou diretórios. Isto *não* funciona, porque as informações contidas nestes arquivos possuem apenas meia verdade. A outra metade está nos arquivos de registro de efetivação pg_clog/*, que contêm o status de efetivação de todas as transações. O arquivo da tabela somente possui utilidade com esta informação. É claro que também não é possível restaurar apenas uma tabela e seus dados associados em pg_clog, porque isto torna todas as outras tabelas do agrupamento de bancos de dados inúteis. Portanto, as cópias de segurança do sistema de arquivos somente funcionam para a restauração completa de todo o agrupamento de bancos de dados.

Deve ser observado que a cópia de segurança do sistema de arquivos não será necessariamente menor que a do Método SQL-dump. Ao contrário, é mais provável que seja maior; por exemplo, o pg_dump não necessita fazer cópia de segurança dos índices, mas apenas dos comandos para recriá-los.

Mais detalhes em:

<http://pgdocptbr.sourceforge.net/pg80/backup-file.html>

<http://www.postgresql.org/docs/8.3/interactive/backup-file.html>

Entendendo o WAL (Write Ahead Log) e o PITR no PostgreSQL

O *registro prévio da escrita* (WAL = *write ahead logging*) é uma abordagem padrão para registrar transações. A descrição detalhada pode ser encontrada na maioria (se não em todos) os livros sobre processamento de transação. Em poucas palavras, o conceito central do WAL é que as alterações nos arquivos de dados (onde as tabelas e os índices residem) devem ser escritas somente após estas alterações terem sido registradas, ou seja, quando os registros que descrevem as alterações tiverem sido descarregados em um meio de armazenamento permanente. Se este procedimento for seguido, não será necessário descarregar as páginas de dados no disco a cada efetivação de transação, porque se sabe que no evento de uma queda será possível recuperar o banco de dados utilizando o registro: todas as alterações que não foram aplicadas às páginas de dados são refeitas a partir dos registros (isto é a recuperação de rolar para a frente, *roll-forward*, também conhecida como *REDO*),

Benefícios do WAL

O primeiro grande benefício da utilização do WAL é a redução significativa do número de escritas em disco, uma vez que na hora em que a transação é efetivada somente precisa ser descarregado em disco o arquivo de registro, em vez de todos os arquivos de dados modificados pela transação. Em ambiente multiusuário, a efetivação de várias transações pode ser feita através de um único `fsync()` do arquivo de registro. Além disso, o arquivo de registro é escrito seqüencialmente e, portanto, o custo de sincronizar o registro é muito menor do que o custo de descarregar as páginas de dados. Isto é especialmente verdade em servidores tratando muitas transações pequenas afetando partes diferentes do armazenamento de dados.

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

1. linhas de índice apontando para linhas inexistentes da tabela
2. perda de linhas de índice nas operações de quebra de página (*split*)
3. conteúdo da página da tabela ou do índice totalmente danificado, por causa das páginas de dados parcialmente escritas

Os problemas com os índices (problemas 1 e 2) possivelmente poderiam ter sido resolvidos através de chamadas adicionais à função `fsync()`, mas não é óbvio como tratar o último caso sem o WAL; se for necessário, o WAL salva todo o conteúdo da página de dados no registro, para garantir a consistência da página na recuperação após a queda.

Por fim, o WAL permite que seja feita cópia de segurança em linha e recuperação para um ponto no tempo, conforme descrito na [Seção 22.3](#). Fazendo cópia dos arquivos de segmento do WAL pode-se retornar para qualquer instante no tempo coberto pelos

registros do WAL: simplesmente se instala uma versão anterior da cópia de segurança física do banco de dados, e se refaz o WAL até o ponto desejado no tempo. Além disso, a cópia de segurança física não precisa ser um instantâneo do estado do banco de dados — se a cópia for realizada durante um período de tempo, quando o WAL for refeito para este período de tempo da cópia serão corrigidas todas as inconsistências internas.

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

Internamente

O WAL é ativado automaticamente; não é requerida nenhuma ação por parte do administrador, exceto garantir que o espaço em disco adicional necessário para o WAL seja atendido, e que seja feito qualquer ajuste necessário (consulte a [Seção 25.2](#)).

O WAL é armazenado no diretório pg_xlog, sob o diretório de dados, como um conjunto de arquivos de segmento, normalmente com o tamanho de 16 MB cada. Cada segmento é dividido em páginas, normalmente de 8 kB cada. Os cabeçalhos dos registro estão descritos em access/xlog.h; o conteúdo do registro depende do tipo de evento que está sendo registrado. São atribuídos para nomes dos arquivos de segmento números que sempre aumentam, começando por 000000010000000000000000. Atualmente os números não recomeçam, mas deve demorar muito tempo até que seja exaurido o estoque de números disponíveis.

Os *buffers* do WAL e estruturas de controle ficam na memória compartilhada e são tratados pelos processos servidor filhos; são protegidos por bloqueios de peso leve. A demanda por memória compartilhada é dependente do número de *buffers*. O tamanho padrão dos *buffers* do WAL é 8 *buffers* de 8 kB cada um, ou um total de 64 kB.

É vantajoso o WAL ficar localizado em um disco diferente do que ficam os arquivos de banco de dados principais. Isto pode ser obtido movendo o diretório pg_xlog para outro local (enquanto o servidor estiver parado, é óbvio), e criando um vínculo simbólico do local original no diretório de dados principal para o novo local.

A finalidade do WAL, garantir que a alteração seja registrada antes que as linhas do banco de dados sejam alteradas, pode ser subvertida pelos controladores de disco (*drives*) que informam ao núcleo uma escrita bem-sucedida falsa, e na verdade apenas colocam os dados no *cache* sem armazenar no disco. Numa situação como esta a queda de energia pode conduzir a uma corrupção dos dados não recuperável. Os administradores devem tentar garantir que os discos que armazenam os arquivos de segmento do WAL do PostgreSQL não fazem estes falsos relatos.

Após um ponto de verificação ter sido feito e o registro descarregado, a posição do ponto de verificação é salva no arquivo pg_control. Portanto, quando uma recuperação vai ser feita o servidor lê primeiro pg_control, e depois o registro de ponto de verificação; em seguida realiza a operação de REDO varrendo para frente a partir da posição indicada pelo registro de ponto de verificação. Como, após o ponto de verificação, na primeira

modificação feita em uma página de dados é salvo todo o conteúdo desta página, todas as páginas modificadas desde o último ponto de verificação serão restauradas para um estado consistente.

Para tratar o caso em que o arquivo pg_control foi danificado, é necessário haver suporte para a possibilidade de varrer os arquivos de segmento do WAL em sentido contrário — mais novo para o mais antigo — para encontrar o último ponto de verificação. Isto ainda não foi implementado. O arquivo pg_control é pequeno o suficiente (menos que uma página de disco) para não estar sujeito a problemas de escrita parcial, e até o momento em que esta documentação foi escrita não haviam relatos de falhas do banco de dados devido unicamente a incapacidade de ler o arquivo pg_control. Portanto, embora este seja teoricamente um ponto fraco, na prática o arquivo pg_control não parece ser um problema.

PIRT

Esse documento é uma tradução de um capítulo de um livro no qual a fonte foi citada no final do documento. Vejo que muitas pessoas detêm dúvidas em relação a Point-in-time recovery, essa é minha forma de contribuição com a comunidade PostgreSQL. Desculpem pela falta de acentuação no documento (traduzi o documento ontem às 11/04/2006 01:30 AM, não me preocupei muito com a estética :P) por qualquer falha de tradução, acho que não são muitas e não põe em risco o bom entendimento.

Point-in-time recovery

Quando você faz uma mudança em algum banco de dados do PostgreSQL, PostgreSQL grava suas mudanças no shared-buffer pool, o write ahead log(WAL) e eventualmente no arquivo que você mudou. O WAL contém registros completos das mudanças que você fez. O mecanismo de Point-in-time recovery usa um histórico de modificações gravados nos arquivos WAL. Imagine o PITR como um esquema de backup incremental. Você começa com um backup completo e depois, arquiva as mudanças realizadas. Quando ocorrer um crash no seu servidor, você restaura o backup completo(full backup) e aplica as mudanças, em sequência, até que se recupere todos os dados que deseja recuperar.

Point-in-time recovery(PITR) pode parecer muito intimidador se você comece a ler a documentação. Para você ter uma boa visão sobre o sistema de PITR, iremos criar uma nova base, da um crash nela e recuperar os dados usando o mecanismo de PITR. Você pode seguir se quiser, mas você irá precisar de um espaço em disco extra.

Iremos começar criando um novo cluster.

```
$ export PGDATA=/usr/local/pgPITR
initdb
```

The files belonging to this database system will be owned by user "pg".
This user must also own the server process.

...
Success. You can now start the database server using:

```
postmaster -D /usr/local/pgPITR/data
or
pg_ctl -D /usr/local/pgPITR/data -l logfile start
```

Agora iremos mudar o arquivo \$PGDATA/postgresql.conf para acionar o mecanismo de PITR. A unica mudança que voce deve fazer e definir o parametro archive_command. O parametro archive_command diz ao Postgresql como os arquivos de WAL(write-ahead-log) irão ser gerados pelo servidor. Tomemos como exemplo a seguinte configuracao:

```
archive_command='cp %p /tmp/wals/%f'
```

Postgresql ira executar o archive_command ao inves de simplesmente deletar os arquivos WAL como ele normalmente faz. %p significa o caminho completo do WAL e o %f é o nome do arquivo.

Agora iremos criar o diretorio /tmp/wals, startar o processo postmaster e criar um banco de dados para que possamos trabalhar.

```
$mkdir /tmp/wals
$pg_ctl -l /tmp/pg.log start
$createdb teste
CREATE DATABASE
```

Nesse momento eu tenho um cluster completo , os dados relativos ao cluster estao em \$PGDATA, e um banco de dados chamado teste. Quando eu realizo mudancas no meu banco de dados teste, essas mudancas sao gravados nos arquivos WAL que estao em \$PGDATA/pg_xlog(como em qualquer outro cluster). Quando um arquivo WAL cresce, Postgresql ira copiar ele para o diretorio /tmp/wals para mante-los sao e salvos. A unica diferenca entre um cluster convencional e um cluster que esteja com o mecanismo de PITR acionado e que o servidor Postgresql ira arquivar os arquivos WAL ao inves de deleta-los.

Como o mecanismo de PITR trabalha com as mudancas efetuadas no banco de dados e gravadas nos arquivos WAL, iremos gerar alguns arquivos WAL criando algumas tabelas. Nao interessa o que dados irao conter as tabelas, nos queremos somente gerar os arquivos WAL, ate que eles crescam(cada arquivo WAL contem 16 megas).

```
$psql
```

```
Welcome to psql 8.0.0, the PostgreSQL interactive terminal. ...
teste=# BEGIN WORK; /* aqui comecamos a nossa transacao*/
teste=# create table teste1 as select * from pg_class,pg_attribute;
SELECT
teste=# COMMIT; executed at 12:30:00 pm /*terminamos a nossa transacao*/
teste=#\q
```

O command create table produz uma tabela que contem mais de 245000 registros e produz bastante arquivos WAL capas de chegar aos 16 megas cada um. Voce pode ver os segmentos WAL olhando o diretorio /tmp/wals

```
$ls /tmp/wals
```

```
000000010000000000000000
000000010000000000000001
000000010000000000000002
000000010000000000000003
000000010000000000000004
```

Agora iremos criar um backup completo de todo os meus dados do meu cluster. Para simplificar o exemplo irei criar um arquivo .tar.gz e salva-lo no diretorio /tmp. Antes de eu comecar o meu backup, iremos dizer ao Postgresql o que estamos prestes a fazer chamando a funcao pg_start_backup()

```
$psql teste
```

```
Welcome to psql 8.0.0, the PostgreSQL interactive terminal. ...
teste=# select pg_start_backup('full backup-Segunda');
pg_start_backup
-----
0/52EA2B8 (1 row)
teste=# \q
```

Agora iremos criar o backup dos nossos dados do cluster


```
0000000100000000000000000005
0000000100000000000000000005.002EA2B8.backup
0000000100000000000000000006
0000000100000000000000000007
0000000100000000000000000008
0000000100000000000000000009
000000010000000000000000000A
000000010000000000000000000B
000000010000000000000000000C
000000010000000000000000000D
000000010000000000000000000E
```

Nesse ponto acontece um desastre, a energia cai, um furacão inesperado, ou meu computador pega fogo(tudo acontece, arrasta, mensalão e tal menos o meu diretório /tmp é destruído) para simular um desastre iremos dar um kill no processo postmaster.

```
$KILL -9 (head -1 $PGDATA/postmaster.pid)
```

Agora é hora de recuperarmos todo o nosso cluster. Começaremos por renomear o nosso cluster detonado.
\$ mv \$PGDATA \$PGDATA.old

Agora iremos restaurar o backup da nossa tarball

```
$ cp /tmp/pgdata.tar.gz $PGDATA
$ tar -xvf pgdata.tar.gz
```

Agora temos o nosso \$PGDATA.old(que é o diretório dos arquivos do nosso cluster detonado) e o backup do nosso cluster antigo no \$PGDATA. Iremos dar uma limpada no cluster restaurado do backup antigo removendo arquivos WAL antigos e o arquivo postmaster.pid

```
$ rm -rf $PGDATA/pg_xlog/0*
$ rm -rf $PGDATA/postmaster.pid
```

Para garantir que posso recuperar a maior quantidade de dados possíveis irei copiar os arquivos WAL do cluster danificado dentro do diretório do cluster restaurado

```
$cp %PGDATA.old/pg_xlog/0* $PGDATA/pg_xlog/
```

Se os arquivos do cluster danificado não estiverem ao nosso alcance(podem ter queimado quando o computador pegou fogo), podemos recuperar todas as transações comitadas antes dos mais recentes arquivos WALs que foram criados. Em um banco de dados com muitas transações, perderíamos apenas alguns minutos, cerca de 16 megabytes.

Agora iremos começar o processo de recuperação , mas antes devemos nos lembrar do que aconteceu...

```
12:30:00pm criamos a tabela teste1 and commitamos as mudanças
2:38:00pm criamos a tabela teste2 and commitamos as mudanças
12:39:00pm criamos a tabela teste3 and commitamos as mudanças
12:40:00pm dropamos a tabela teste3 and commitamos as mudanças
```

Algum tempo entre 12:30 e 12:38 nos realizamos o backup de todo o banco. (lembre-se que nosso backup físico só continha a tabela teste1).

Quando iniciamos o processo de recuperação, podemos recuperar todas as mudanças ou podemos dizer ao Postgresql para parar em certo ponto do tempo(Point in time). Se a recuperação parar antes de 12:38, só teremos a tabela teste1, não teremos a tabela teste2 nem a teste3. Se a recuperação parar antes de 12:39, deveremos ter as tabelas teste1, teste3, mas não teremos a teste3. Se a recuperação parar antes de 12:40 teremos as 3 tabelas. Se deixarmos o Postgresql recuperar todas as mudanças a tabela teste3 deve desaparecer, porque dropamos ela.

Para controlarmos o processo de recuperação , criaremos um arquivo chamado \$PGDATA/recovery.conf que diz ao Postgresql como proceder. O arquivo recovery.conf se parece com isso:

```
$ cat $PGDATA/recovery.conf
```

```
restore_command= 'cp /tmp/wals/%f %p'
recovery_target_time='2005-06-22 12:39:01 EST'
```

O parametro restore_command diz ao postgresql como recuperar os arquivos WAL que estao armazenados em /tmp/wals. O parametro recovery_target_time diz ao Postgreql quando parar. Se voce quer recuperar todas as mudancas simplesmente omita esse parametro. Como dizemos ao postgresql para parar em 12:39 irmos encontrar as tabelas teste1,teste2,teste3.

Postgresql comeca o processo de recuperacao logo que o processo postmaster e startado(postmaster sabe que tem algo a fazer porque ele acha o arquivo \$PGDATA/recovery.conf

```
$pg_ctl -l /tmp/pg.log start
postmaster started
```

Se voce estiver rodando linux/Unix voce pode observar o arquivo de log do servidor

```
$ tail -f /tmp/pg.log
LOG: starting archive recovery
LOG: restore_command = "cp /tmp/wals/%f %p"
LOG: recovery_target_time = 2005-06-22 13:05:00-05
...
LOG: restored log file "000000010000000000000006" from archive
LOG: restored log file "000000010000000000000007" from archive
LOG: restored log file "000000010000000000000008" from archive
LOG: restored log file "000000010000000000000009" from archive
...
LOG: archive recovery complete
LOG: database system is ready
```

Quando o processo de recuperacao se completa, Postgresql renomeia o arquivo recovery.conf para recovery.done, para evitar que quando o processo postmaster seja iniciado novamente ocorra de novo a restauracao.

Agora posso me conectar ao meu servidor e ver as tabelas teste1,teste2,teste3.

```
$psql teste
```

```
Welcome to psql 8.0.0, the PostgreSQL interactive terminal. ...
test=# \d
```

Como voce pode ver PITR e facil de configurar(somente definir o archive_command no postgresql.conf).

Recapitulando todo o processo de configuracao:

- 1- configure o PITR definindo o archive_command no postgresql.conf
- 2- Restart o postmaster para habilitar o processamento do arquivo WAL
- 3- conecte em uma database e chame a funcao pg_start_backup(rotulo)
- 4- Faça o backup do cluster(Voce nao precisa parar o servidor)

Recapitulando o processo de recuperacao

- 1- Se possivel renomeie o cluster danificado para que possa copiar a maior quantidade de arquivos wal possiveis
- 2- Descompacte o cluster do arquivo que foi criado o backup
- 3- No diretorio do cluster restaurado remova os seguintes arquivos \$PG_DATA/pg_xlog/0* e o \$PGDATA/postmaster.pid
- 4- Se possivel copie os arquivos WAL do cluster danificado para o cluster restaurado para garantir que as transacoes mais recentes sejam tambem recuperadas
- 5- Crie o arquivo recovery.conf, com no minimo o seguinte parametro restore_command
- 6- Start o postmaster
- 7- Verifique se os dados que vc esperava se encontram no lugar

Fonte: PostgreSQL: The comprehensive guide to building, programming, and administering PostgreSQL databases, Second Edition By Korry Douglas, Susan Douglas

Traducao: Joao Cosme de Oliveira Junior : joaocosme@pop.com.br

Cópia de segurança em-linha (PITR)

Durante todo o tempo, o PostgreSQL mantém o *registro de escrita prévia* (WAL = *write ahead log*) no subdiretório pg_xlog do diretório de dados do agrupamento. O WAL contém todas as alterações realizadas nos arquivos de dados do banco de dados. O WAL existe, principalmente, com a finalidade de fornecer segurança contra quedas: se o sistema cair, o banco de dados pode retornar a um estado consistente "refazendo" as entradas gravadas desde o último ponto de verificação. Entretanto, a existência do WAL torna possível uma terceira estratégia para fazer cópia de segurança de banco de dados: pode ser combinada a cópia de segurança do banco de dados no nível de sistema de arquivos, com cópia dos arquivos de segmento do WAL. Se for necessário fazer a recuperação, pode ser feita a recuperação da cópia de segurança do banco de dados no nível de sistema de arquivos e, depois, refeitas as alterações a partir da cópia dos arquivos de segmento do WAL, para trazer a restauração para o tempo presente. A administração desta abordagem é mais complexa que a administração das abordagens anteriores, mas existem alguns benefícios significativos:

- O ponto de partida não precisa ser uma cópia de segurança totalmente consistente. Toda inconsistência interna na cópia de segurança é corrigida quando o WAL é refeito (o que não é muito diferente do que acontece durante a recuperação de uma queda). Portanto, não é necessário um sistema operacional com capacidade de tirar instantâneos, basta apenas o tar, ou outra ferramenta semelhante.
- Como pode ser reunida uma seqüência indefinidamente longa de arquivos de segmento do WAL para serem refeitos, pode ser obtida uma cópia de segurança contínua simplesmente continuando a fazer cópias dos arquivos de segmento do WAL. Isto é particularmente útil para bancos de dados grandes, onde pode não ser conveniente fazer cópias de segurança completas regularmente.
- Não existe nada que diga que as entradas do WAL devem ser refeitas até o fim. Pode-se parar de refazer em qualquer ponto, e obter um instantâneo consistente do banco de dados como se tivesse sido tirado no instante da parada. Portanto, esta técnica suporta a *recuperação para um determinado ponto no tempo*: é possível restaurar voltando o banco de dados para o estado em que se encontrava a qualquer instante posterior ao da realização da cópia de segurança base.
- Se outra máquina, carregada com a mesma cópia de segurança base do banco de dados, for alimentada continuamente com a série de arquivos de segmento do WAL, será criado um sistema reserva à quente (*hot standby*): a qualquer instante

esta outra máquina pode ser ativada com uma cópia quase atual do banco de dados.

Da mesma forma que o método de cópia de segurança no nível de sistema de arquivos simples, este método suporta apenas a restauração de todo o agrupamento de bancos de dados, e não a restauração de apenas um subconjunto deste. Requer, também, grande volume de armazenamento de arquivos: a cópia de segurança base pode ser grande, e um sistema carregado gera vários megabytes de tráfego para o WAL que precisam ser guardados. Ainda assim, é o método de cópia de segurança preferido para muitas situações onde é necessária uma alta confiabilidade.

Para fazer uma recuperação bem-sucedida utilizando cópia de segurança em-linha, é necessária uma seqüência contínua de arquivos de segmento do WAL guardados, que venha desde, pelo menos, o instante em que foi feita a cópia de segurança base do banco de dados. Para começar, deve ser configurado e testado o procedimento para fazer cópia dos arquivos de segmento do WAL, *antes* de ser feita a cópia de segurança base do banco de dados. Assim sendo, primeiro será explicada a mecânica para fazer cópia dos arquivos de segmento do WAL.

Cópia dos arquivos de segmento do WAL

Em um sentido abstrato, a execução do sistema PostgreSQL produz uma seqüência indefinidamente longa de entradas no WAL. O sistema divide fisicamente esta seqüência em *arquivos de segmento* do WAL, normalmente com 16 MB cada (embora o tamanho possa ser alterado durante a construção do PostgreSQL). São atribuídos nomes numéricos aos arquivos de segmento para refletir sua posição na seqüência abstrata do WAL. Quando não é feita cópia dos arquivos de segmento do WAL, normalmente o sistema cria apenas uns poucos arquivos de segmento e, depois, "recicla-os" renomeando os arquivos que não são mais de interesse com número de segmento mais alto. Assume-se não existir mais interesse em um arquivo de segmento cujo conteúdo preceda o ponto de verificação anterior ao último, podendo, portanto, ser reciclado.

Quando é feita a cópia dos arquivos de segmento do WAL, deseja-se capturar o conteúdo de cada arquivo quando este é completado, guardando os dados em algum lugar antes do arquivo de segmento ser reciclado para ser reutilizado. Dependendo da aplicação e dos periféricos disponíveis, podem haver muitas maneiras de "guardar os dados em algum lugar": os arquivos de segmento podem ser copiados para outra máquina usando um diretório NFS montado, podem ser escritos em uma unidade de fita (havendo garantia que os arquivos poderão ser restaurados com seus nomes originais), podem ser agrupados e gravados em CD, ou de alguma outra forma. Para que o administrador de banco de dados tenha a máxima flexibilidade possível, o PostgreSQL tenta não assumir nada sobre como as cópias serão feitas. Em vez disso, o PostgreSQL deixa o administrador escolher o comando a ser executado para copiar o arquivo de segmento completado para o local de destino. O comando pode ser tão simples como cp, ou pode envolver um script complexo para o interpretador de comandos — tudo depende do administrador.

O comando a ser executado é especificado através do parâmetro de configuração [archive_command](#), que na prática é sempre colocado no arquivo `postgresql.conf`. Na cadeia de caracteres do comando, todo `%p` é substituído pelo caminho absoluto do arquivo a ser copiado, enquanto todo `%f` é substituído pelo nome do arquivo apenas. Se for necessário incorporar o caractere `%` ao comando, deve ser escrito `%%`. A forma mais simples de um comando útil é algo como

```
archive_command = 'cp -i %p /mnt/servidor/dir_copias/%f </dev/null'
```

que irá copiar os arquivos de segmento do WAL, prontos para serem copiados, para o diretório `/mnt/servidor/dir_copias` (Isto é um exemplo, e não uma recomendação, e pode não funcionar em todas as plataformas).

O comando para realizar a cópia é executado sob a propriedade do mesmo usuário que está executando o servidor PostgreSQL. Como a série de arquivos do WAL contém efetivamente tudo que está no banco de dados, deve haver certeza que a cópia está protegida contra olhos curiosos; por exemplo, colocando a cópia em diretório sem acesso para grupo ou para todos.

É importante que o comando para realizar a cópia retorne o status de saída zero se, e somente se, for bem-sucedido. Ao receber o resultado zero, o PostgreSQL assume que a cópia do arquivo de segmento do WAL foi bem-sucedida, e remove ou recicla o arquivo de segmento. Entretanto, um status diferente de zero informa ao PostgreSQL que o arquivo não foi copiado; serão feitas tentativas periódicas até ser bem-sucedida.

Geralmente o comando de cópia deve ser projetado de tal forma que não sobrescreva algum arquivo de cópia pré-existente. Esta é uma característica de segurança importante para preservar a integridade da cópia no caso de um erro do administrador (tal como enviar a saída de dois servidores diferentes para o mesmo diretório de cópias).

Aconselha-se a testar o comando de cópia proposto para ter certeza que não sobrescreve um arquivo existente, e que retorna um status diferente de zero neste caso. Tem sido observado que `cp -i` faz isto corretamente em algumas plataformas, mas não em outras. Se o comando escolhido não tratar este caso corretamente por conta própria, deve ser adicionado um comando para testar a existência do arquivo de cópia. Por exemplo, algo como

```
archive_command = 'test ! -f .../%f && cp %p .../%f'
```

funciona corretamente na maioria das variantes do Unix.

Ao projetar a configuração de cópia deve ser considerado o que vai acontecer quando o comando de cópia falhar repetidas vezes, seja porque alguma funcionalidade requer intervenção do operador, ou porque não há espaço para armazenar a cópia. Esta situação pode ocorrer, por exemplo, quando a cópia é escrita em fita e não há um sistema automático para troca de fitas: quando a fita ficar cheia, não será possível fazer outras cópias enquanto a fita não for trocada. Deve-se garantir que qualquer condição de erro,

ou solicitação feita a um operador humano, seja relatada de forma apropriada para que a situação possa ser resolvida o mais rápido possível. Enquanto a situação não for resolvida, continuarão sendo criados novos arquivos de segmento do WAL no diretório pg_xlog.

A velocidade do comando de cópia não é importante, desde que possa acompanhar a taxa média de geração de dados para o WAL. A operação normal prossegue mesmo que o processo de cópia fique um pouco atrasado. Se o processo de cópia ficar muito atrasado, vai aumentar a quantidade de dados perdidos caso ocorra um desastre. Significa, também, que o diretório pg_xlog vai conter um número grande de arquivos de segmento que ainda não foram copiados, podendo, inclusive, exceder o espaço livre em disco. Aconselha-se que o processo de cópia seja monitorado para garantir que esteja funcionando da forma planejada.

Havendo preocupação em se poder recuperar até o presente instante, devem ser efetuados passos adicionais para garantir que o arquivo de segmento do WAL corrente, parcialmente preenchido, também seja copiado para algum lugar. Isto é particularmente importante no caso do servidor gerar pouco tráfego para o WAL (ou tiver períodos ociosos onde isto acontece), uma vez que pode levar muito tempo até que o arquivo de segmento fique totalmente preenchido e pronto para ser copiado. Uma forma possível de tratar esta situação é definir uma entrada no cron [1] que periodicamente, talvez uma vez por minuto, identifique o arquivo de segmento do WAL corrente e o guarde em algum lugar seguro. Então, a combinação dos arquivos de segmento do WAL guardados, com o arquivo de segmento do WAL corrente guardado, será suficiente para garantir que o banco de dados pode ser restaurado até um minuto, ou menos, antes do presente instante. Atualmente este comportamento não está presente no PostgreSQL, porque não se deseja complicar a definição de [archive_command](#) requerendo que este acompanhe cópias bem-sucedidas, mas diferentes, do mesmo arquivo do WAL. O [archive_command](#) é chamado apenas para segmentos do WAL completados. Exceto no caso de novas tentativas devido a falha, só é chamado uma vez para um determinado nome de arquivo.

Ao escrever o comando de cópia, deve ser assumido que os nomes dos arquivos a serem copiados podem ter comprimento de até 64 caracteres, e que podem conter qualquer combinação de letras ASCII, dígitos e pontos. Não é necessário recordar o caminho original completo (%p), mas é necessário recordar o nome do arquivo (%f).

Deve ser lembrado que embora a cópia do WAL permita restaurar toda modificação feita nos dados dos bancos de dados do PostgreSQL, não restaura as alterações feitas nos arquivos de configuração (ou seja, nos arquivos postgresql.conf, pg_hba.conf e pg_ident.conf), uma vez que estes arquivos são editados manualmente, em vez de através de operações SQL. Aconselha-se a manter os arquivos de configuração em um local onde são feitas cópias de segurança regulares do sistema de arquivos. Para mudar os arquivos de configuração de lugar, deve ser consultada a [Seção 16.4.1](#).

Criação da cópia de segurança base

O procedimento para fazer a cópia de segurança base é relativamente simples:

1. Garantir que a cópia dos arquivos de segmento do WAL esteja ativada e funcionando.
2. Conectar ao banco de dados como um superusuário e executar o comando

```
SELECT pg_start_backup('rótulo');
```

onde rótulo é qualquer cadeia de caracteres que se deseje usar para identificar unicamente esta operação de cópia de segurança (Uma boa prática é utilizar o caminho completo de onde se deseja colocar o arquivo de cópia de segurança). A função pg_start_backup cria o arquivo *rótulo da cópia de segurança*, chamado backup_label, com informações sobre a cópia de segurança, no diretório do agrupamento.

Para executar este comando, não importa qual banco de dados do agrupamento é usado para fazer a conexão. O resultado retornado pela função pode ser ignorado; mas se for relatado um erro, este deve ser tratado antes de prosseguir.

3. Realizar a cópia de segurança utilizando qualquer ferramenta conveniente para cópia de segurança do sistema de arquivos, como tar ou cpio. Não é necessário, nem desejado, parar a operação normal do banco de dados enquanto a cópia é feita.
4. Conectar novamente ao banco de dados como um superusuário e executar o comando:

```
SELECT pg_stop_backup();
```

Se a execução for bem sucedida, está terminado.

Não é necessário ficar muito preocupado com o tempo decorrido entre a execução de pg_start_backup e o início da realização da cópia de segurança, nem entre o fim da realização da cópia de segurança e a execução de pg_stop_backup; uns poucos minutos de atraso não vão criar nenhum problema. Entretanto, deve haver certeza que as operações são realizadas seqüencialmente, sem que haja sobreposição.

Deve haver certeza que a cópia de segurança inclui todos os arquivos sob o diretório do agrupamento de bancos de dados (por exemplo, /usr/local/pgsql/data). Se estiverem sendo utilizados espaços de tabelas que não residem sob este diretório, deve-se ter o cuidado de inclui-los também (e ter certeza que a cópia de segurança guarda vínculos simbólicos como vínculos, senão a restauração vai danificar os espaços de tabelas).

Entretanto, podem ser omitidos da cópia de segurança os arquivos sob o subdiretório pg_xlog do diretório do agrupamento. Esta pequena complicação a mais vale a pena ser feita porque reduz o risco de erros na restauração. É fácil de ser feito se pg_xlog for um

vínculo simbólico apontando para algum lugar fora do agrupamento, o que é uma configuração comum por razões de desempenho.

Para poder utilizar esta cópia de segurança base, devem ser mantidas por perto todas as cópias dos arquivos de segmento do WAL gerados no momento ou após o início da mesma. Para ajudar a realizar esta tarefa, a função `pg_stop_backup` cria o *arquivo de histórico de cópia de segurança*, que é armazenado imediatamente na área de cópia do WAL. Este arquivo recebe um nome derivado do primeiro arquivo de segmento do WAL, que é necessário possuir para fazer uso da cópia de segurança. Por exemplo, se o arquivo do WAL tiver o nome `0000000100001234000055CD`, o arquivo de histórico de cópia de segurança vai ter um nome parecido com `0000000100001234000055CD.007C9330.backup` (A segunda parte do nome do arquivo representa a posição exata dentro do arquivo do WAL, podendo normalmente ser ignorada). Uma vez que o arquivo contendo a cópia de segurança base tenha sido guardado em local seguro, podem ser apagados todos os arquivos de segmento do WAL com nomes numericamente precedentes a este número. O arquivo de histórico de cópia de segurança é apenas um pequeno arquivo texto. Contém a cadeia de caracteres rótulo fornecida à função `pg_start_backup`, assim como as horas de início e fim da cópia de segurança. Se o rótulo for utilizado para identificar onde está armazenada a cópia de segurança base do banco de dados, então basta o arquivo de histórico de cópia de segurança para se saber qual é o arquivo de cópia de segurança a ser restaurado, no caso disto precisar ser feito.

Uma vez que é necessário manter por perto todos os arquivos de segmento do WAL copiados desde a última cópia de segurança base, o intervalo entre estas cópias de segurança geralmente deve ser escolhido tendo por base quanto armazenamento se deseja consumir para os arquivos do WAL guardados. Também deve ser considerado quanto tempo se está preparado para despendar com a restauração, no caso de ser necessário fazer uma restauração — o sistema terá que refazer todos os segmentos do WAL, o que pode ser muito demorado se tiver sido decorrido muito tempo desde a última cópia de segurança base.

Também vale a pena notar que a função `pg_start_backup` cria no diretório do agrupamento de bancos de dados um arquivo chamado `backup_label`, que depois é removido pela função `pg_stop_backup`. Este arquivo fica guardado como parte do arquivo de cópia de segurança base. O arquivo rótulo de cópia de segurança inclui a cadeia de caracteres rótulo fornecida para a função `pg_start_backup`, assim como a hora em que `pg_start_backup` foi executada, e o nome do arquivo de segmento inicial do WAL. Em caso de dúvida, é possível olhar dentro do arquivo de cópia de segurança base e determinar com exatidão de qual sessão de cópia de segurança este arquivo provém.

Também é possível fazer a cópia de segurança base enquanto o postmaster está parado. Neste caso, obviamente não podem ser utilizadas as funções `pg_start_backup` e `pg_stop_backup`, sendo responsabilidade do administrador controlar a que cópia de

segurança cada arquivo pertence, e até quanto tempo atrás os arquivos de segmento do WAL associados vão. Geralmente é melhor seguir os procedimentos para cópia de segurança mostrados acima.

Recuperação a partir de cópia de segurança em-linha

Certo, aconteceu o pior e é necessário recuperar a partir da cópia de segurança. O procedimento está mostrado abaixo:

1. Parar o postmaster, se estiver executando.
2. Havendo espaço para isso, copiar todo o diretório de dados do agrupamento, e todos os espaços de tabelas, para um lugar temporário, para o caso de necessidade. Deve ser observado que esta medida de precaução requer a existência de espaço no sistema suficiente para manter duas cópias do banco de dados existente. Se não houver espaço suficiente, é necessário pelo menos uma cópia do conteúdo do subdiretório pg_xlog do diretório de dados do agrupamento, porque pode conter arquivos de segmento do WAL que não foram copiados quando o sistema parou.
3. Apagar todos os arquivos e subdiretórios existentes sob o diretório de dados do agrupamento, e sob os diretórios raiz dos espaços de tabelas em uso.
4. Restaurar os arquivos do banco de dados a partir da cópia de segurança base. Deve-se tomar cuidado para que sejam restaurados com o dono correto (o usuário do sistema de banco de dados, e não o usuário root), e com as permissões corretas. Se estiverem sendo utilizados espaços de tabelas, deve ser verificado se foram restaurados corretamente os vínculos simbólicos no subdiretório pg_tblspc/.
5. Remover todos os arquivos presentes no subdiretório pg_xlog; porque estes vêm da cópia de segurança base e, portanto, provavelmente estão obsoletos. Se o subdiretório pg_xlog não fizer parte da cópia de segurança base, então este subdiretório deve ser criado, assim como o subdiretório pg_xlog/archive_status.
6. Se existirem arquivos de segmento do WAL que não foram copiados para o diretório de cópias, mas que foram salvos no passo 2, estes devem ser copiados para o diretório pg_xlog; é melhor copiá-los em vez de movê-los, para que ainda existam arquivos não modificados caso ocorra algum problema e o processo tenha de ser recomeçado.
7. Criar o arquivo de comando de recuperação recovery.conf no diretório de dados do agrupamento (consulte [Recovery Settings](#)). Também pode ser útil modificar temporariamente o arquivo pg_hba.conf, para impedir que os usuários

comuns se conectem até que se tenha certeza que a recuperação foi bem-sucedida.

8. Iniciar o postmaster. O postmaster vai entrar no modo de recuperação e prosseguir lendo os arquivos do WAL necessários. Após o término do processo de recuperação, o postmaster muda o nome do arquivo `recovery.conf` para `recovery.done` (para impedir que entre novamente no modo de recuperação no caso de uma queda posterior), e depois começa as operações normais de banco de dados.
9. Deve ser feita a inspeção do conteúdo do banco de dados para garantir que a recuperação foi feita até onde deveria ser feita. Caso contrário, deve-se retornar ao passo 1. Se tudo correu bem, liberar o acesso aos usuários retornando `pg_hba.conf` à sua condição normal.

A parte chave de todo este procedimento é a definição do arquivo contendo o comando de recuperação, que descreve como se deseja fazer a recuperação, e até onde a recuperação deve ir. Pode ser utilizado o arquivo `recovery.conf.sample` (geralmente presente no diretório de instalação `share`) na forma de um protótipo [2]. O único parâmetro requerido no arquivo `recovery.conf` é `restore_command`, que informa ao PostgreSQL como trazer de volta os arquivos de segmento do WAL copiados. Como no `archive_command`, este parâmetro é uma cadeia de caracteres para o interpretador de comandos. Pode conter `%f`, que é substituído pelo nome do arquivo do WAL a ser trazido de volta, e `%p`, que é substituído pelo caminho absoluto para onde o arquivo do WAL será copiado. Se for necessário incorporar o caractere `%` ao comando, deve ser escrito `%%`. A forma mais simples de um comando útil é algo como

```
restore_command = 'cp /mnt/servidor/dir_copias/%f %p'
```

que irá copiar os arquivos de segmento do WAL previamente guardados a partir do diretório `/mnt/servidor/dir_copias`. É claro que pode ser utilizado algo muito mais complicado, talvez um script que solicite ao operador a montagem da fita apropriada.

É importante que o comando retorne um status de saída diferente de zero em caso de falha. Será solicitado ao comando os arquivos do WAL cujos nomes não estejam presente entre as cópias; deve retornar um status diferente de zero quando for feita a solicitação. Esta não é uma condição de erro. Deve-se tomar cuidado para que o nome base do caminho `%p` seja diferente de `%f`; não deve ser esperado que sejam intercambiáveis.

Os arquivos de segmento do WAL que não puderem ser encontrados entre as cópias, serão procurados no diretório `pg_xlog/`; isto permite que os arquivos de segmento recentes, ainda não copiados, sejam utilizados. Entretanto, os arquivos de segmento que estiverem entre as cópias terão preferência sobre os arquivos em `pg_xlog`. O sistema não sobrescreve os arquivos presentes em `pg_xlog` quando busca os arquivos guardados.

Normalmente a recuperação prossegue através de todos os arquivos de segmento do WAL, portanto restaurando o banco de dados até o presente momento (ou tão próximo quanto se pode chegar utilizando os segmentos do WAL). Mas se for necessário recuperar até algum ponto anterior no tempo (digamos, logo antes do DBA júnior ter apagado a tabela principal de transação de alguém), deve-se simplesmente especificar no arquivo `recovery.conf` o ponto de parada requerido. O ponto de parada, conhecido como "destino da recuperação", pode ser especificado tanto pela data e hora quanto pelo término de um ID de transação específico. Até o momento em que este manual foi escrito, somente podia ser utilizada a opção data e hora, pela falta de ferramenta para ajudar a descobrir com precisão o identificador de transação a ser utilizado.

Nota: O ponto de parada deve estar situado após o momento de término da cópia de segurança base (o momento em que foi executada a função `pg_stop_backup`). A cópia de segurança não pode ser utilizada para recuperar até um momento em que a cópia de segurança base estava em andamento (Para recuperar até este ponto, deve-se retornar para uma cópia de segurança base anterior e refazer a partir desta cópia).

Definições de recuperação

Estas definições somente podem ser feitas no arquivo `recovery.conf`, e são aplicadas apenas durante a recuperação. As definições deverão ser definidas novamente nas próximas recuperações que se desejar realizar. As definições não podem ser alteradas após a recuperação ter começado.

restore_command (string)

O comando, para o interpretador de comandos, a ser executado para trazer de volta os segmentos da série de arquivos do WAL guardados. Este parâmetro é requerido. Todo `%f` presente na cadeia de caracteres é substituído pelo nome do arquivo a ser trazido de volta das cópias, e todo `%p` é substituído pelo caminho absoluto para onde o arquivo será copiado no servidor. Se for necessário incorporar o caractere `%` ao comando, deve ser escrito `%%`.

É importante que o comando retorne o status de saída zero se, e somente se, for bem-sucedido. Será solicitado ao comando os arquivos cujos nomes não estejam presentes entre as cópias; deve retornar um status diferente de zero quando for feita a solicitação. Exemplos:

```
restore_command = 'cp /mnt/servidor/dir_copias/%f "%p"'
restore_command = 'copy /mnt/servidor/dir_copias/%f "%p"' # Windows
```

recovery_target_time (timestamp)

Este parâmetro especifica o carimbo do tempo até onde a recuperação deve prosseguir. Somente pode ser especificado um entre `recovery_target_time` e `recovery_target_xid`. O padrão é recuperar até o fim do WAL. O ponto de parada preciso também é influenciado por `recovery_target_inclusive`.

recovery_target_xid (string)

Este parâmetro especifica o identificador de transação até onde a recuperação deve prosseguir. Deve-se ter em mente que enquanto os identificadores são atribuídos seqüencialmente no início da transação, as transações podem ficar completas em uma ordem numérica diferente. As transações que serão recuperadas são aquelas que foram efetivadas antes (e opcionalmente incluindo) a transação especificada. Somente pode ser especificado um entre `recovery_target_xid` e `recovery_target_time`. O padrão é recuperar até o fim do WAL. O ponto de parada preciso também é influenciado por `recovery_target_inclusive`.

`recovery_target_inclusive` (boolean)

Especifica se a parada deve acontecer logo após o destino de recuperação especificado (true), ou logo antes do destino de recuperação especificado (false). Aplica-se tanto a `recovery_target_time` quanto a `recovery_target_xid`, o que for especificado para esta recuperação. Indica se as transações que possuem exatamente a hora de efetivação ou o identificador de destino, respectivamente, serão incluídas na recuperação. O valor padrão é true.

`recovery_target_timeline` (string)

Especifica a recuperação de uma determinada cronologia. O padrão é recuperar ao longo da cronologia que era a cronologia corrente quando foi feita a cópia de segurança base. Somente será necessário definir este parâmetro em situações de re-recuperações complexas, onde é necessário retornar para um estado a que se chegou após uma recuperação para um ponto no tempo. Consulte a explicação na [Seção 22.3.4](#).

Cronologias

A capacidade de restaurar um banco de dados para um determinado ponto anterior no tempo cria algumas complexidades que são semelhantes às das narrativas de ficção científica sobre viagem no tempo e universos paralelos. Por exemplo, na história original do banco de dados talvez tenha sido removida uma tabela importante às 5:15 da tarde de terça-feira. Imperturbável, o administrador pega a cópia de segurança e faz uma restauração para o ponto no tempo 5:14 da tarde de terça feira, e o sistema volta a funcionar. Nesta história do universo do banco de dados, a tabela nunca foi removida. Mas suponha que mais tarde seja descoberto que esta não foi uma boa idéia, e que se deseja voltar para algum ponto posterior da história original. Mas isto não poderá ser feito, porque quando o banco de dados foi posto em atividade este sobrescreveu alguns dos arquivos de segmento do WAL que levariam ao ponto no tempo onde agora quer se chegar. Portanto, realmente é necessário fazer distinção entre a série de entradas no WAL geradas após a recuperação para um ponto no tempo, e àquelas geradas durante a história original do banco de dados.

Para lidar com estes problemas, o PostgreSQL possui a noção de *cronologias (timelines)*. Cada vez que é feita uma recuperação no tempo anterior ao fim da seqüência do WAL, é criada uma nova cronologia para identificar a série de registros do WAL geradas após a

recuperação (entretanto, se a recuperação prosseguir até o final do WAL, não é criada uma nova cronologia: apenas se estende a cronologia existente). O número identificador da cronologia é parte dos nomes dos arquivos de segmento do WAL e, portanto, uma nova cronologia não sobrescreve os dados do WAL gerados pelas cronologias anteriores. É possível, na verdade, guardar muitas cronologias diferentes. Embora possa parecer uma funcionalidade sem utilidade, muitas vezes é de grande valia. Considere a situação onde não se tem certeza absoluta de até que ponto no tempo deve ser feita a recuperação e, portanto, devem ser feitas muitas tentativas de recuperação até ser encontrado o melhor lugar para se desviar da história antiga. Sem as cronologias este processo em pouco tempo cria uma confusão impossível de ser gerenciada. Com as cronologias, pode ser feita a recuperação até *qualquer* estado anterior, inclusive os estados no desvio de cronologia abandonados posteriormente.

Toda vez que é criada uma nova cronologia, o PostgreSQL cria um arquivo de "história da cronologia" que mostra de que cronologia foi feito o desvio, e quando. Os arquivos de cronologia são necessários para permitir o sistema buscar os arquivos de segmento do WAL corretos ao fazer a recuperação a partir de uma área de cópias que contém várias cronologias. Portanto, estes arquivos são guardados na área de cópias como qualquer arquivo de segmento do WAL. Os arquivos de cronologia são apenas pequenos arquivos texto sendo, portanto, barato e apropriado mantê-los guardados indefinidamente (ao contrário dos arquivos de segmento que são grandes). É possível, caso se deseje fazê-lo, adicionar comentários aos arquivos de cronologia para fazer anotações personalizadas sobre como e porque foi criada uma determinada cronologia. Estes comentários são muito úteis quando há um grande número de cronologias criadas como resultado de experiências.

O comportamento padrão de recuperação é recuperar ao longo da cronologia que era a cronologia corrente quando foi feita a cópia de segurança base. Se for desejado fazer a recuperação utilizando uma cronologia filha (ou seja, deseja-se retornar para algum estado que foi gerado após uma tentativa de recuperação), é necessário especificar o identificador da cronologia de destino no arquivo `recovery.conf`. Não é possível fazer a recuperação para um estado que foi um desvio anterior à cópia de segurança base.

Cuidado

No momento em que esta documentação foi escrita, haviam várias limitações para o método de cópia de segurança em-linha, que provavelmente serão corrigidas nas versões futuras:

- Atualmente não são gravadas no WAL as operações em índices que não são B-tree (índices hash, R-tree e GiST), portanto quando o WAL é refeito os índices destes tipos não são atualizados. A prática recomendada para contornar este problema é reindexar manualmente todos os índices destes tipos após o término da operação de recuperação.

Também deve ser notado que o formato atual do WAL é muito volumoso, uma vez que inclui muitos instantâneos de páginas de disco. Isto é apropriado para a finalidade de recuperação de quedas, uma vez que pode ser necessário corrigir páginas de disco parcialmente preenchidas. Entretanto, não é necessário armazenar tantas cópias de páginas para operações de recuperação para um determinado ponto no tempo. Uma área para desenvolvimento futuro é a compressão dos dados do WAL copiados, pela remoção das cópias de página desnecessárias.

Detalhes em:

<http://www.postgresql.org/docs/8.3/interactive/continuous-archiving.html>

<https://www.postgresql.org/docs/9.6/static/continuous-archiving.html>

Sintaxe dos comandos para backup

pg_dumpall

-h - host

-p - porta

Sintaxe completa

```
pg_dump -U user_name -h localhost -d dbname -n schemaname -t table_name --data-only  
--column-inserts db_name > data_dump.sql
```

Usando psql para dumps em texto plano

pg_restore para dumps em tar e comprimidos

-a - somente dados

-C - recriar banco ao restaurar

-d - banco

-f - nome do arquivo

-F{c|t}

-h - host

-p - porta

-t - restaurar somente uma tabela

-T - restaurar somente uma trigger

pg_dump

-a - somente dados

-C - recriar os bancos de dados no restore

-d - incluir INSERTs no script

-F{c|t|p} - c(gzip), t(tar), p(plain text)

-O - sem o user dono. No restore será o dono quem restaura

-s - somente esquema/estrutura, sem dados

-t - backup somente de tabela

-Z - compressão de 0 a 9

Exemplos de uso

`pg_dump banco > banco.sql`

Carregar um script em um novo banco

`psql -d novobanco -f novobanco.sql`

Gerar dump em formato customizado

`pg_dump -Fc mydb > mydb.dump`

Gerar dump de apenas uma tabela

`pg_dump -t mytab mydb > mytab.sql`

Gerar um dump de todas as tabelas começadas com 'emp' do esquema detroit, exceto a tabela chamada employee_log:

`pg_dump -t 'detroit.emp*' -T detroit.employee_log mydb > db.sql`

Gerar dump de todos os esquemas começados com east ou west e terminados com gsm, excluindo qualquer esquema cujo nome contém test:

`pg_dump -n 'east*gsm' -n 'west*gsm' -N '*test*' mydb > db.sql`

Dump de todos os objetos do banco, exceto as tabelas começadas com ts_:

`pg_dump -T 'ts_*' mydb > db.sql`

Restartar

`/etc/init.d/postgresql restart`

Acessar com:

`psql -h IP -U usuario -d banco`

ou (acessar banco default, postgres)

`psql -h IP -U usuario`

Listar bancos remotamente

`psql -l -h 192.168.1.12 -U postgres`

Ajuda

`psql --help`

Backup local e restore remoto

```
pg_dump banco | psql -h hostname banco -U postgres
```

```
pg_restore apoena -f
```

Usando o PostgreSQL

Remoto

```
pg_dump -h host1 dbname | psql -h host2 dbname
```

Local

```
sudo su - postgres
```

```
pg_dump postgres > postgres_db.bak
```

Remoto

```
pg_dump -h remote_host -p remote_port name_of_database > name_of_backup_file
```

Diferente usuário

```
pg_dump -U user_name -h remote_host -p remote_port name_of_database > name_of_backup_file
```

Só esquema:

Bypassar tabela: --exclude-table=table

```
pg_dump -U ribamar_sousa -h 10.0.0.60 -p 5432 apoena -Fc -n patrimonio --column-inserts > apoena_patrimonio.sql
```

```
pg_dump -U ribamar_sousa apoena -Fc -n patrimonio --column-inserts --exclude-table=tabela1 --exclude-table=tabela2 > apoena_patrimonio.sql
```

```
pg_dump ...other...options... -Fc -n B >dump.dmp
```

Restore o dump file:

```
pg_restore -d somedb dump.dmp
```

```
pg_restore -d apoena apoena_veiculos.dmp
```

Backup plain text sql com inserts somente uma tabela somente os dados

```
pg_dump --table=veiculos --data-only --column-inserts apoena -n veiculos > veiculos.sql
```

For a data-only export use COPY.

You get a file with one table row per line as plain text (not INSERT commands), it's smaller and faster:

```
COPY (SELECT * FROM nyummy.cimory WHERE city = 'tokio') TO '/path/to/file.csv';
```

Import the same to another table of the same structure anywhere with:

```
COPY other_tbl FROM '/path/to/file.csv';
```

Backup e Restore de Dados

Comandos de Backup

pg_dumpall - backup em modo texto de todos os bancos do cluster

pg_dump -

psql

Os dois primeiros são mais consistentes

Backup de todos os bancos

```
pg_dumpall -h localhost -p 5432 -U postgres -v -f "/backup/dbserver.sql"
```

Restore num PostgreSQL limpo

su postgres

```
psql -f /backup/dbserver.sql
```

O pg_dump faz backup tipo texto e tipo binário para um maior controle da restauração.

Backup de todos os bancos de dados do tipo texto.

A ideia por traz do dump é criar um arquivo com comandos SQL capaz de recriar o estado do banco quando restaurada.

```
pg_dumpall > cluster.sql
```

Restore do dump acima:

```
psql -f cluster.sql postgres
```

Backup de apenas um banco no formato texto:

```
pg_dump --inserts intranet > intranet.sql
```

Restaurando:

```
psql intranet < intranet.sql
```

Trabalhando com grandes bancos de dados:

```
pg_dump banco | gzip > banco.gz
```

Restaurando:

```
gunzip -c banco.gz | psql banco
```

ou

```
cat banco.gz | gunzip | psql banco
```

Usando split, que permite quebrar um arquivo em pedaços. Exemplo para 1MB:

```
pg_dump banco | split -b 1m - script
```

Restaurar

```
cat script* | psql banco
```

Backups Customizados

Gerar backup com formatos customizados. Estes formatos de backup oferecem a facilidade de permitir que o restore aconteça de forma seletiva.

Criar um backup usando o formato customizado:

```
pg_dump -Fc banco > banco.dump
```

Restore

```
pg_restore -d banco banco.dump
```

Para grandes bancos use o formato paralelo (-j) para compactar e para restaurar
com o pg_restore.

Backup no formato tar:

```
pg_dump -Ft banco > banco.tar
```

Restore

```
pg_restore -d banco banco.dump.tar
```

Para que o backup contenha os comandos SQL devemos compactar com:

```
pg_dump -Ft -c banco > banco.tar
```

Efetuar backup de apenas um único esquema de um banco

```
pg_dump -n dp_k1 intranet > dp_k1.sql
```

Restore

```
psql intranet < dp_k1.sql
```

Backup de uma única tabela

```
pg_dump -n dp_k1 -t lotes intranet > dp_k1_lotes.sql
```

Restore

```
psql intranet < dp_k1_lotes.sql
```

Efetuar o backup de um banco em um servidor e recriar o banco em outro servidor

```
pg_dump -h hostlocal bancolocal | psql -h hostremoto bancoremoto
```

Restore

Usar somente em caso de desastre ou quando se confia que o backup está realmente melhor que o atual.

Usar preferentemente um novo banco para restaurações.

O psql é usado para restaurar backups de arquivos texto.

Após a restauração de backup executar o

analyze

para atualizar as estatísticas e melhorar a performance do novo banco.

Restore para backup customizado/binário:

```
pg_restore opções arquivo.tar
```

PostgreSQL - Backup e Restore

Gerar backup de um banco:

```
pg_dump -U postgres nomebanco > nomebanco.sql
```

Restaurar o backup do banco num banco recem-criado:

```
psql -U postgres -d nomebanco -f novobanco.sql
```

Restaurar backup de um servidor para outro (remoto):

```
pg_dump -U postgres -h host1 nomebanco | psql -h host2 nomebanco
```

Backup de todos os bancos do servidor:

```
pg_dumpall -U postgres > todos.sql
```

Restaurando todos os bancos no banco postgres (ou template1):

```
psql -U postgres -f todos.sql postgres
```

Manipulando grandes bancos de dados:

```
pg_dump -U postgres banco | gzip > banco.gz
```

Restaurar com:

```
gunzip -c banco.gz | psql banco
```

ou:

```
cat banco.gz | gunzip | psql banco
```

Usando split:pg_dump -Fc mydb > db.dump

```
pg_dump -U postgres banco | split -b 1m - banco.sql
```

Restaurar com:

```
cat banco* | psql banco
```

Usando formatos personalizados:

```
pg_dump -U postgres -Fc banco > banco.sql
```

Restaurando:

```
pg_restore -U postgres -d banco banco.sql
```

Usando o pg_restore:

Quando usamos formatos personalizados: Fc ou Ft então podemos restaurar com pg_restore

Assumindo que o dump foi feito usando um formato personalizado:

```
pg_dump -U postgres -Fc banco > banco.dump
```

```
dropdb -U postgres banco
```

```
pg_restore -U postgres -C -d postgres banco.dump -- Criar o banco e então restaurar o dump
```

Restaurando num novo e limpo banco:

```
createdb -U postgres -T template0 novobanco
```

```
pg_restore -U postgres -d novobanco banco.dump
```

```
pg_dump -U postgres -Fc banco > banco.sql
```

```
pg_dump -U postgres -Ft banco > banco.tar
```

```
pg_restore -U postgres -d novobanco banco.tar
```

Gerar o backup de uma única tabela:

```
pg_dump -U postgres -t tabela banco > banco.sql
```

Gerar um backup de todas as tabelas que começam com emp do esquema pessoal, exceto a tabela empregados_log:

```
pg_dump -U postgres -t 'pessoal.emp*' -T pessoal.empregados_log banco > banco.sql
```

Gerar backup de todos os objetos dos bancos de dados exceto tabelas com nomes iniciados com ar_:

```
pg_dump -U postgres -T 'ar_*' banco > banco.sql
```

Restaurar apenas a estrutura de um banco, sem os dados:

```
pg_restore -U postgres -s -d novobanco banco.sql
```

Restaurar somente os dados de um banco sem a estrutura:

```
pg_restore -U postgres -a -d novobanco banco.sql
```

Restaurar somente um esquema de um banco:

```
pg_restore -U postgres -d novobanco -n nomeesquema banco.sql
```

Restaurar somente uma tabela de um banco:

```
pg_restore -U postgres -d novobanco -t nometabela banco.sql
```

Criar banco limpo, tendo como base o template0:

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

Referências:

<http://www.postgresql.org/docs/8.3/interactive/app-pgdump.html>

<http://www.postgresql.org/docs/current/static/backup.html>

<http://www.postgresql.org/docs/8.3/interactive/app-pgrestore.html>

Script para backup do PostgreSQL

```
#!/bin/sh

#/usr/local/bin/pgback.sh

# Adaptado de - http://www.sertoriopen.com.br/?p=55
# Documentação: https://www.postgresql.org/docs/9.6/static/app-pgdump.html
# Uso: pgback.sh nomebanco nomeesquema

if [ "$1" = "-h" ] || [ "$1" = "--help" ] || [ -z "$1" ]
then
echo "Sintaxe correta:\n\npgback.sh banco [esquema] [tabela]"
exit 1
fi

DATA=`/bin/date +%d-%m-%Y`

# diretório de backup
DIR="/backup/pg_backup/"

if [ ! -d "$DIR" ]
then
mkdir -p "$DIR"
fi

ARQUIVO="$DIR$1-$2-$DATA.sql"
#echo $ARQUIVO;

# variáveis
HOST="localhost"
USER="postgres"
export PGPASSWORD="postgres"

# backup
# --no-owner = sem owner, --inserts = com inserts, -Fp = customizado com plain text,
# Customizado permite restore de apenas uma tabela, ou um esquema
# $1 - nome do banco, $2 - nome do esquema, -t = tabela
if [ ! -z $3 ]
then
pg_dump --no-owner --inserts -Fp -t $3 $1 $2 > $ARQUIVO
else
pg_dump --no-owner --inserts -Fp $1 $2 > $ARQUIVO
fi
```

then

```
/usr/bin/psql -U $USER -c "alter user postgres set search_path to \"$2"
/usr/bin/pg_dump -h $HOST -U $USER -n $2 -t $3 --no-owner --inserts -Fp $1 -f $ARQUIVO
gzip -9 $ARQUIVO
exit 1

elif [ ! -z $2 ]
then
/usr/bin/psql -U $USER -c "alter user postgres set search_path to \"$2"
/usr/bin/pg_dump -h $HOST -U $USER -n $2 --no-owner --inserts -Fp $1 -f $ARQUIVO
gzip -9 $ARQUIVO
exit 1

else
/usr/bin/pg_dump -h $HOST -U $USER --no-owner --inserts -Fp $1 -f $ARQUIVO
gzip -9 $ARQUIVO
fi
```

Usando o Comando Copy

Exportar dados de uma tabela para a saída padrão

Exportar dados de uma tabela para um arquivo

Copia de uma tabela para a saída padrão:

\c intranet

set search_path to dp_k1

copy lotes to stdout;

Exportar a tabela lotes para o arquivo lotes.sql:

copy lotes to '/home/ribafs/lotes.sql';

Importar dados do arquivo lotes.sql para a tabela lotes:

copy lotes from '/home/ribafs/lotes.sql';

Exportando do PostgreSQL para HTML

<https://www.vivaolinux.com.br/dicas/impressora.php?codigo=1193>

Migração de um banco de dados no 8.3 para o PostgreSQL 9.5

Não será apenas uma migração de um banco num servidor para outro, estaremos também corrigindo seus erros/deficiências de forma a ter no novo servidor um banco íntegro.

Começaremos apenas com duas tabelas do esquema patrimonio, que deveriam estar relacionadas: material e plano_contas_geral

Alguns dos problemas encontrados:

- Ausência de chave primária (contas_grupos)
- Ausência de chave estrangeira entre materiais e contas

Por conta dos problemas havia

- contas duplicadas,
- grupo de material que não existia em contas
- e o grupo era armazenado duplicado nas duas tabelas

Material mudará para materiais para maior compatibilidade com o framework CakePHP.

Conectar do desktop ao servidor de testes de bancos de dados em 10.0.0.60 usando
ribamar_sousa

Senha do AD

Exportar material e plano_contas_geral e guardar o .sql

As tabelas originalmente estavam assim:

```
DROP TABLE IF EXISTS "plano_contas_geral";
CREATE TABLE "patrimonio"."plano_contas_geral" (
    "pc_tipo" character varying(1),
    "pc_grupo" character varying(4),
    "pc_descricao" character varying(50)
) WITH (oids = true);
```

```
INSERT INTO "plano_contas_geral" ("pc_tipo", "pc_grupo", "pc_descricao") VALUES
```

```

CREATE TABLE "patrimonio"."material_" (
    "mat_codigo" character varying(6) NOT NULL,
    "mat_plano_de_contas" character varying(4) NOT NULL,
    "mat_descricao" character varying(54) NOT NULL,
    "uid_inclusao" character varying(10),
    "uid_data_inclusao" timestamp,
    "uid.Alteracao" character varying(10),
    "uid_data.Alteracao" timestamp,
    CONSTRAINT "material_pkey" PRIMARY KEY ("mat_codigo")
) WITH (oids = true);

```

INSERT INTO "material_" ("mat_codigo", "mat_plano_de_contas", "mat_descricao", "uid_inclusao", "uid_data_inclusao", "uid.Alteracao", "uid_data.Alteracao") VALUES

Fazer uma cópia do sql original para materiais_contas_desktop.sql

Mover a tabela plano_contas_geral e seus registros para o início do arquivo

Renomear a tabela plano_contas_gerais para contas e adicionar um campo chave primária e renomear a tabela e os campos para que fique assim:

```

CREATE TABLE sc_patrimonio.contas (
    id serial primary key,
    tipo character varying(1),
    grupo character varying(4),
    descricao character varying(50)
) WITH (oids = true);

```

INSERT INTO contas (tipo, grupo, descricao) VALUES

Agora importar a tabela contas e seus registros para o banco db_intranet

Remover Duplicados

Agora vamos remover os registros duplicados de contas

Veja quantos registros existem:

select count(id) from contas - 130

```

DELETE FROM
    contas a
USING contas b
WHERE
    a.id < b.id
    AND a.grupo = b.grupo

```

Agora veja novamente

```
select count(id) from contas - 62
```

Alterando tipo de dados de varchar par integer em materiais

Editar a tabele material no script, mudando seu nome para materiais, e inserindo uma chave estrangeira que a ligará à tabela contas, ficando assim

```

CREATE TABLE sc_patrimonio.materiais (
    codigo character varying(6) unique NOT NULL,
    conta_id varchar(4) NOT NULL,
    descricao character varying(54) NOT NULL,
    uid_inclusao" character varying(10),
    uid_data_inclusao" timestamp,
    uid_alteracao" character varying(10),
    uid_data_alteracao" timestamp
) WITH (oids = true);

```

- Importar a tabela materiais com todos os registros do script para o banco bd_patrimonio em 10.0.0.60

Alterar o campo conta_id de varchar para integer em materiais e contas

```

ALTER TABLE materiais
    ALTER COLUMN conta_id TYPE integer
    USING conta_id::integer;

```

Adicionar uma chave estrangeira ligando materiais a contas

constraint grupo_fk foreign key (plano_conta_grupo) references plano_contas_gerais(grupo) ON
DELETE RESTRICT ON UPDATE CASCADE

Podemos facilmente fazer isso através do adminer:

- Selecionar a tabela materiais
- Adicionar chave estrangeira
- Tabela de destino – contas
- Origem – conta_id
- Destino – id
- On delete – restrict
- On update – cascade

Salvar

Ao clicar em salvar ele nos informa que o registro com grupo 5112 não está presente na tabela "contas". Então o removemos da tabela materiais ou o adicionamos na contas.

Após remover consegui criar o relacionamento.

Update em materiais recebendo em contas_id o valor de grupo de contas

UPDATE materiais m

SET conta_id = c.id

FROM contas c

WHERE c.grupo = m.conta_id

As tabelas finalmente ficarão assim:

```
CREATE TABLE "sc_patrimonio"."contas" (
    "id" serial primary key,
    "grupo" integer NOT NULL,
    "tipo" character varying(1),
    "descricao" character varying(50),
    CONSTRAINT "contas_grupo_key" UNIQUE ("grupo"),
) WITH (oids = true);
```

INSERT INTO contas (grupo, tipo, descricao) VALUES

CREATE TABLE "sc_patrimonio"."materiais" (

```

"id" serial primary key,
"codigo" character varying(6) NOT NULL,
"conta_id" integer NOT NULL,
"descricao" character varying(54) NOT NULL,
"uid_inclusao" character varying(10),
"uid_data_inclusao" timestamp,
"uid_alteracao" character varying(10),
"uid_data_alteracao" timestamp,
"status" boolean,
CONSTRAINT "materiais_codigo_key" UNIQUE ("codigo"),
constraint grupo_fk foreign key (conta_id) references contas(id) on delete restrict on update
cascade
) WITH (oids = true);

INSERT INTO materiais (codigo, contas_id, descricao, uid_inclusao, uid_data_inclusao,
uid_alteracao, uid_data_alteracao, status) VALUES

```

Agora já podemos criar o aplicativo usando o CakePHP 3 para estas duas tabelas.

IMPORTAR SCRIPTS TIPO CSV E SQL PARA O POSTGRESQL

1 - Importar Banco de CEPs (do Brasil) de arquivo tipo CSV para o PostgreSQL

Remover a chave primária da tabela cep do banco cadastro_clientes.

- Baixar CEP do site - <http://ribafs.byethost2.com/> :

1 - http://ribafs.byethost2.com/index.php?option=com_repository&Itemid=26&func=fileinfo&id=43

2 - http://ribafs.byethost2.com/index.php?option=com_repository&Itemid=26&func=fileinfo&id=44

Descompactar o primeiro com o winrar ou fillzip.

Renomear o arquivo descompactado para cep.csv e copiar para o c:\

Abrir o psql com:

psql -U postgres -W

Após fornecer a senha acesse o banco `cadastro_clientes` com:

```
\c cadastro_clientes
```

Então importe o arquivo com:

```
\copy cep from c:\cep.csv e aguarde.
```

2 - Importar arquivos .sql para o PostgreSQG

Abrir o `psql` com:

```
psql -U postgres -W
```

Após fornecer a senha acesse o banco `cadastro_clientes` (por exemplo) com:

```
\c cadastro_clientes
```

Então importe o arquivo com:

```
\i c:\nomearquivo.sql e aguarde
```

47 - Configurações do PostgreSQL

Configurar o PostgreSQL para que um banco de dados seja acessado por determinado servidor é uma tarefa que exige conhecimento das regras do SGBD. Isso torna o PostgreSQL mais trabalhoso e também mais seguro, pois não é uma tarefa trivial.

O arquivo pg_hba.conf

A autenticação do cliente é controlada pelo arquivo que por tradição se chama `pg_hba.conf` e é armazenado no diretório de dados do agrupamento de bancos de dados. HBA significa “autenticação baseada no hospedeiro” (*host-based authentication*). É instalado um arquivo `pg_hba.conf` padrão quando o diretório de dados é inicializado pelo utilitário `initdb`. Entretanto, é possível colocar o arquivo de configuração da autenticação em outro local; consulte o parâmetro de configuração [hba_file](#).

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

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

O registro pode ter um dos sete formatos a seguir:

```
local      banco_de_dados  usuário  método_de_autenticação [opção_de_autenticação]
host       banco_de_dados  usuário  endereço_de_CIDR   método_de_autenticação [opção_de_autenticação]
hostssl    banco_de_dados  usuário  endereço_de_CIDR   método_de_autenticação [opção_de_autenticação]
hostnossal banco_de_dados  usuário  endereço_de_CIDR   método_de_autenticação [opção_de_autenticação]
host       banco_de_dados  usuário  endereço_de_IP     máscara_de_IP   método_de_autenticação [opção_de_autenticação]
hostssl    banco_de_dados  usuário  endereço_de_IP     máscara_de_IP   método_de_autenticação [opção_de_autenticação]
hostnossal banco_de_dados  usuário  endereço_de_IP     máscara_de_IP   método_de_autenticação [opção_de_autenticação]
```

O significado de cada campo está descrito abaixo:

local

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

host

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

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

hostssl

Este registro corresponde às tentativas de conexão feitas utilizando o protocolo TCP/IP, mas somente quando a conexão é feita com a criptografia SSL.

Para esta opção poder ser utilizada o servidor deve ter sido construído com o suporte a SSL ativado. Além disso, o SSL deve ser ativado na inicialização do servidor através do parâmetro de configuração [ssl](#) (Para obter informações adicionais deve ser consultada a [Seção 16.7](#)).

hostnossal

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

banco_de_dados

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

usuário

Especifica quais usuários do PostgreSQL este registro corresponde. O valor all especifica que corresponde a todos os usuários. Senão, é o nome de um usuário

específico do PostgreSQL. Podem ser fornecidos vários nomes de usuário separados por vírgula. Podem ser especificados nomes de grupo precedendo o nome do grupo por +. Pode ser especificado um arquivo contendo nomes de usuário precedendo o nome do arquivo por @.

endereço_de_CIDR

Especifica a faixa de endereços de IP da máquina cliente que este registro corresponde. Contém um endereço de IP na notação padrão decimal com pontos, e o comprimento da máscara de CIDR (Os endereços de IP somente podem ser especificados numericamente, e não como domínios ou nomes de hospedeiro). O comprimento da máscara indica o número de bits de mais alta ordem que o endereço de IP do cliente deve corresponder. Os bits à direita devem ser zero em um determinado endereço de IP. Não pode haver espaços em branco entre os endereços de IP, a / e o comprimento da máscara de CIDR. [1]

Um endereço_de_CIDR típico seria 172.20.143.89/32 para um único hospedeiro, ou 172.20.143.0/24 para uma rede. Para especificar um único hospedeiro deve ser utilizada uma máscara de CIDR igual a 32 para o IPv4, ou igual a 128 para o IPv6.

Um endereço especificado no formato IPv4 corresponde às conexões IPv6 que possuem o endereço correspondente como, por exemplo, 127.0.0.1 corresponde ao endereço de IPv6 ::ffff:127.0.0.1. Uma entrada especificada no formato IPv6 corresponde apenas às conexões IPv6, mesmo que represente um endereço na faixa IPv4-em-IPv6. Deve ser observado que as entradas no formato IPv6 serão rejeitadas se a biblioteca C do sistema não possuir suporte a endereços IPv6.

Este campo se aplica apenas aos registros host, hostssl e hostnossal.

endereço_de_IP máscara_de_IP

Estes campos podem ser utilizados como uma alternativa à notação endereço_de_CIDR. Em vez de especificar o comprimento da máscara, a máscara é especificada como uma coluna em separado. Por exemplo, 255.0.0.0 representa uma máscara de CIDR para endereços de IPv4 com comprimento igual a 8, e 255.255.255.255 representa uma máscara de CIDR com comprimento igual a 32.

Estes campos se aplicam apenas aos registros host, hostssl e hostnossal.

método_de_autenticação

Especifica o método de autenticação a ser utilizado para se conectar através deste registro. Abaixo está mostrado um resumo das escolhas possíveis; os detalhes podem ser encontrados na [Seção 19.2](#).

trust

A conexão é permitida incondicionalmente. Este método permite a qualquer um que possa se conectar ao servidor de banco de dados PostgreSQL se

autenticar como o usuário do PostgreSQL que for desejado, sem necessidade de senha. Consulte a [Seção 19.2.1](#) para obter detalhes.

reject

A conexão é rejeitada incondicionalmente. É útil para "eliminar por filtragem" certos hospedeiros de um grupo.

md5

Requer que o cliente forneça uma senha criptografada pelo método md5 para autenticação. Consulte a [Seção 19.2.2](#) para obter detalhes.

crypt

Requer que o cliente forneça uma senha criptografada através de crypt() para autenticação. Deve-se dar preferência ao método md5 para os clientes com versão 7.2 ou posterior, mas os clientes com versão anterior a 7.2 somente suportam crypt. Consulte a [Seção 19.2.2](#) para obter detalhes.

password

Requer que o cliente forneça uma senha não criptografada para autenticação. Uma vez que a senha é enviada em texto puro pela rede, não deve ser utilizado em redes não confiáveis. Consulte a [Seção 19.2.2](#) para obter detalhes.

krb4

É utilizado Kerberos V4 para autenticar o usuário. Somente disponível para conexões TCP/IP. Consulte a [Seção 19.2.3](#) para obter detalhes.

krb5

É utilizado Kerberos V5 para autenticar o usuário. Somente disponível para conexões TCP/IP. Consulte a [Seção 19.2.3](#) para obter detalhes.

ident

Obtém o nome de usuário do sistema operacional do cliente (para conexões TCP/IP fazendo contato com o servidor de identificação no cliente, para conexões locais obtendo a partir do sistema operacional) e verifica se o usuário possui permissão para se conectar como o usuário de banco de dados solicitado consultando o mapa especificado após a palavra chave ident. Consulte a [Seção 19.2.4](#) para obter detalhes.

pam

Autenticação utilizando o serviço *Pluggable Authentication Modules* (PAM) fornecido pelo sistema operacional. Para obter detalhes deve ser consultada a [Seção 19.2.5](#).

opção_de_autenticação

O significado deste campo opcional depende do método de autenticação escolhido, estando descrito na próxima seção.

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

Uma vez que os registros de pg_hba.conf são examinados seqüencialmente a cada tentativa de conexão, a ordem dos registros possui significado. Normalmente, os primeiros registros possuem parâmetros de correspondência de conexão mais exigentes e métodos de autenticação menos exigentes, enquanto os últimos registros possuem parâmetros de correspondência menos exigentes e métodos de autenticação mais exigentes. Por exemplo, pode-se desejar utilizar a autenticação trust para conexões TCP/IP locais, mas requerer o uso de senha para conexões TCP/IP remotas. Neste caso, o registro especificando a autenticação trust para conexões a partir de 127.0.0.1 deve aparecer antes do registro especificando autenticação por senha para uma faixa mais ampla de endereços de IP de cliente permitidos.

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

Exemplos do pg_hba.conf

Exemplo de registros do arquivo pg_hba.conf

```
# Permitir qualquer usuário do sistema local se conectar a qualquer banco
# de dados sob qualquer nome de usuário utilizando os soquetes do domínio
# Unix (o padrão para conexões locais).
#
# TYPE    DATABASE        USER            CIDR-ADDRESS            METHOD
local    all             all             trust
#
# A mesma coisa utilizando conexões locais TCP/IP retornantes (loopback).
#
# TYPE    DATABASE        USER            CIDR-ADDRESS            METHOD
host     all             all             127.0.0.1/32          trust
#
# O mesmo que o exemplo anterior mas utilizando uma coluna em separado para
# máscara de rede.
#
```

```

# TYPE  DATABASE  USER        IP-ADDRESS      IP-MASK          METHOD
host   all       all        127.0.0.1     255.255.255.255 trust

# Permitir qualquer usuário de qualquer hospedeiro com endereço de IP
# 192.168.93.x
# se conectar ao banco de dados "template1" com o mesmo nome de usuário que
# "ident"
# informa para a conexão (normalmente o nome de usuário do Unix).
#
# TYPE  DATABASE  USER        CIDR-ADDRESS      METHOD
host   template1 all       192.168.93.0/24    ident sameuser

# Permitir o usuário do hospedeiro 192.168.12.10 se conectar ao banco de dados
# "template1" se a senha do usuário for fornecida corretamente.
#
# TYPE  DATABASE  USER        CIDR-ADDRESS      METHOD
host   template1 all       192.168.12.10/32    md5

# Na ausência das linhas "host" precedentes, estas duas linhas rejeitam todas
# as conexões oriundas de 192.168.54.1 (uma vez que esta entrada será
# correspondida primeiro), mas permite conexões Kerberos V de qualquer ponto
# da Internet. A máscara zero significa que não é considerado nenhum bit do
# endereço de IP do hospedeiro e, portanto, corresponde a qualquer hospedeiro.
#
# TYPE  DATABASE  USER        CIDR-ADDRESS      METHOD
host   all       all        192.168.54.1/32    reject
host   all       all        0.0.0.0/0         krb5

# Permite os usuários dos hospedeiros 192.168.x.x se conectarem a qualquer
# banco de dados se passarem na verificação de "ident". Se, por exemplo, "ident"
# informar que o usuário é "oliveira" e este requerer se conectar como o usuário
# do PostgreSQL "guest1", a conexão será permitida se houver uma entrada
# em pg_ident.conf para o mapa "omicron" informando que "oliveira" pode se
# conectar como "guest1".
#
# TYPE  DATABASE  USER        CIDR-ADDRESS      METHOD
host   all       all        192.168.0.0/16    ident omicron

# Se as linhas abaixo forem as únicas três linhas para conexão local, vão
# permitir os usuários locais se conectarem somente aos seus próprios bancos de
# dados (bancos de dados com o mesmo nome que seus nomes de usuário), exceto
# para os administradores e membros do grupo "suporte" que podem se conectar a
# todos os bancos de dados. O arquivo $PGDATA/admins contém a lista de nomes de
# usuários. A senha é requerida em todos os casos.
#
# TYPE  DATABASE  USER        CIDR-ADDRESS      METHOD
local  sameuser  all       md5
local  all       @admins    md5
local  all       +suporte   md5

# As duas últimas linhas acima podem ser combinadas em uma única linha:
local  all       @admins,+suporte md5

# A coluna banco de dados também pode utilizar listas e nomes de arquivos,
# mas não grupos:
local  db1,db2,@demodbs all      md5

```

Gerência dos Recursos do Core

Uma instalação grande do PostgreSQL pode ter seus recursos exauridos rapidamente.

Memória Compartilhada e Semáforo

A memória compartilhada e os semáforos são referenciados coletivamente como "System V IPC".

Quase todo sistema operacional moderno fornece estas funcionalidades, mas nem todos as têm ativas ou com tamanho suficiente por padrão.

Limites de Recursos

Os sistemas operacionais da família Unix obrigam respeitar vários tipos de limite de recursos que podem interferir com a operação do servidor PostgreSQL. São de particular importância os limites do número de processos por usuário, de número de arquivos abertos por processo, e a quantidade de memória disponível para cada processo. Cada um destes possui um limite "rígido" e um "flexível". O limite flexível é o que realmente conta, mas pode ser alterado pelo usuário até o limite rígido. O limite rígido somente pode ser alterado pelo usuário root.

Sobre Alocação de Memória no Linux

No Linux 2.4 e posteriores, o comportamento padrão de memória virtual não é o ótimo para o PostgreSQL. Devido à maneira como o núcleo implementa a sobre-alocação (overcommit) de memória, o núcleo pode fechar o servidor PostgreSQL (o processo postmaster), se a demanda por memória de um outro processo fizer com que o sistema fique sem memória virtual.

No Linux 2.6 e posteriores, uma solução melhor é modificar o comportamento do núcleo para que não haja "sobre-alocação" de memória. Isto é feito selecionando o modo estrito de sobre-alocação através do comando sysctl:

```
sysctl -w vm.overcommit_memory=2
```

pg_hba.conf em detalhes

Referências

- <http://pgdocptbr.sourceforge.net/pg80/client-authentication.html>
- <https://www.postgresql.org/docs/9.5/static/auth-methods.html>
- <https://www.postgresql.org/docs/9.5/static/auth-pg-hba-conf.html>

Autenticação é o processo usado pelo servidor de bancos de dados/PostgreSQL para estabelecer a identificação do cliente e determina se o aplicativo cliente em permissão para se conectar com o nome de usuário que foi informado.

Recomenda-se, por conta da segurança, que os usuários do postgresql sejam diferentes dos usuários do sistema operacional.

O arquivo pg_hba.conf

Este arquivo é carregado durante a inicialização do PostgreSQL.

A autenticação dos clientes é controlada pelo arquivo pg_hba.conf, que no Linux Mint 18.2/Ubuntu 16.04, fica em:

/etc/postgresql/9.5/main

HBA significa Host-Based Authentication (autenticação baseada no hospedeiro).

A localização do pg_hba.conf pode ser alterada no postgresql.conf, no parâmetro hba_file.

Apenas tome o cuidado de anotar as respectivas localizações e qual arquivo é o que está valendo, para evitar conflitos. Já vi se tentar alterar um arquivo quando era outro o que estava sendo usado.

Cada linha do pg_hba.conf contém um único registro

Cada registro contém vários campos separados por espaço ou tabulação

Cada registro contém os campos:

tipo_conexão banco(s) user(s) endereço_ip método_autenticação opções (opcional)

Exemplos:

local	all	postgres	peer	
host	banco1	user_um	192.168.0.10/32	md5
host	banco1	user_dois	192.168.20.0/24	md5

Importante: se um cliente tentar se conectar e não for bem sucedido em um dos registros ele será descartado sem tentar os registros seguintes.

Tipo de conexão - pode ser local, host, hostssl, hostnossal (somente conexões sem SSL)

Banco(s) - nome_do_banco, all (todos), podemos ter vários nomes de bancos separados por vírgula

User(s) - all (todos), podem ser inseridos vários nomes de usuários separados por vírgula

Endereço IP - Exemplos: somente uma máquina - 192.168.0.10/32, para toda uma rede 192.168.0.0/24. Aceita somente IP e nunca domínio

Para dar acesso à toda a internet usamos - 0.0.0.0/0

Método de autenticação (alguns):

trust - não requer senha (aconselhada para servidores com apenas um usuário)

peer - apenas conexão local

reject - rejeita conexão deste cliente

md5 - exige senha criptografada com md5 (para isso requer atribuição de senha com ... password 'senha')

ldap - autenticação usando LDAP

E outras

Opções - este campo é opcional e depende do tipo de autenticação escolhido.

Podemos inserir um registro para cada integrante de uma equipe de programadores, contendo o banco, seu nome, IP, método (geralmente md5)

Exemplo de arquivo pg_hba.conf original do PostgreSQL 9.5 no Linux Mint 18.2:

Início

```
# PostgreSQL Client Authentication Configuration File
# =====
#
# Refer to the "Client Authentication" section in the PostgreSQL
# documentation for a complete description of this file. A short
# synopsis follows.
#
# This file controls: which hosts are allowed to connect, how clients
# are authenticated, which PostgreSQL user names they can use, which
```

```
# databases they can access. Records take one of these forms:  
  
#  
  
# local   DATABASE USER METHOD [OPTIONS]  
  
# host    DATABASE USER ADDRESS METHOD [OPTIONS]  
  
# hostssl DATABASE USER ADDRESS METHOD [OPTIONS]  
  
# hostnossal DATABASE USER ADDRESS METHOD [OPTIONS]  
  
#  
  
# (The uppercase items must be replaced by actual values.)  
  
#  
  
# The first field is the connection type: "local" is a Unix-domain  
  
# socket, "host" is either a plain or SSL-encrypted TCP/IP socket,  
  
# "hostssl" is an SSL-encrypted TCP/IP socket, and "hostnossal" is a  
  
# plain TCP/IP socket.  
  
#  
  
# DATABASE can be "all", "sameuser", "samerole", "replication", a  
  
# database name, or a comma-separated list thereof. The "all"  
  
# keyword does not match "replication". Access to replication  
  
# must be enabled in a separate record (see example below).  
  
#  
  
# USER can be "all", a user name, a group name prefixed with "+", or a  
  
# comma-separated list thereof. In both the DATABASE and USER fields  
  
# you can also write a file name prefixed with "@" to include names  
  
# from a separate file.  
  
#  
  
# ADDRESS specifies the set of hosts the record matches. It can be a
```

```
# host name, or it is made up of an IP address and a CIDR mask that is
# an integer (between 0 and 32 (IPv4) or 128 (IPv6) inclusive) that
# specifies the number of significant bits in the mask. A host name
# that starts with a dot (.) matches a suffix of the actual host name.

# Alternatively, you can write an IP address and netmask in separate
# columns to specify the set of hosts. Instead of a CIDR-address, you
# can write "samehost" to match any of the server's own IP addresses,
# or "samenet" to match any address in any subnet that the server is
# directly connected to.

#
# METHOD can be "trust", "reject", "md5", "password", "gss", "sspi",
# "ident", "peer", "pam", "ldap", "radius" or "cert". Note that
# "password" sends passwords in clear text; "md5" is preferred since
# it sends encrypted passwords.

#
# OPTIONS are a set of options for the authentication in the format
# NAME=VALUE. The available options depend on the different
# authentication methods -- refer to the "Client Authentication"
# section in the documentation for a list of which options are
# available for which authentication methods.

#
# Database and user names containing spaces, commas, quotes and other
# special characters must be quoted. Quoting one of the keywords
# "all", "sameuser", "samerole" or "replication" makes the name lose
# its special character, and just match a database or username with
```

```
# that name.

#
# This file is read on server startup and when the postmaster receives
# a SIGHUP signal. If you edit the file on a running system, you have
# to SIGHUP the postmaster for the changes to take effect. You can
# use "pg_ctl reload" to do that.

# Put your actual configuration here

# -----
#
# If you want to allow non-local connections, you need to add more
# "host" records. In that case you will also need to make PostgreSQL
# listen on a non-local interface via the listen_addresses
# configuration parameter, or via the -i or -h command line switches.

# DO NOT DISABLE!

# If you change this first entry you will need to make sure that the
# database superuser can access the database using some other method.

# Noninteractive access to all databases is required during automatic
# maintenance (custom daily cronjobs, replication, and similar tasks).

#
# Database administrative login by Unix domain socket

local  all      postgres          peer
# TYPE DATABASE   USER   ADDRESS      METHOD
# "local" is for Unix domain socket connections only

local  all      all               peer
# IPv4 local connections:
```

```

host all      all      127.0.0.1/32      md5

# IPv6 local connections:

host all      all      ::1/128      md5

# Allow replication connections from localhost, by a user with the
# replication privilege.

#local replication postgres      peer
#host replication postgres      127.0.0.1/32      md5
#host replication postgres      ::1/128      md5

== Final

```

Logo após instalar o PostgreSQL eu atribuo senha ao mesmo pelo psql, assim:

```

sudo su

su - postgres

psql

alter role postgres password 'postgres';

```

Com isso eu mudei o método de autenticação do usuário postgres para md5.

Veja na linha:

```
local all      postgres      peer
```

A autenticação padrão para o postgres é peer, que é uma autenticação usada apenas para conexão local, portanto precisamos mudar de peer para md5 no pg_hba.conf e reiniciar o postgresql:

```
local all      postgres      md5

sudo /etc/init.d/postgresql restart
```

Agora já podemos conectar ao postgresql com o usuário postgres através de um cliente web como o adminer, ou com o PGAdmin.

Entendendo e manipulando o arquivo pg_ident.conf

O método de autenticação ident funciona obtendo o nome de usuário do sistema operacional do cliente, e determinando os nomes de usuário do PostgreSQL permitidos utilizando um arquivo de mapa que lista os pares de nomes de usuários correspondentes permitidos.

sameuser é o usuário padrão no ident.conf (significa o mesmo user do sistema operacional).

Este script de autenticação é utilizado nos UNIX, pois no Windows o usuário não é usuário com direito a login.

Exemplo de entrada no arquivo pg_hba.conf:

```
hostall all 10.0.0.7/32 ident      mapatest
```

Todos os usuários da máquina 10.0.0.7 terão acesso a todos os bancos do PostgreSQL. Os usuários desta máquina serão mapeados pelo mapa ident mapatest. No caso teremos os seguintes registros no pg_ident.conf:

```
mapatest joaojoaozinho
```

```
mapatest mich michael
```

```
mapatest danidaniel
```

Quando o usuário joao (em 10.0.0.7) se conecta remotamente ao servidor do PostgreSQL ele será mapeado para o usuário joaozinho, que é uma conta do sistema no servidor. Os demais de forma similar.

Caso precise usar este arquivo e ele não exista, crie no diretório sugerido acima, com as entradas desejadas.

Mais detalhes em:

<http://www.postgresql.org.br/Documenta%C3%A7%C3%A3o?action=AttachFile&do=get&target=postgresql.conf.pdf>

<http://pgdocptbr.sourceforge.net/pg80/auth-methods.html#AUTH-IDENT>

<http://www.postgresql.org/docs/8.3/interactive/client-authentication.html>

Livro PostgreSQL 8 for Windows de Richard Blum, capítulo 3.

Alerta Ao consultar a documentação fique atento para a versão do PostgreSQL que estás utilizando. Fortemente recomendada a consulta ao script postgresql.conf e sua documentação (comentários).

Configurações do Servidor

<https://www.postgresql.org/docs/9.5/static/runtime-config.html>

Configuração de Parâmetros

<https://www.postgresql.org/docs/9.5/static/config-setting.html>

- Nomes e Valores de Parâmetros

Todos os nomes de parâmetros são case-insensitivos.

Cada parâmetro tem um valor de um dos cinco tipos:

boolean, string, integer, floating point ou enumerado (enum).

listen_addresses (string): IP (IPV4 ou IPV6)

Porta (5432)

Configurar PostgreSQL

Para ser acessado pela rede

sudo nano /etc/postgresql/9.3/main/postgresql.conf

listen_addresses = '*'

Configurando o ambiente do PostgreSQL

Através do postgresql.conf e de comandos SQL 'SET'

Através do postgresql.conf

Lembrar que as linhas iniciadas com # estão comentadas e que todos os parâmetros comentados são o padrão. Caso queira alterar algum deixe comentado e copie a linha logo abaixo descomentada e com o valor desejado.

Alguns parâmetros importantes

```
#data_directory = 'ConfigDir'
```

Caso queira alterar o data_directory para outro diretório:

- copie o diretório atual ou mova-o para o novo diretório
- altere este parâmetro no postgresql.conf
- recarregue o serviço

```
#hba_file = 'ConfigDir/pg_hba.conf'
```

```
#listen_addresses = 'localhost'
```

Por default o PostgreSQL somente dá acesso para usuários locais, pela console.

Para dar permissão para acesso pelo PGAdmin, mesmo local e a outros hosts não locais devemos entrar com seu IP ou nome neste parâmetro.

Entrar com '*' para dar acesso a qualquer host.

Para uma lista de IPs separar por vírgula, assim: '10.0.0.7,10.0.0.8,10.0.0.9'.

Obs.: Mudanças requerem restartar serviço.

```
#port = 5432
```

```
#ssl = off
```

Uso de conexões seguras através do openssl. Mudanças requerem restartar serviço.

Para aumentar a segurança criptografando as comunicações cliente/servidor, o PostgreSQL possui suporte nativo para conexões SSL. É necessário que o OpenSSL esteja instalado tanto no cliente quanto no servidor, e que o suporte no PostgreSQL tenha sido ativado em tempo de instalação.

Detalhes em: <http://pgdocptbr.sourceforge.net/pg80/ssl-tcp.html> e

<http://www.postgresql.org/docs/8.3/interactive/ssl-tcp.html>

```
#password_encryption = on
```

Quando é especificada uma senha em CREATE USER ou ALTER USER , sem que seja escrito ENCRYPTED ou UNENCRYPTED, este parâmetro determina se a senha deve ser criptografada. Altamente recomendado o uso de senhas criptografadas.

```
max_connections = 100
```

Máximo de conexões simultâneas que serão atendidas pelo servidor. Manter o mais baixo possível, tendo em vista que cada conexão ativa requer recursos do hardware.

Mudanças requerem restartar serviço

autovacuum = on

habilita o subprocesso autovacuum

Para ativar, obrigatoriamente ative também track_counts (na versão 8.3)

#autovacuum_naptime = 1min

Tempo entre as execuções do vacuum

#datestyle = 'iso, mdy'

Formato/estilo da data

Para ter a data no formato do Brasil (dd/mm/yyyy) altere para: datestyle = 'sql, dmy'

#client_encoding = latin1

Na verdade, os padrões escolhidos pelo utilitário initdb são escritos no arquivo de configuração postgresql.conf apenas para servirem como padrão quando o servidor é inicializado. Se forem removidas as atribuições presentes no arquivo postgresql.conf, o servidor herdará as definições do ambiente de execução.

#lc_messages = 'C'

lc_messages = 'pt_BR' # Mensagens de erro em português do Brasil

Obs.: Alguns parâmetros do postgresql.conf podem ser alterados pelo comando "SET" do SQL e também pelo comando "ALTER".

Configurações Através do comando 'SET' do SQL

Através do comando SET podemos alterar as configurações do PostgreSQL.

O comando SET muda os parâmetros de configuração de tempo de execução. Muitos parâmetros de tempo de execução podem ser mudados dinamicamente pelo comando SET (Mas alguns requerem privilégio de superusuário para serem mudados, e outros não podem ser mudados após o servidor ou a sessão iniciar). O comando SET afeta apenas os valores utilizados na sessão atual.

Para exibir o formato de data atual do PostgreSQL:

```
SHOW DATESTYLE;
```

Execute a consulta:

```
SELECT CURRENT_DATE;
```

Para alterar o estilo da data na seção atual para o formato dd-mm-yyyy:

```
SET DATESTYLE TO 'postgresql,dmy';
```

Execute novamente a consulta:

```
SELECT CURRENT_DATE;
```

Para alterar o estilo da data na seção atual para o formato dd/mm/yyyy:

```
SET DATESTYLE TO 'sql,dmy';
```

Execute novamente a consulta:

```
SELECT CURRENT_DATE;
```

```
SET DATESTYLE TO ISO;
```

```
SELECT CURRENT_TIMESTAMP;
```

```
SET DATESTYLE TO PostgreSQL;
```

```
SELECT CURRENT_DATE;
```

```
SET DATESTYLE TO US;
```

```
SELECT CURRENT_DATE;
```

```
SET DATESTYLE TO NONEUROPEAN, GERMAN;
```

```
SELECT CURRENT_DATE;
```

```
SET DATESTYLE TO EUROPEAN;
```

```
SELECT CURRENT_DATE;
```

Para exibir todas as configurações:

```
SHOW ALL;
```

Também podemos Configurar um banco para Aceitar datas no formato praticado no Brasil:

`CREATE DATABASE bancobrasil;`

`ALTER DATABASE SET DATESTYLE TO 'SQL,DMY';`

Também pode ser atribuído juntamente com o Usuário:

`ALTER ROLE nomeuser SET DATESTYLE TO SQL, DMY;`

Alerta: para bancos que já estejam em uso, geralmente fica inviável uma alteração permanente (`postgresql.conf`), caso se esteja tratando as datas. Exemplo: ao consultar um campo data, a aplicação espera um formato e vai estar outro. A não ser que se altere consultas SQL existentes nos aplicativos.

`ALTER DATABASE cep_brasil SET client_encoding=win1252;`

ISO-8859-15 é o Latin9

`SHOW ENABLE_SEQSCAN;`

`SET ENABLE_SEQSCAN TO OFF;`

Mais detalhes em:

<http://pgdocptbr.sourceforge.net/pg80/sql-set.html> e

<http://www.postgresql.org/docs/8.3/interactive/sql-set.html>

Configurar estilo de datas para permitir entrada e exibição de datas assim:

31/12/2016

Configurar no script `postgresql.conf`

Assim fica configurado para todos os bancos do cluster

`datestyle = sql, dmy;`

Outros usos para `SHOW`:

`SHOW server_version;`

`SHOW server_encoding; -- Idioma para ordenação do texto (definido pelo initdb)`

`SHOW lc_collate; -- Idioma para classificação de caracteres (definido pelo initdb)`

`SHOW all; -- Mostra todos os parâmetros`

Também podemos setar o datestyle quando alteramos um banco:

```
ALTER DATABASE nomebanco SET DATESTYLE = SQL, DMY;
```

Também pode ser atribuído juntamente com o Usuário:

```
ALTER ROLE nomeuser SET DATESTYLE TO SQL, DMY;
```

48 - Entendendo e trabalhando com CLUSTERS no PostgreSQL

- 1 - Iniciando e parando o serviço do PostgreSQL
- 2 - Entendendo os Tablespaces
- 3 - Criação de bancos de dados
- 4 - Removendo bancos de dados
- 5 - Bancos de dados de template/modelo
- 6 - Entendendo o layout físico dos bancos de dados

A palavra "cluster" pode significar várias coisas diferentes dependendo do contexto:

1) Existe o cluster de tabelas:

<http://www.postgresql.org/docs/8.3/static/sql-cluster.html> ou
<http://www.postgresql.org/docs/8.3/static/app-clusterdb.html>

2) Existe o cluster do postgresql, que consiste em todos os arquivos que compõem um conjunto de bancos de dados administrados por uma instância do postgresql que sobe em uma porta só dele. Este cluster é criado pelo initdb:

<http://www.postgresql.org/docs/8.3/static/app-initdb.html>

3) Existe o conceito de cluster de bancos de dados que são várias instâncias de bancos de dados rodando em máquinas diferentes e se comportando como se fossem uma só. O PostgreSQL não possui uma implantação deste tipo.

4) Existe do pg_cluster que é uma implementação de replicação e é parecido com um cluster, mas na verdade é uma replicação:

<http://pgfoundry.org/projects/pgcluster/>

Para ver a diferença entre cluster e replicação, veja:

<http://www.midstorm.org/~telles/2007/08/24/cluster-replicacao/>

Cluster de Versões

1) Os pacotes do Debian fazem isso automaticamente, de forma limpa.

Você instala o 8.1 e 8.2 na mesma máquina e ele sobe cada um em uma porta diferente. Vale a pena conferir para ver como ele organiza as coisas, particularmente a estrutura de diretórios é o init.d.

(Fábio Telles na lista pgbr-geral.)

Instalação do PostgreSQL-8.3 através dos repositórios do Ubuntu 7.10

Para instalar o Servidor, acesse um terminal e digite:

```
sudo apt-get install postgresql-8.3 (com este pacote serão instalados o server, o client e o common)
sudo apt-get install postgresql-contrib-8.3
sudo apt-get install postgresql-doc-8.3 (ficará em /usr/share/doc/postgresql-doc-8.3/html/index.html)
```

Em máquinas que somente precisam ter o cliente instalado use:

```
sudo apt-get install postgresql-client-8.3
sudo apt-get install postgresql-doc-8.3
sudo apt-get install pgadmin3
```

Após instalar o 8.3 repita os passos para instalar também a versão 8.2, de forma que fiquemos com dois clusters instalados, um da versão 8.3 e outro da versão 8.2. O Ubuntu gerencia bem isso e de forma transparente. Geralmente o primeiro instalado fica na porta 5432 e o segundo na 5433.

Após instalar o pgadmin execute o script abaixo para habilitar algumas funções úteis:
 sudo -u postgres psql -d postgres < /usr/share/postgresql/8.3/contrib/adminpack.sql

Para acessar a console do psql:

Para acessar o 8.3 use
 sudo -u postgres psql

Para acessar o 8.2 use
 sudo -u postgres psql -p 5433

Ou mais adequadamente com:

sudo -u postgres /usr/lib/postgresql/8.2/bin/psql -p 5433

Observe que não será solicitada a senha do postgres. Isso devido ao método de autenticação usado por padrão no /etc/postgresql/8.3/main/pg_hba.conf, que identifica o usuário do sistema operacional, sem senha. Na primeira vez que acessar altere a senha do super-usuário:

ALTER ROLE postgres WITH ENCRYPTED PASSWORD 'postgres';
 Para oferecer mais segurança no acesso remoto, já que localmente não requer senha.

Para constatar os dois serviços no ar abra um terminal execute:

ps ax | grep post

Diretórios quando instalado pelos repositórios:

- Diretório base (com bancos e outros) - /var/lib/postgresql/8.3/main
- Arquivos de configuração (pg_hba.conf, pg_ident.conf e postgresql.conf) - /etc/postgresql/8.3/main
- Diretório de gerenciamento dos clusters - /etc/postgresql-common
- Diretório dos binários - /usr/lib/postgresql/8.3/bin

Exercício

Importar o banco de dados de CEP existente no CD ou em:
http://tudoemum.ribafs.net/includes/cep_brasil.sql.bz2

Faça o download e copie para a pasta /home/aluno
 Descompacte:

Abra o gerenciador de arquivos (locais – pasta pessoal). Clicando com o botão direito e extrair aqui.

Este banco de CEPs (do Brasil), não é completo nem está atualizado mas é uma boa base de testes e contém os CEPs das grandes cidades e capitais.

- Mude as permissões do arquivo cep_brasil_unique.csv para que o usuário postgres tenha permissão de acessá-lo:

sudo chown postgres:postgres cep_brasil_unique.csv

Experimente criar o banco na versão 8.3 com codificação latin1.

Ele não criará, informando incompatibilidade com as configurações do servidor/sistema operacional.
Para ter compatibilidade com latin1 no 8.3 devemos criar o clustar passando a codificação pelo comando initdb.

```
sudo -u postgres psql
create database cep_brasil with encoding 'latin1';
```

```
\c cep_brasil
create table cep_full (
    cep char(8),
    tipo char(72),
    logradouro char(70),
    bairro char(72),
    municipio char(60),
    uf char(2)
);
```

Execute os comandos abaixo e veja as informações:

```
\l
Depois:
\d
```

Importar o script de ceps para a tabela que acabamos de criar:

```
\copy cep_full from /home/ribafs/cep_brasil_unique.csv
```

Então, para ativar o cronômetro, execute:

```
\timing
```

Faça uma consulta que retorne apenas o registro com o CEP da sua rua:

```
select logradouro from cep_full where cep = '60420440';
```

Dual Core 2.2, 2GB RAM e Ubuntu 7.10 gastou 6711,204 ms.

Centrino 1.86 1GB de RAM e Ubuntu 7.10 gastou 8476.877 ms

Caso repita a consulta, levará apenas 514 ms na última máquina, pois está no cache.

Agora adicione uma chave primária na tabela:

```
ALTER TABLE cep_full add constraint cep_pk primary key (cep);
```

Então repita a mesma consulta anterior e veja a diferença de desempenho por conta do índice adicionado.
Tenha o cuidado de executar o comando ANALYZE antes da consulta.

Aqui gastou somente 1.135ms.

1 - Iniciando e parando o serviço do PostgreSQL

Parar:

```
sudo /etc/init.d/postgresql-8.3 stop
```

Iniciar:

```
sudo /etc/init.d/postgresql-8.3 start
```

Reiniciar:

```
sudo /etc/init.d/postgresql-8.3 restart
```

Para o 8.2 é de forma semelhante.

Observação

Estamos usando o script oferecido pela distribuição ou então aquele dos contribs que acompanha os fontes.

Manualmente:

```
sudo /usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data start
```

```
sudo /usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data stop
```

```
sudo /usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data restart
```

2 - Entendendo os Tablespace

Como o nome sugere, um espaço destinado a armazenar tabelas. Pode ser criado para proteção/segurança de algumas tabelas (que ficam em localização diferente do restante dos bancos) como para balancear carga e também para otimizar desempenho de apenas algumas tabelas ou bancos que requerem maior desempenho.

Utilizando espaços de tabelas, o administrador pode controlar a organização em disco da instalação do PostgreSQL. É útil pelo menos de duas maneiras:

Primeira: se a partição ou volume onde o agrupamento foi inicializado ficar sem espaço, e não puder ser estendido, pode ser criado um espaço de tabelas em uma partição diferente e utilizado até que o sistema possa ser reconfigurado.

Segunda: os espaços de tabelas permitem que o administrador utilize seu conhecimento do padrão de utilização dos objetos de banco de dados para otimizar o desempenho. Por exemplo, um índice muito utilizado pode ser colocado em um disco muito rápido com alta disponibilidade, como uma unidade de estado sólido. [4] Ao mesmo tempo, uma tabela armazenando dados históricos raramente utilizados, ou que seu desempenho não seja crítico, pode ser armazenada em um sistema de disco mais barato e mais lento.

Para definir um espaço de tabelas é utilizado o comando [CREATE TABLESPACE](#) como, por exemplo:

```
CREATE TABLESPACE nometablespace [ OWNER nomedono ] LOCATION 'diretório';
```

```
CREATE TABLESPACE area_veloz LOCATION '/mnt/sda1/postgresql/data';
```

Ao criar um banco, uma tabela ou um índice podemos escolher a qual tablespace pertencerá, usando o parâmetro tablespace.

Criar Novo Cluster

Caso sinta necessidade pode criar outros clusters, especialmente indicado para grupos de tabelas com muito acesso.

O comando para criar um novo cluster é:

```
\h create tablespace
```

Comando: CREATE TABLESPACE

Descrição: define uma nova tablespace

Sintaxe:

`CREATE TABLESPACE nome_tablespace [OWNER usuário] LOCATION 'diretório'`

Exemplo:

```
CREATE TABLESPACE ncluster OWNER usuário LOCATION '/usr/local/pgsql/nc';
CREATE TABLESPACE ncluster [OWNER postgres] LOCATION 'c:\\ncluster';
```

Importante: O diretório deve estar vazio e pertencer ao usuário.

Criando um banco no novo cluster:

```
CREATE DATABASE bdcluster TABLESPACE = ncluster;
```

Obs: Podem existir numa mesma máquina vários agrupamentos de bancos de dados (cluster) gerenciados por um mesmo ou por diferentes postmasters.

Se usando tablespace o gerenciamento será de um mesmo postmaster, se inicializados por outro initdb será por outro.

Setar o Tablespace default:

```
SET default_tablespace = tablespace1;
```

Listar os Tablespaces existentes:

```
\db
SELECT spcname FROM pg_tablespace;
```

Na documentação oficial em português do PG 8.0:

<http://pgdocptbr.sourceforge.net/pg80/manage-ag-tablespaces.html>

<http://pgdocptbr.sourceforge.net/pg80/sql-createtablespace.html>

E diversos outros capítulos fazem referência.

Na documentação do 8.3 em inglês:

<http://www.postgresql.org/docs/8.3/interactive/manage-ag-tablespaces.html>

<http://www.postgresql.org/docs/8.3/interactive/sql-createtablespace.html>

<http://www.postgresql.org/docs/8.3/interactive/sql-droptablespace.html>

<http://www.postgresql.org/docs/8.3/interactive/sql-altertablespace.html>

O Capítulo 5 do Livro Dominando o PostgreSQL aborda os tablespaces.

3 - Criação de bancos de dados

No prompt de comando:

```
createdb -p numeroporta -h nomehost -E LATIN1 -e nomebanco
```

Mais Detalhes: <http://www.postgresql.org/docs/8.3/interactive/app-createdb.html>

Como SQL dentro do psql ou em outro cliente:

```
CREATE DATABASE banco OWNER dono TABLESPACE nometablespace ENCODING 'LATIN1';
```

Mais Detalhes em: <http://www.postgresql.org/docs/8.3/interactive/sql-createdatabase.html>

4 - Removendo bancos de dados

DROP DATABASE nomebanco;

Este comando não pode ser desfeito nem executado dentro de uma transação.

Mais detalhes: <http://www.postgresql.org/docs/8.3/interactive/sql-dropdatabase.html>

dropdb -U usuario nomebanco

Detalhes em: <http://www.postgresql.org/docs/8.3/interactive/app-dropdb.html>

5 - Bancos de dados de template/modelo

Na verdade o comando CREATE DATABASE funciona copiando um banco de dados existente. Por padrão, copia o banco de dados padrão do sistema chamado template1. Portanto, este banco de dados é o "modelo" a partir do qual os novos bancos de dados são criados. Se forem adicionados objetos ao template1, estes objetos serão copiados nos próximos bancos de dados de usuário criados. Este comportamento permite modificar localmente o conjunto padrão de objetos nos bancos de dados. Por exemplo, se for instalada a linguagem procedural PL/pgSQL em template1, esta se tornará automaticamente disponível em todos os próximos bancos de dados dos usuários sem que precise ser feito qualquer procedimento adicional na criação dos bancos de dados.

Existe um segundo banco de dados padrão do sistema chamado template0. Este banco de dados contém os mesmos dados contidos inicialmente em template1, ou seja, contém somente os objetos padrão pré-definidos pela versão do PostgreSQL. **O banco de dados template0 nunca deve ser modificado após a execução do utilitário `initdb`. Instruindo o comando `CREATE DATABASE` para copiar template0 em vez de template1, pode ser criado um banco de dados de usuário "intacto", não contendo nenhuma adição feita ao**

banco de dados template1 da instalação local. É particularmente útil ao se restaurar uma cópia de segurança feita por `pg_dump`: o script da cópia de segurança deve ser restaurado em um banco de dados intocado, para garantir a recriação do conteúdo correto da cópia de segurança do banco de dados, sem conflito com as adições que podem estar presentes em template1.

Para criar um banco de dados copiando template0 deve ser utilizado:

CREATE DATABASE nome_do_banco_de_dados TEMPLATE template0;
a partir do ambiente SQL, ou

createdb -T template0 nome_do_banco_de_dados
a partir do interpretador de comandos.

É possível criar bancos de dados modelo adicionais e, na verdade, pode ser copiado qualquer banco de dados do agrupamento especificando seu nome como modelo no comando CREATE DATABASE. Entretanto, é importante compreender que não há intenção (ainda) que este seja um mecanismo tipo "COPY DATABASE" de uso geral. Em particular, é essencial que o banco de dados de origem esteja inativo (nenhuma transação em andamento alterando dados) durante a operação de cópia. O comando CREATE DATABASE verifica se nenhuma sessão (além da própria) está conectada ao banco de dados de origem no início da operação, mas não garante que não possa haver

alteração durante a execução da cópia, resultando em um banco de dados copiado inconsistente.

Portanto, recomenda-se que os bancos de dados utilizados como modelo sejam tratados como somente para leitura.

Após preparar um banco de dados modelo, ou fazer alguma mudança em um deles, é recomendado executar o comando VACUUM FREEZE ou VACUUM FULL FREEZE neste banco de dados. Se for feito quando não houver nenhuma outra transação aberta no mesmo banco de dados, é garantido que todas as linhas no banco de dados serão "congeladas" e não estarão sujeitas a problemas de recomeço do ID de transação. Isto é particularmente importante em um banco de dados que terá datallowconn definido como falso, uma vez que não será possível executar a rotina de manutenção VACUUM neste banco de dados. Para obter informações adicionais deve ser consultada a [Seção 21.1.3](#) do manual oficial em português do Brasil em: <http://pgdocptbr.sourceforge.net/pg80/maintenance.html#VACUUM-FOR-WRAPAROUND>.

Nota: Os bancos de dados template1 e template0 não possuem qualquer status especial além do fato do nome template1 ser o nome padrão para banco de dados de origem do comando CREATE DATABASE, e além de ser o banco de dados padrão para se conectar utilizado por vários programas, como o [createdb](#). Por exemplo, template1 pode ser removido e recriado a partir de template0 sem qualquer efeito prejudicial. Esta forma de agir pode ser aconselhável se forem adicionadas, por descuido, coisas inúteis ao template1.

Na versão 8 apareceu um novo template, que é uma cópia do template1, é o **postgres**, este a partir de então é o template default.

Recriação do banco de dados template1

Neste exemplo o banco de dados template1 é recriado. Deve ser observado na seqüência de comandos utilizada que não é possível remover o banco de dados template1 conectado ao mesmo, e enquanto este banco de dados estiver marcado como modelo no catálogo do sistema pg_database.

Para recriar o banco de dados template1 é necessário: se conectar a outro banco de dados (teste neste exemplo); atualizar o catálogo pg_database para que o banco de dados template1 não fique marcado como um banco de dados modelo; remover e criar o banco de dados template1; conectar ao banco de dados template1; executar os comandos VACUUM FULL e VACUUM FREEZE; atualizar o catálogo do sistema pg_database para que o banco de dados template1 volte a ficar marcado como um banco de dados modelo.

Abaixo está mostrada a seqüência de comandos utilizada:

```
template1=# DROP DATABASE template1;
ERRO: não é possível remover o banco de dados aberto atualmente
template1=# \c teste
Conectado ao banco de dados "teste".
teste=# DROP DATABASE template1;
```

ERRO: não é possível remover um banco de dados modelo

```
teste=# UPDATE pg_database SET datistemplate=false WHERE datname='template1';
UPDATE 1
```

```
teste=# DROP DATABASE template1;
DROP DATABASE
```

```
teste=# CREATE DATABASE template1 TEMPLATE template0 ENCODING 'latin1';
CREATE DATABASE
```

```
teste=# \c template1
```

Conectado ao banco de dados "template1".

```
template1=# VACUUM FULL;
```

```
VACUUM
```

```
template1=# VACUUM FREEZE;
```

```
VACUUM
```

```
template1=# UPDATE pg_database SET datistemplate=true WHERE datname='template1';
UPDATE 1
```

Mais detalhes em: <http://pgdocptbr.sourceforge.net/pg80/manage-ag-templatedbs.html>

6 - Entendendo o layout físico dos bancos de dados

Formato de armazenamento no nível de arquivos e diretórios.

Todos os dados necessários para um agrupamento de bancos de dados são armazenados dentro do diretório de dados do agrupamento. Podem existir na mesma máquina vários agrupamentos, gerenciados por diferentes postmaster.

Quando instalamos pelos fontes e não alteramos o default ficam em /usr/localpgsql/data.

O diretório data contém vários subdiretórios e arquivos de controle, conforme mostrado na tabela abaixo. Além destes itens requeridos, os arquivos de configuração do agrupamento postgresql.conf, pg_hba.conf e pg_ident.conf são tradicionalmente armazenados no data (embora a partir da versão 8.0 do PostgreSQL seja possível mantê-los em qualquer outro lugar).

Conteúdo de diretório data (lembre: algumas distribuições usam outro diretório)

Item	Descrição
PG_VERSION	Arquivo contendo o número de versão principal do PostgreSQL
base	Subdiretório contendo subdiretórios por banco de dados
global	Subdiretório contendo tabelas para todo o agrupamento, como pg_database
pg_clog	Subdiretório contendo dados sobre status de efetivação de transação
pg_subtrans	Subdiretório contendo dados sobre status de subtransação
pg_tblspc	Subdiretório contendo vínculos simbólicos para espaços de tabelas
pg_xlog	Subdiretório contendo os arquivos do WAL (registro prévio da escrita)
postmaster.opts	Arquivo contendo as opções de linha de comando com as quais o postmaster foi inicializado da última vez
postmaster.pid	Arquivo de bloqueio contendo o PID corrente do postmaster, e o ID do segmento de memória compartilhada (não mais presente após o postmaster ser parado)

Em caso de problema ao tentar iniciar o PostgreSQL remova o postmaster.pid.

Para cada banco de dados do agrupamento existe um subdiretório dentro de PGDATA/base, com nome correspondente ao OID do banco de dados em pg_database. Este subdiretório é o local padrão para os arquivos do banco de dados; em particular, os catálogos do sistema do banco de dados são armazenados neste subdiretório.

Cada tabela e índice é armazenado em um arquivo separado.

Mais detalhes em: <http://pgdocptbr.sourceforge.net/pg80/storage.html> e em <http://www.postgresql.org/docs/8.3/interactive/storage-file-layout.html>

Criando Novos Clusters no PostgreSQL para Windows

Para criar um novo cluster:

Crie um diretório para abrigar o novo cluster (lembre que o usuário postgres deve ter permissão de escrita nele).

- Ex.: data2 no diretório bin.

Criar o novo cluster (acesse o terminal no diretório C:\Program Files\PostgreSQL\8.3\bin) e execute:

- C:\Program Files\PostgreSQL\8.3\bin>initdb -U postgres -D data2

Editar o data2\postgresql.conf e alterar a porta para 5444

Iniciar o servidor do novo cluster

- pg_ctl -D data2 start

Acessar a console (psql) do novo cluster

- psql -p 5444 -U postgres

Listar os bancos

- \l — Observe que somente existem os bancos de templates. Temos um novo cluster.

Obs.: No Windows não consegui dar suporte a latin1 em novos clusters nem o original suporta.

Isso só foi conseguido em novos clusters no Linux.

Tecle Ctrl+Alt+Del no Windows ou 'ps ax|grep post' no Linux e veja que agora temos dois pg_ctl na memória.

Criação de Novos Clusters no PostgreSQL 8.3 for Linux (Ubuntu 7.10):

Criando os clusters

cluster em latin1

Criação do diretório para o cluster, data_latin1, tornando o usuário postgres seu dono:

mkdir data_latin1

su - postgres

```
export LANG=pt_BR.ISO-8859-1  
bin/initdb –encoding latin1 -D data_latin1  
Editar o script data_latin1/postgresql.conf e alterar a porta para 5433
```

49 - Trabalhando com Esquemas no PostgreSQL

- 1) O que são esquemas e sua importância
- 2) Gerenciando esquemas

1) O que são esquemas e sua importância

Usuários e grupos são compartilhados entre os vários bancos do cluster, mas nenhum dado é compartilhado entre os bancos.

Quando qualquer usuário ou aplicação cliente se conecta ao servidor, somente tem acesso ao banco ao qual se conecta.

Um banco de dados pode conter um ou mais esquemas, que podem conter tabelas e outros objetos.

Um usuário pode acessar todos os esquemas de um banco, desde que tenha os devido privilégios.

Razões para se usar esquemas:

- Permitir que vários usuários usem um banco sem interferir no demais;
- Organizar as tabelas em grupos/esquemas

Esquemas parecem com diretórios, exceto que não podem ser aninhados. Não podemos ter um esquema dentro de outro esquema.

Sugestão: usar o nome do esquema "externo", como prefixo.

Tenho um esquema dp e quero que o esquema k1 pertença ao dp:

dp

k1

dp
dp_k1
dp_gab
dp_secretaria
etc

Referência a esquemas:

esquema.tabela

Quem cria um objeto torna-se seu dono, por isso quem cria um esquema é seu dono.
Para criar um esquema e tornar outro usuário seu dono precisa ser super usuário.

```
create role dp_k1 with login password 'senha';
create schema dp_k1 authorization dp_k1;
ou
alter schema dp_k1 owner to dp_k1;
```

Consultar ordem de pesquisa:

```
show search_path;
```

Remover esquema public:

```
drop schema public;
```

Configurar a ordem de pesquisa dos esquemas:

```
set search_path to dp_k1;
```

No Adminer fica transparente, pois precisamos abrir o esquema para trabalhar nele.

Cada banco de dados, além do esquema public e dos esquemas criados pelo usuário tem o esquema pg_catalog que contém as tabelas de sistema e todos os tipos de dados, funções e operadores. O pg_catalog é sempre parte do search_path.

É permitido mas é prudente evitar o prefixo pg_ no nome das tabelas. Em bancos pequenos e com apenas um usuário não existe a necessidade de se criar esquemas.

Tabelas e outros objetos que serão compartilhados por vários esquemas devem ser criados num esquema separado.

Para maior portabilidade para cada usuário criar um esquema com o mesmo nome dele. Como também evitar o uso ou remover o esquema public.

As tabelas são os principais objetos de bancos de dados relacionais.

Outros objetos:

- Views
- Funções e operadores
- Tipos de dados e domínios
- Trigger e rewrite rules

Um agrupamento de bancos de dados do PostgreSQL contém um ou mais bancos de dados com nome. Os usuários e os grupos de usuários são compartilhados por todo o agrupamento, mas nenhum outro dado é compartilhado entre os bancos de dados. Todas as conexões dos clientes com o servidor podem acessar somente os dados de um único banco de dados, àquele que foi especificado no pedido de conexão.

Nota: Os usuários de um agrupamento de bancos de dados não possuem, necessariamente, o privilégio de acessar todos os bancos de dados do agrupamento. O compartilhamento de nomes de usuários significa que não pode haver, em dois bancos de dados do mesmo agrupamento, mais de um usuário com o mesmo nome como, por exemplo, joel; mas o sistema pode ser configurado para permitir que o usuário joel acesse apenas determinados bancos de dados.

Um banco de dados contém um ou mais *esquemas* com nome, os quais por sua vez contêm tabelas. Os esquemas também contêm outros tipos de objetos com nome, incluindo tipos de dado, funções e operadores. O mesmo nome de objeto pode ser utilizado em esquemas diferentes sem conflito; por exemplo, tanto o esquema_1 quanto o meu_esquema podem conter uma tabela chamada minha_tabela. Diferentemente dos bancos de dados, os esquemas não são separados rigidamente: um usuário pode acessar objetos de vários esquemas no banco de dados em que está conectado, caso possua os privilégios necessários para fazê-lo.

Existem diversas razões pelas quais pode-se desejar utilizar esquemas:

- Para permitir vários usuários utilizarem o mesmo banco de dados sem que um interfira com o outro.

- Para organizar objetos do banco de dados em grupos lógicos tornando-os mais gerenciáveis.
- Os aplicativos desenvolvidos por terceiros podem ser colocados em esquemas separados, para não haver colisão com nomes de outros objetos.

Os esquemas são análogos a diretórios no nível do sistema operacional, exceto que os esquemas não podem ser aninhados.

Mais detalhes em: <http://pgdocptbr.sourceforge.net/pg80/ddl-schemas.html>

<http://www.postgresql.org/docs/8.3/interactive/ddl-schemas.html>

\dn – visualizar esquemas

Um banco de dados pode conter vários esquemas e dentro de cada um desses podemos criar várias tabelas. Ao invés de criar vários bancos de dados, criamos um e criamos esquemas dentro desse. Isso permite uma maior flexibilidade, pois uma única conexão ao banco permite acessar todos os esquemas e suas tabelas. Portanto devemos planejar bem para saber quantos bancos precisaremos, quantos esquemas em cada banco e quantas tabelas em cada esquema.

Cada banco ao ser criado traz um esquema public, que é onde ficam todas as tabelas, caso não seja criado outro esquema. Este esquema public não é padrão ANSI. Caso se pretenda ao portável devemos excluir este esquema public e criar outros. Por default todos os usuários criados tem privilégio CREATE e USAGE para o esquema public.

É muito útil para estes casos criar um único banco e neste criar vários esquemas, organizados por áreas: pessoal, administracao, contabilidade, engenharia, etc.

Mas e quando uma destas áreas tem outras sub-áreas, como por exemplo a engenharia, que tem reservatórios, obras, custos e cada um destes tem diversas tabelas. O esquema engenharia ficará muito desorganizado. Em termos de organização o ideal seria criar um banco para cada área, engenharia, contabilidade, administração, etc. E para engenharia, por exemplo, criar esquemas para cada subarea, custos, obras, etc. Mas não o ideal em termos de comunicação e acesso entre todos os bancos.

Quando se cria um banco no PostgreSQL, por default, ele cria um esquema público (public) no mesmo e é neste esquema que são criados todos os objetos quando não especificamos o esquema. A este esquema public todos os usuários do banco têm livre acesso, mas aos demais existe a necessidade de se dar permissão para que os mesmos acessem.

"Esquemas são partições lógicas de um banco de dados. São formas de se organizar logicamente um banco de dados em conjuntos de objetos com características em comum sem criar bancos de dados distintos. Por exemplo, podem ser criados esquemas diferentes para dados dos níveis operacional, tático e estratégico de uma empresa, ou ainda esquemas para dados de cada mês de um ano.

Podem ser utilizados também como meios de se estabelecer melhores procedimentos de segurança, com autorizações de acesso feitas por esquema ao invés de serem definidas por objeto do banco de dados.

O PostgreSQL possui um conjunto de esquemas padrão, sendo que a inclusão de objetos do usuário por padrão é feita no esquema PUBLIC. Ao se criar um banco de dados, o banco apresentará inicialmente os seguintes esquemas:

- information_schema – informações sobre funções suportadas pelo banco. Armazena informações sobre o suporte a SQL, linguagens suportadas e tamanho máximo de variáveis como nome de tabela, identificadores, nomes de colunas, etc.
- pg_catalog – possui centenas de funções e dezenas de tabelas com os metadados do sistema. Guarda informações sobre as tabelas, suas colunas, índices, estatísticas, tablespaces, triggers e demais objetos.
- pg_toast – Informações relativas ao uso de TOAST (The Oversized-Attribute Storage Technique).
- public – Esquema com as tabelas e objetos do usuário.

Exibir a ordem de busca dos Esquemas:

```
SHOW SEARCH_PATH;
```

```
postgres=# SHOW SEARCH_PATH;
search_path
```

```
-----  
"$user",public
```

\$user – mostra que será procurado o esquema com o mesmo nome do usuário atual.
public – que será procurado no esquema public.

Alterando a ordem de busca dos Esquemas adicionando o esquema esquema1:

```
SET SEARCH_PATH TO public, esquema1;
```

Fonte: <http://postgresqlbr.blogspot.com/2007/06/esquemas-no-postgresql.html>

Criando Um Esquema

```
CREATE SCHEMA nomeesquema;
```

Criando Esquema e tornando um Usuário dono

```
CREATE SCHEMA nomeesquema AUTHORIZATION nomeusuario;
```

Removendo privilégios de acesso a usuário em esquema

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC
```

Com isso estamos tirando o privilégio de todos os usuários acessarem o esquema public.

Excluindo Um Esquema

```
DROP SCHEMA nomeesquema;
```

Aqui, quando o esquema tem tabelas em seu interior, não é possível apagar dessa forma, temos que utilizar:

```
DROP SCHEMA nomeesquema CASCADE;
```

Que apaga o esquema e todas as suas tabelas, portanto muito cuidado.

Obs.: O padrão SQL exige que se especifique RESTRICT (default no PostgreSQL) OU CASCADE, mas nenhum SGBD segue esta recomendação.

Obs.: é recomendado ser explícito quanto aos campos a serem retornados, ao invés de usar * para todos, entrar com os nomes de todos os campos. Assim fica mais claro. Além do mais a consulta terá um melhor desempenho.

Acessando Tabelas Em Esquemas

```
SELECT * FROM nomeesquema.nometabela;
```

Privilégios Em Esquemas

\dp – visualizar permissões

REVOKE CREATE ON SCHEMA public FROM PUBLIC; -- Remove o privilégio CREATE de todos os usuários.

Obtendo Informações sobre os Esquemas:

\dn

```
SELECT current_schema();
SELECT current_schemas(true);
SELECT current_schemas(false);
```

Schemas ou Databases?

Fabio Telles - na lista de postgresql-br

Realmente é difícil justificar o uso de bancos de dados separados. Em geral, utilizar schemas é sempre mais indicado. Vale a pena lembrar que existem 3 níveis de compartilhamento num mesmo servidor:

- * Cluster, que compartilham a mesma porta, os mesmos processos e a mesma configuração global (postgresql.conf e pg_hba.conf);
- * Bancos de Dados, que compartilham o mesmo cluster mas podem ter codificação de caracteres distintos e dependendo da configuração, podem ter usuários distintos (se 'db_user_namespace' for configurado como ON);
- * Schemas, que compartilham o mesmo banco de dados;

Em geral, utilizar mais de um banco de dados pode ser uma boa se:

- Você realmente precisa ter usuários com poderes plenos num banco de dados sem afetar outras aplicações. Isto é muito incomum, pois é normal você dar permissões para um usuário em um schema específico.

Mas se você quiser soltar seu estagiário para brincar no seu servidor... esta pode ser uma opção (mas eu instalaria o PostgreSQL localmente na máquina dele)

- Você precisa de ambientes de teste, homologação e/ou produção na mesma máquina. Em geral é melhor deixar seu ambiente de teste em outro servidor. Você vai descobrir que uma única consulta mal feita no ambiente de teste pode sentar todo o servidor... melhor evitar isso a todo o custo! Você também pode criar ambientes isolados em clusters separados e garantir mais recursos para o ambiente de testes ao ambiente de produção e ainda deixar os ambientes mais isolados, utilizando portas distintas.

- Você precisa de bancos de dados com codificações de caracteres diferentes. Bom... seria ideal você pensar em utilizar utf-8 para todas as bases se você está nesta situação. Mas isto é uma looooonga conversa, para lá de complicada.

- Você precisa de configurações de desempenho diferentes para diferentes aplicações (OLTP x BI por exemplo). Bom, você pode fazer isso com schemas também... especificando parâmetros por usuário, ao invés de parâmetros por banco de dados, mas se você quer levar isto realmente a sério, é melhor criar clusters distintos e melhor ainda, utilizar servidores distintos.

Conclusão:

Se mais de uma aplicação podem conviver no mesmo servidor e no mesmo cluster, então elas podem estar no mesmo banco de dados. Existem raras casos em que isto não se aplique... mas são exceções e não a regra.

Criei um pequeno tutorial explicando como unificar vários bancos de dados distintos utilizando schemas... vide:

<http://www.midstorm.org/~telles/2006/09/28/unificando-bases-de-dados-com-schemas/>

Atenciosamente,
Fábio Telles

O ideal seria realmente o uso de schemas e centralização de informação. Se você mantiver diversos banco de dados, para cada bando terá que abrir uma conexão, ou então ficar mudando o tempo todo de banco em uso nas suas queries, o que pode causar facilmente erros. Se você tem tabelas compartilhadas entre os sistemas, melhor centralizar tudo mesmo. Aí, você pode aumentar o número de conexões simultâneas sem maiores problemas.

Hoje, onde trabalho, tenho os banco de dados divididos em schemas.

Por exemplo.

O schema security tem tabelas relacionados com segurança como menus dos usuários, direito de cada módulo para cada usuário.

No schema scp temos o sistema de controle de processos.

No schema comum, temos tabelas que são utilizados por qualquer sistema ou schema, como por exemplo o cadastro de usuários que acessam o sistema.

Quando você coloca em banco de dados separado, você pode fazer views para fazer select com junções entre tabelas de banco de dados distintos, mas para isso vc tem que compilar e instalar o módulo dblink e utilizá-lo sempre que efetuar consultas com bancos distintos.

O schema facilita muito a criação de views e selects de tabelas distribuidas.

Quanto ao desempenho, bem, não notei redução de desempenho em nenhum serviço rodando nas 350 estações.

Uma dica. O banco de dados depende muito das operações de leitura e escrita. O mais importante no servidor é ter um bom disco e a configuração do postgresql.conf bem afinada.

Se você ver que o seu disco está sendo muito utilizado e o servidor está meio lento, você pode dividir as tabelas ou até mesmo os schemas em tablespaces. Isso permite vc colocar tabelas em um segundo disco. Quando você faz isso, o planejador pode trabalhar de forma a dividir o trabalho dos discos. Por exemplo:

Você tem os schemas A, B e C. Cada schema tem seu próprio sistema e cada sistema tem em média 100 usuários conectado simultaneamente.

Se você colocar em 3 discos e colocar cada schema em um disco, quando o usuário do sistema A, B e C gravarem dados simultaneamente o postgres vai gravar esses dados mais rápido. Isso porque cada transação é uma thread e as threads vão trabalhar em paralelo gravando cada um em um disco ao invés de uma thread esperar os dados do usuário A gravar para começar a gravar os dados do usuário B.

Cleberson Costa Silva.

Se você tem cópias de tabelas mantidas em seus diversos bancos de dados e necessita manter a integridade entre elas então é melhor utilizar esquemas. Se seus bancos de dados são totalmente independentes então é melhor mantê-los separados. Analise sob o aspecto da integridade dos dados e seu peso nos sistemas.

Esquemas - <https://www.javatpoint.com/postgresql-schema>
<https://www.postgresql.org/docs/9.6/static/ddl-schemas.html>

Conectando no cluster em latin1

bin/pg_ctl -D data_latin1 start

bin/psql -U postgres postgres -p 5433

\l — Veja que a codificação de todos os bancos é a latin1.

create database testeutf8 with encoding 'utf8';

Obs.: Cluster em latin1 com suporte a UTF-8.

cluster em utf-8

su - postgres

bin/initdb -D data_utf8

Como utf8 o default no Ubuntu, não preciso passar parâmetro.

Editar o script data_utf8/postgresql.conf e alterar a porta para 5434

Conectando no cluster utf-8

bin/pg_ctl -D data_utf8 start

bin/psql -U postgres postgres -p 5434

Bem, a saída para quem quer usar o 8.3.x e precisa de latin, no Linux, é esta (dica que recebi na lista de PostgreSQL pgbr-geral, do Euler).

Fonte: <http://ribafs.wordpress.com/2008/04/01/criando-clusters-no-postgresql-83-windows-e-linux/>

50 - Manutenção do PostgreSQL

Usando o Vacuum e Analyze

Otimizando o Desempenho com Vacuum

Gerenciar e otimizar o espaço em disco. No PostgreSQL as alterações e remoções de registros são apenas marcadas mas o espaço continua ocupado. Para a remoção de fato precisamos executar o comando vacuum. Existe também o comando vacuumdb para a linha de comando.

Sintaxe

```
vacuum [full | freeze | verbose] [tabela]
      vacuum [full | freeze |verbose] analyze [tabela]{  
campo[,...])}]
```

```
\c intranet
vacuum verbose lotes;
```

O vacuum atualiza a tabela de estatísticas.

```
vacuum analyze perímetros;
```

ANALYZE - Atualiza as estatísticas do banco de dados no catálogo (na tabela pg_statistic).

Após o seu uso o PostgreSQL melhora seu desempenho, tomando decisões mais adequadas para atender às solicitações dos usuários.

Sintaxe

```
analyze [verbose [[tabela[(campo[,...])]]]
```

```
\c intranet
analyze lotes;
analyze verbose lotes;
```

Autovacuum:

<https://www.postgresql.org/docs/9.6/static/runtime-config-autovacuum.html>
<https://www.postgresql.org/docs/9.6/static/routine-vacuuming.html#AUTOVACUUM>

Atualmente o autovacuum roda como um processo, veja:

```
ps ax | grep postgres
```

```
1325 ? Ss 0:00 postgres: autovacuum launcher process
```

Otimizando o Banco
vacuum;

```

vacuum -a
    -a todos os databases
vacuumdb -f -a -z
    -f
    -a todos os databases
    -z analize

```

Uso do Vacuum

VACUUM

Vacuum - limpa e opcionalmente analisa um banco de dados. Recupera a área de armazenamento ocupada pelos registros excluídas. Na operação normal do PostgreSQL os registros excluídos, ou tornados obsoletos por causa de uma atualização, não são fisicamente removidos da tabela; permanecem presentes até o comando VACUUM ser executado. Portanto, é necessário executar o comando VACUUM periodicamente, especialmente em tabelas freqüentemente atualizadas.

Sem nenhum parâmetro, o comando VACUUM processa todas as tabelas do banco de dados corrente. Com um parâmetro, o comando VACUUM processa somente esta tabela. O comando VACUUM ANALYZE executa o VACUUM e depois o ANALYZE para cada tabela selecionada. Esta é uma forma de combinação útil para scripts de rotinas de manutenção. Para obter mais detalhes sobre o seu processamento deve ser consultado o comando ANALYZE.

O comando VACUUM simples (sem o FULL) apenas recupera o espaço, tornando-o disponível para ser reutilizado. Esta forma do comando pode operar em paralelo com a leitura e escrita normal da tabela, porque não é obtido um bloqueio exclusivo. O VACUUM FULL executa um processamento mais extenso, incluindo a movimentação das tuplas entre blocos para tentar compactar a tabela no menor número de blocos de disco possível. Esta forma é muito mais lenta, e requer o bloqueio exclusivo de cada tabela enquanto está sendo processada.

Parâmetros

FULL

Seleciona uma limpeza "completa", que pode recuperar mais espaço, mas é muito mais demorada e bloqueia a tabela no modo exclusivo.

FREEZE

Seleciona um "congelamento" agressivo das tuplas. Especificar FREEZE é equivalente a realizar o VACUUM com o parâmetro vacuum_freeze_min_age definido como zero. A opção FREEZE está em obsolescência e será removida em uma versão futura; em vez de utilizar esta opção deve ser definido o parâmetro vacuum_freeze_min_age. (adicionar ao postgresql.conf).

`vacuum_freeze_min_age (integer)`

Specifies the cutoff age (in transactions) that VACUUM should use to decide whether to replace transaction IDs with FrozenXID while scanning a table. The default is 100000000 (100 million). Although users can set this value anywhere from zero to 1000000000, VACUUM will silently limit the effective value to half the value of

`autovacuum_freeze_max_age`, so that there is not an unreasonably short time between forced autovacuums. For more information see Seção 22.1.3.

A convenient way to examine this information is to execute queries such as `SELECT relname, age(relfrozenxid) FROM pg_class WHERE relkind = 'r';`
`SELECT datname, age(datfrozenxid) FROM pg_database;`

VERBOSE

Mostra, para cada tabela, um relatório detalhado da atividade de limpeza.

ANALYZE

Atualiza as estatísticas utilizadas pelo planejador para determinar o modo mais eficiente de executar um comando.

tabela

O nome (opcionalmente qualificado pelo esquema) da tabela específica a ser limpa. Por padrão todas as tabelas do banco de dados corrente.

coluna

O nome da coluna específica a ser analisada. Por padrão todas as colunas.

Executando o Vacuum Manualmente

Para uma tabela

`VACUUM ANALYZE tabela;`
`VACUUM VERBOSE ANALYZE clientes;`

Para todo um banco

`\c nomebanco`
`VACUUM FULL ANALYZE;`

Encontrar as 5 maiores tabelas e índices

`SELECT relname, relpages FROM pg_class ORDER BY relpages DESC LIMIT 5;`

Ativando o daemon do auto-vacuum

Iniciando na versão 8.1 é um processo opcional do servidor, chamado de autovacuum daemon, cujo uso é para automatizar a execução dos comandos VACUUM e ANALYZE.

Roda periodicamente e checa o uso em baixo nível do coletor de estatísticas.

Só pode ser usado se `stats_start_collector` e `stats_row_level` forem alterados para true.

Veja detalhes na aula 5 do módulo 2 (Monitorando as Atividades do Servidor do PostgreSQL).

Por default será executado a casa 60 segundos. Para alterar descomente e mude a linha:
autovacuum_naptime = 60

Autovacuum

Assim que você entra em produção no 8.0, você vai querer fazer um plano de manutenção incluindo VACUUMs e ANALYZEs. Se seus bancos de dados envolvem um fluxo contínuo de escrita de dados, mas não requer a maciças cargas e apagamentos de dados ou freqüentes reinícios, isto significa que você deve configurar o pg_autovacuum. Isto é melhor que agendar vaccuns porque:

- * Tabelas sofrem o vacuum baseados nas suas atividades, excluindo tabelas que apenas sofrem leituras.
- * A freqüência dos vaccums cresce automaticamente com o crescimento da atividade no banco de dados.
- * É mais fácil calcular o mapa de espaço livre e evitar o inchaço do banco de dados.

Configurando o autovacuum requer a fácil compilação de um módulo do diretório contrib/pg_autovacuum da fonte do seu PostgreSQL (usuários Windows devem procurar o autovacuum incluído no pacote do instalador). Você liga as estatísticas de configuração detalhadas no README. Então você inicia o autovacuum depois de o PostgreSQL ser iniciado como um processo separado; ele será desligado automaticamente quando o PostgreSQL desligar.

As configurações padrões do autovacuum são muito conservadores, imagino, e são mais indicadas para bancos de dados muito pequenos. Eu geralmente uso algo mais agressivo como:

```
-D -v 400 -V 0.4 -a 100 -A 0.3
```

Isto irá rodar o vacuum nas tabelas após 400 linhas + 40% da tabela ser atualizada ou apagada e irá rodar o analyze após 100 linhas + 30% das tabelas sofrer inserções, atualizações ou ser apagada. As configurações acima também me permitem configurar o meu max_fsm_pages para 50% das páginas de dados com a confiança de que este número não será subestimado gerando um inchaço no banco de dados. Nós atualmente estamos testando várias configurações na OSDL e teremos mais informações em breve.

Note que você também pode usar o autovacuum para configurar opções de atraso ao invés de configura-lo no PostgreSQL.conf. O atraso no Vacuum pode ser de vital importância em sistemas que tem tabelas e índices grandes; em último caso pode parar uma operação importante.

Existem infelizmente um par de limitações sérias para o autovacuum no 8.0 que serão eliminadas em versões futuras:

* Não tem memória de longa duração: autovacuum esquece toda a sua atividade quando você reinicia o banco de dados. Então se você reinicia regularmente, você deve realizar um VACUUM ANALYZE em todo o banco de dados imediatamente antes ou depois.

* Preste atenção em quanto o servidor está ocupado: há planos de checar a carga do servidor antes de realizar o vacuum, mas não é uma facilidade corrente. Então se você tem picos de carga extremos, o autovacuum não é para você.

Fábio Telles em - <http://www.midstorm.org/~telles/2006/10/>

Tudo que você sempre quis saber sobre discos em servidores PostgreSQL e tinha vergonha de perguntar

Leandro Dutra

Este é um texto que eu tenho vontade de escrever já faz bem um ano. Não escrevi antes por preguiça (é um texto um pouco longo) e porque é um texto um tanto pretensioso então tive receio de falar muita bobagem. Esta semana eu resolvi correr o risco. Já fazia um tempo que eu não me debruçava sobre um texto técnico um pouco mais longo, então tomei coragem e coloquei as idéias no papel, ou melhor, no blog. Muitos DBAs experientes sentirão que eu posso ter escorregado ou simplificado demais aqui ou ali. O propósito era escrever algo didático e não um livro inteiro sobre o assunto. Mas se você encontrar imprecisões técnicas ou tiver alguma sugestão para melhorar/corrigir o texto, por favor deixe um comentário.

Todo DBA é ou deveria ser tarado por discos. A não ser que você seja um daqueles que acreditam que um banco de dados possa conviver inteiro na memória sem nenhum problema (já ouviram falar de um SGDB escrito em Java que é mais rápido que o MySQL e o PostgreSQL?) você verá que os discos são o ponto chave no desempenho, na segurança e no custo do hardware. Neste texto vamos abordar alguns aspectos importantes e tentar visualizar um pouco deles em termos práticos no final. Vejamos alguns tópicos a serem abordados:

- RAID
- Discos
- Controladoras de Discos
- Tipos de Arquivos
- Particionamento
- Como distribuir as partições nos discos existentes

RAID

Em termos de desempenho o mantra sempre foi “quanto mais discos melhor”. Mas algo mudou no meio do caminho. Há tempos atrás as pessoas ficavam fazendo uma ginastica

danada para distribuir os tablespaces em discos diferentes. O resultado era a paralelização no acesso ao disco. Se bem realizado com sucesso, o processo iria dobrar a velocidade quando uma operação fosse realizada em dois discos ao mesmo tempo. Fazer isso não é simples. Uma formula mágica muito divulgada era o de separar os índices das tabelas. Como é comum acessar uma tabela e acessar seus índices também, isto parecia fazer muito sentido. Surpresa, não é bem assim que as coisas funcionam. Você tem que fazer uma avaliação real de quais objetos (tabelas e índices) são acessados mais concomitante mente. Ocorre que o otimizador ao fazer uma consulta que envolve várias tabelas pode realizar o acesso a disco de diversas formas diferentes. De acordo com a época do ano, a frequência no acesso aos objetos pode mudar, enfim, são inúmeros detalhes para serem avaliados. Resultado: distribuir os objetos em dois discos ao invés de um não significa ter o dobro de velocidade.

O uso massivo do RAID começou a trazer uma nova abordagem. Quanto mais discos você colocar no RAID, mais rápido o acesso será como um todo. Então se você dobra o número de discos no RAID, você dobra a velocidade em todo o acesso aos objetos armazenados naqueles discos. Então a questão fundamental é em quantos discos a informação vai ser dividida para gravarmos ela simultaneamente. É aqui que começa a guerra dos tipos de RAID a se utilizar.

Com o RAID 0, você tem o aproveitamento máximo dos discos. A implementação é simples e o ganho de desempenho é máximo também: o ganho de desempenho é exatamente igual ao número de discos utilizados. 2 discos, 2 vezes mais rápido. 5 discos, 5 vezes mais rápido, 100 discos, 100 vezes mais rápido. Não é maravilhoso? Simples, barato e eficiente. Só tem um problema. Se um único disco falhar... todos os seus dados em todos os discos do RAID vão para o espaço, junto com o seu emprego. Resultado, o RAID 0 só pode ser utilizado em um servidor de banco de dados em uma situação: dados temporários cuja perda não causa perda de dados nem indisponibilidade no sistema. Há um bom exemplo disso. Se você utiliza sistemas que fazem consultas muito complexas, numa base grande (vejamos, pelo menos uns 500GB é algo de tamanho considerável) como num data warehouse, você terá um volume de dados temporários considerável. Neste caso, vale a pena ter um RAID separado só para os tablespaces temporários. O PostgreSQL 8.3 traz a capacidade de indicar um lugar específico para os dados temporários. Aqui é o lugar para isso.

Então vem o nosso amigo RAID 5, que é muito rápido para leitura, mas é considerado lento para gravação. Se você tem um grande volume de dados estáticos, com muita leitura e pouca gravação, o RAID 5 pode ser para você. É verdade que o RAID 5 tem desempenho inferior em gravação. Mas se você colocar um volume grande de discos, com pelo menos 5 discos, este custo passa a ser compensado pelo aumento no número de discos. Existem também implementações do RAID 5 em hardwares proprietários que não apresentam uma penalização de gravação tão alta quanto se divulga por aí. É claro que isto depende do uso de uma ótima controladora de discos. Mas o fato é que o RAID 5 tem má fama devido ao seu problema com a segurança. Enquanto no RAID 0 você não

tem segurança nenhuma, o RAID 5 permite que você perca até um disco. O problema é que se você comprar vários discos num mesmo lote, existe uma grande chance deles apresentarem defeito em no mesmo período. Se você observar dezenas de lâmpadas trocadas ao mesmo tempo, verá que elas começam a queimar na mesma época. Se isto ocorrer com seu RAID 5, você terá problemas. Então, se você se preocupa com segurança, não use RAID 5. No RAID 5 você precisa ter no mínimo 3 discos, mas você não deveria jamais montar um RAID 5 com 3 discos. Por outro lado, se você se preocupa com a segurança, colocar um número muito grande de discos aumenta a chance de haver uma falha em mais de um disco. Outro detalhe bizarro é que se ocorrer uma falha e você tiver um hot spare, este entrará em operação e começará a remontar todo o esquema de redundância novamente. Essa operação de reconstrução da paridade do RAID 5 é pesada e lenta, então se os seus discos forem do mesmo lote, a chance de um segundo disco quebrar durante a reconstrução é grande. Se isso ocorrer, você perderá todos os seus dados. Se você se preocupa com desempenho só use RAID 5 com pelo menos 5 discos. Se você se preocupa com a segurança, use um hot spare e também não aumente muito o número de discos.

Então porquê todos usam tanto o RAID 5? É simples... ele é muito mais barato que o RAID 1. Destes discos, o espaço total de armazenamento será equivalente ao espaço de todos os discos juntos menos 1. Então se você tem 10 discos de 300 GB, o espaço útil é de 3TB - 300GB = 2.7GB. Nada mau, não? Mas veja bem... a quantidade de discos tem influência enorme não apenas no custo, mas na segurança e no desempenho também. Então onde posso usar o RAID 5? Se você se preocupa exclusivamente com o custo, o RAID 5 é uma solução barata em termos de capacidade de armazenamento, combinando uma segurança mínima que diminui conforme o número de discos aumenta. Se você pensa em desempenho e tem dados atualizados com pouca freqüência, como em dados históricos, o RAID 5 é uma boa solução. Mas se você se preocupa com segurança, evite o RAID 5 e só use para armazenar dados não críticos.

Existe também o RAID 6 que começou a ser mais utilizado por aí. Ele é muito semelhante ao RAID 5, mas permite a perda de até 2 discos sem interrupção no funcionamento. É mais seguro sem ser muito mais caro. Você provavelmente só vai encontrar o RAID 6 em controladoras mais sofisticadas e storages externos. Num RAID 6, com 10 discos de 300GB o espaço útil é de 2.4TB. Para um número pequeno de discos (o mínimo são 4 discos, mas você deveria pensar e começar com 6) o uso de um disco a mais para a paridade é significativo, mas para um número grande isto se torna uma questão menor. Em termos de segurança, poder perder 2 discos é muito interessante. O RAID 6 não sofre tanto com o problema da reconstrução da paridade como no RAID 5, o que aumenta bem a sua segurança. Em termos de desempenho ele é semelhante ao RAID 6.

O nosso amigo RAID 1 é o mais simples e o mais caro tipo de RAID. O ganho de desempenho dele é nulo na gravação e mas dobra a velocidade na leitura. Além disso a capacidade total de armazenamento é metade da soma dos discos. O RAID 1 é o simples espelhamento dos discos. Vejamos como as coisas ficam. Se você tem dois discos de

300GB em RAID 1, o seu espaço útil é de 300GB. Simples assim. Se você tem apenas 2 discos e não quer se arriscar com o RAID 0, o RAID 1 é sua única opção.

O RAID que fez as pessoas largarem mão de distribuir os dados em diferentes discos isolados (sem RAID), foi o RAID 10, ou RAID 1 + 0. O RAID 10 combina o aumento de segurança do RAID 1 com o aumento de desempenho do RAID 0. Só tem um problema, assim como o RAID 1, ele precisa de 50% dos discos para o espelhamento, o que encarece a solução. Outro detalhe, é que você começa a brincar de RAID 10 a partir de 4 discos e daí para frente sempre em números pares como 4, 6, 8, 10, 12, etc. Com dois discos você tem apenas o RAID 1. Assim, o único problema do RAID 10 é o custo.

Imagine que com 4 discos de 300GB, você só aproveitará 600GB. Com 10 discos de 300GB, só aproveitará 1,5TB. É bem possível também que com 10 discos em RAID 5 ou 6 você tenha um desempenho de gravação semelhante ou superior e um desempenho em leitura muito maior. Então a questão do RAID 10 é realmente a segurança.

Então fica a questão: quando eu devo usar o RAID? Resposta: SEMPRE, escolhendo o RAID 10, 1, 5 ou 6 dependendo das suas prioridades e RAID 0 para casos muito especiais como um esquema complementar. A cultura que está se formando em torno dos DBAs hoje é a de usar RAID 1 ou 10 para tudo, uma vez que a segurança raramente é uma questão menor e o custo \$/GB estar caindo, particularmente com os discos SAS em substituição aos SCSI.

Discos

Falando em tipos de discos, outra coisa que você deve lembrar é que existem no mercado 3 tipos de interfaces para discos: fibre channel, SAS e SCSI. Não existe SATA, esqueça que eles existem mesmo em RAID. Não importa o que você leu no blog do beltrinho, SATA é muito mais lento e muito menos confiável. Mesmo os discos SAS estão ainda sob vigilância, uma vez que sempre que o preço dos discos cai muito, a desconfiança aumenta. De qualquer forma, os discos SAS devem dominar o mercado rapidamente. Os discos fibre channel são usados apenas em storages externos de alto desempenho. É comum ver storages que suportem discos SCSI, fibre channel, SAS e SATA. Mesmo que o seu storage seja caríssimo, o fato de ele suportar os discos SATA não significa que eles sejam adequados para bancos de dados. Pode ser que para servidores de arquivo o seu uso junto com RAID seja aceitável, para bancos de dados não. Você realmente vai ter que comprar uma controladora descente e discos dedicados a servidores e não a desktops.

Sobre o tamanho dos discos, há outro detalhe importante. Pelo menos na tradição dos discos SCSI, os discos de maior capacidade são os que têm o custo de \$/GB melhor e o pior desempenho. Não sei se a mesma tendência se repetirá com os discos SAS, mas é bom ficar de olho. Mesmo porquê, se você precisa de muito espaço em disco, é claro que a melhor solução é ter muitos discos pequenos e não poucos discos enormes. Aumentar o número de discos é sinônimo de aumentar o desempenho. Não existem servidores de bancos de dados sérios com 1 disco. Comece com 2 discos e aumente este número aos pares. Isto o deixa na posição de começar com um RAID 1 que é uma solução aceitável

para um servidor pequeno e crescer para 4 ou 6 no futuro com um RAID 10. É claro que se você não utiliza um storage externo, o gabinete do servidor vai limitá-lo seriamente. Então evite os servidores 1U (estou falando da altura do servidor no rack, é claro) que só comportam cerca de 2 discos e prefira os servidores com 2U que comportam em geral até 6 discos ou 4U que comportam cerca de 15 discos. Existem discos novos com tamanho de 2,5 polegadas ao invés das tradicionais 3,5 polegadas. Estes devem possibilitar um aumento no número de discos num mesmo gabinete além de diminuir o consumo de energia. Ainda é muito cedo para fazer uma comparação séria.

A estatística que realmente interessa para o desempenho do banco de dados é o número de operações por segundo, o IOPS e a taxa de transferência sustentada. O IOPS depende não apenas dos discos, mas da controladora e do SO utilizado. Você pode verificar o IOPS do seu servidor no Linux através do iostat. A coluna “tps” do relatório do iostat é equivalente ao IOPS do seu disco. A taxa de transferência sustentada é a quantidade de bytes que o disco consegue transferir por um longo período. Os fabricantes costumam publicar nas suas especificações a taxa de transferência e a taxa de transferência sustentada. Você deve se preocupar apenas com a segunda. É aqui que os HDs SATA perdem e muito, pois o protocolo utilizado pelo HD SATA não consegue manter uma taxa de transferência razoável. Já os discos Fibre Channel, SCSI e SAS (lembre-se que o SAS nada mais é do que o protocolo SCSI com uma interface serial).

Ao comparar os últimos lançamentos dos discos da Seagate por exemplo, vemos o disco [Savvio](#) de 2.5 polegadas, 15Krpm com interface SAS e capacidade de 36GB e 72GB. Já em 3.5 polegadas temos o [Cheetah](#) um disco de 15,6Krpm com interface SAS ou Fibre Channel e capacidade de 146GB, 300GB ou 450GB. Vejamos alguns detalhes das especificações:

Especificação	2,5" SAS 73GB	3,5" SAS 146GB	3,5" FC 146GB
Taxa de transferência externa (MB/s)	300	300	400
Taxa de transferência sustentada interna(MB/s)	79 a 112	110 a 178	110 a 178
Latência média (ms)	2	2	2
Tempo de busca médio Leitura/Gravação (ms)	2.9/3.3	3.4/3.9	3.4/3.9
Tempo de busca trilha a trilha Leitura/Gravação (ms)	0,2/0,4	0,2/0,4	0,2/0,4
Potência media (W)	7,9	14,4	15,0

Notamos que:

- A taxa de transferência externa do Fibre Channel é maior que o SAS;
- A taxa de transferência sustentada é menor nos discos de 2.5 polegadas. Vale a pena lembrar que quanto maior o diâmetro do disco, maior a quantidade de setores nas trilhas externas do disco. Isto faz com que mantendo o número de rotações do disco, os discos com maior diâmetro conseguem transferir um número maior de dados;
- O tempo de busca que mede a velocidade com a qual o cabeçote se move (de trilha para trilha) é igual, mas como o disco de 3,5 polegadas tem que se movimentar por um disco com maior diâmetro, o tempo médio de busca dele é maior.
- O consumo de energia dos discos de 2,5 é substancialmente menor, devido ao menor peso dos discos;

O custo de cada um destes discos, está entre 400 e 600 dólares. Acredite ou não, este é um preço muito razoável. Não faz nem 2 anos e tive que comprar discos de 146GB em Fibre Channel e 10Krpm por nada menos que R\$ 15.000,00 cada um. Então, considerando que estamos olhando para um hardware de ponta, podemos esperar que em 2 anos estes discos se tornar opções standard.

Controladoras de discos

Bom, para encerrar a questão do hardware, você tem que pensar com cuidado na sua controladora de discos. Assim como quem vê processador não vê chipset, quem vê discos, não vê controladora de discos. Não adianda ter vários discos de 15Krpm SAS e uma controladora standard. Uma boa controladora faz toda a diferença. Há quem diga que com uma controladora ruim, vale mais a pena utilizar RAID por software do que aproveitar o RAID nativo da controladora. Além disso, a quantidade de discos que a controladora suporta, as modalidades de RAID, a quantidade de buffer cache disponível, a presença de bateria, tudo isso influencia muito. Uma controladora mais simples pode ter seu preço em torno dos 250 dólares e uma mais poderosa pode ultrapassar os 1000. Mas você deve reservar uma parte do orçamento para alguns acessórios como cabos, pentes de memória e baterias externas que podem fazer você chegar facilmente aos 2 mil dólares.

Apesar de haverem controladoras que suportam até 128 discos SAS no mercado, pode ser que alocar esta quantidade de discos não seja tão simples. Se você chegar neste ponto, pode ser interessante pensar num storage externo. Há diversos modelos de storage, escaláveis para quantidades consideráveis de discos. Você pode começar com uma gaveta contendo uma dezena de discos e ir adicionando novas gavetas e até mesmo racks chegando aos milhares de discos. Há outras vantagens consideráveis no uso de um storage externo. Você pode compartilhar o mesmo storage com mais de um servidor e montar uma SAN (Storage Area Network), o que pode ser muito interessante, para migrar dados de produção para teste, usar replicação ou até técnicas de cluster. A performance também é um fator fundamental. Eles chegam a ter vários GB de cache e boas baterias

internas, o que lhe permite utilizar o cache de gravação com segurança. Além disso, os storages costumam vender alguns softwares (proprietários até a alma) que ativam capacidades especiais do hardware como tuning de discos, monitoramento avançado e o meu preferido: snapshots! O custos de soluções de médio porte de storage despencaram. Acredite ou não, já é possível contar com uma estrutura de storage completa com menos de R\$ 50 mil. Vale a pena ressaltar, que para bancos de dados, os storages iSCSI embora mais baratos, não são recomendados. Acha muito? Você não tem idéia de o quanto os preços já caíram. Não faz muito tempo que um storage básico com Fibre Channel e 1TB não saia por menos de uns R\$200 mil. É claro que esta conta pode chegar na casa dos milhões muito rápido. É só colocar milhares de discos na conta. Você acha que utilizar milhares de discos é um exagero? Convido você a olhar os testes de performance do [TPC](#)... o teste mais simples, não usam menos de 100 discos. Realmente vale a pena olhar os testes, pois além do resultado de performance, eles detalham todos os custos de hardware e software, incluindo suporte para 3 anos.

Tipos de arquivos

Agora vem uma parte realmente importante que é entender os diferentes tipos de informação que serão gravados no seu servidor. Mesmo se você tem um pequeno servidor de banco de dados com apenas dois discos (se você só tem 1, bem... sinto muito), isso é fundamental antes de sair particionando os discos.

Vejamos como podemos classificá-los:

- Sistema Operacional

Estamos imaginando obviamente que você está criando um servidor dedicado. Servidores de bancos de dados são serviços que gostam de exclusividade, principalmente no acesso a disco. Então, se for inevitável ter que colocar mais um serviço no mesmo servidor, que este não seja um servidor de e-mail ou arquivos. Nada que vá disputar o acesso aos discos com o banco de dados. De toda a forma a idéia é que o SO não costuma ocupar muito espaço, não é um gargalo de desempenho e não compõe uma parte crítica dos dados uma vez que ele pode sempre ser reinstalado. No perder o SO, vai lhe causar um bom tempo com o servidor fora do ar até que ele seja reinstalado, portanto algum tipo de proteção deve existir, como um RAID 1, 5, 6 ou 10. Em geral, uma partição com alguns GB devem ser suficientes para todo o SO, e executáveis do SGDB. Se estiver usando Linux, não esqueça de guardar uma pequena partição para o kernel (algo como 100 ou 200 MB).

- Swap

O Linux e outros SOs utilizam uma parte do disco para servir de memória virtual paginada em caso de falta de memória. Em tese isto nunca deve acontecer e você deve ter memória suficiente para evitar este tipo de situação na maior parte do tempo. No entanto, se acontecer o Swap deve estar lá para evitar que todos os processos se percam repentinamente. O Swap deve sempre possuir sua própria partição que em geral possui o

mesmo tamanho da sua memória RAM. Se você tiver mais de 4GB, pode pensar em usar um pouco menos de isso, chegando a 50% da RAM a partir dos 8GB para cima.

- Logs

O seu servidor gera uma série de logs sobre o que acontece no banco de dados e no servidor. A quantidade de logs gerados variam muito conforme a configuração do banco de dados e demais serviços. Em geral a configuração standard do SO é suficiente para a maioria dos casos enquanto as [configurações do banco de dados](#) devem ser estudadas caso a caso. Você deve determinar onde colocar seus logs, o que logar, quando logar, etc. Uma análise de performance num ambiente de produção pode gerar alguns GB de logs em poucos minutos. Guardar uma partição para estes logs é importante. Os logs devem ficar sob controle e não devem ocupar um espaço maior do que o previsto. É comum executar faxinas periódicas nos logs. Não esqueça de [configurar os logs do PostgreSQL](#).

- Tablespace padrão.

O tablespace padrão é aquele utilizado para guardar o dicionário de dados. Este tablespace é crítico. Ele ocupa pouco espaço pois só consiste nos metadados a respeito dos demais objetos do banco. Se você perder o dicionário de dados, não conseguirá acessar mais nenhum objeto do banco, tornando-o completamente inútil. Em termos de segurança, este é um tablespace que deve ser muito bem protegido. Se você tiver muita memória, provavelmente o dicionário do sistema deve estar quase sempre no buffer tornando o desempenho uma questão secundária. O problema que se encontra com freqüência é que as pessoas não criam novos tablespaces para uso dos objetos normais das aplicações. Misturar ambos não é uma boa idéia. O espaço ocupado pelo tablespace default costuma ser mínimo, se ele realmente contiver apenas o dicionário de dados. A não ser que você tenha uma quantidade muito grande de objetos, particularmente visões e funções, destinar 1GB para este tablespace costuma ser mais que o suficiente.

- Tablespace temporário

O tablespace temporário não tem impacto em termos de segurança para as informações. As informações lá armazenadas são apenas uma área de trabalho para operações intermediárias e tabelas temporárias. Em tese, estas informações jamais precisariam ser guardadas em disco, porém operações pesadas como a ordenação de uma tabela muito grande podem exigir muito deste tablespace. Particularmente os DataWarehouse, Data Marts e relatórios pesados exigem um volume considerável em disco. O desempenho das consultas pesadas podem fazer deste tablespace algo crítico também. Vale a pena conhecer o perfil de carga da sua aplicação para saber se vale a pena investir em uma abordagem específica para este tablespace. e tem grande impácto no desempenho. Por padrão o tablespace temporário fica junto com o dicionário de dados, mas você pode setar a partir da versão 8.3 do PostgreSQL o parâmetro [temp_tablespaces](#) para escolher um local diferente.

- Outros tablespaces

As informações do banco de dados ficam armazenadas ao fim e ao cabo nos seus arquivos de dados que devem possuir seus próprios [tablespaces](#). Um critério para dividir os tablespaces é usar um par de tablespaces para tabelas e outro para índices em cada aplicações. Apesar de não ser mais comum dividir índices e tabelas em discos distintos, isto pode lhe ajudar muito em termos administrativos, resolução e recuperação de desastres. No entanto, alguns objetos podem ser muito grandes e exigir particionamento. Os benefícios do particionamento de tabelas e índices se fazem mais presentes quando colocamos cada partição em um tablespace distinto que devem estar em discos distintos por sua vez. Outra questão é o tipo de uso do tablespace de acordo com o perfil de operações SQL executadas sobre seus objetos. Vou deixar aqui classificados 5 tipos básicos:

Tablespace para OLTP: é o tablespace mais crítico em termos de desempenho e segurança. Isto ocorre pois ele sofre um grande volume de pequenas gravações concorrentes. As gravações concorrentes são o verdadeiro inferno em termos de desempenho. Por outro lado, exigem técnicas mais sofisticadas para evitar a perda de dados. Tablespaces de aplicações com forte carga OLTP devem estar em discos distintos dos demais pois precisam de atenção especial.

Tablespace para BI: as aplicações de BI são completamente diferentes. Elas sofrem em geral grandes cargas periódicas de dados e não sofrem atualizações constantes. Ao contrário da carga em OLTP, em BI temos poucos usuários simultâneos e quase nenhuma operação de gravação. O grande desafio é conseguir suportar consultas complexas que consultam um grande volume de dados. Aqui o desempenho também é crucial, pois uma única consulta pode facilmente levar dias para se concluir. A segurança não é em geral um ponto crucial, visto que os dados em geral podem ser reconstruídos a partir de cargas externas.

Tablespace para Web: as aplicações web tradicionais são aquelas que possuem um volume absurdo de conexões simultâneas em operações predominantemente de pequenas leituras. Devido ao grande volume de acessos que um site na web pode ter, o desempenho é novamente um fator crucial.

Tablespace Histórico: algumas aplicações ou parte delas, carregam um volume enorme de informações históricas e quase sempre estáticas. Estes dados precisam estar on-line para fins de auditoria ou consultas esporádicas. Este é um raro caso onde o desempenho e a segurança não são tão importantes. Aqui é um dos locais onde, tomando os devidos cuidados, podemos economizar um pouco nos discos.

- Logs de transação

Os logs de transação, no PostgreSQL, são conhecidos como [WAL](#) ou Write Ahead Log. Estes arquivos são arquivos de tamanho fixo utilizados para assegurar a segurança dos dados em caso de queda de energia. Sem eles a segurança do banco de dados seria

tragicamente comprometida. Os logs são então alvo de enorme preocupação em termos de segurança contra a perda de dados e ao mesmo tempo tem uma influência enorme no desempenho. Para se ter uma idéia de como o WAL é importante, em [um artigo](#) do Indiano Jignesh Hah (veja os comentários também...) ele demonstra que numa aplicação OLTP pesada com cerca de 3000 transações por segundo, seriam necessários 25 discos para atingir o IOPS necessário para não ter uma grande degradação de performance. Se adicionarmos o espelhamento que é praticamente obrigatório nestes casos estaremos falando de 50 discos só para o WALL!

- Arquivamento dos logs de transação

O [archive](#) são cópias dos logs de transação que são reciclados. Fazer um backup do WALL é considerado obrigatório em ambientes de produção. Sem ele, qualquer arquivo de dados corrompido implicaria obrigatoriamente em perda de dados. O archive junto com o backup on-line dos tablespaces permitem o uso de uma técnica avançadas para recuperações de desastres conhecida como “Point In Time Recovery”. Enquanto o WALL precisa de um pouco espaço de armazenamento, é de praxe deixar em disco algo em torno de uma semana de backups do WAL, o que pode significar vários GB em disco de acordo com o volume de atualizações que o banco sofre.

- [Backup físico](#)

A não ser em caso de bases pequenas com alguns poucos GB, o backup físico é praticamente obrigatório. Isto significa que você tem que ter espaço em disco (local ou remoto) para armazenar uma cópia de todos os seus datafiles. As exigências de desempenho aqui vão depender da sua janela de backup. Se você estiver na graça de possuir um storage com software de snapshot, então sua vida se tornará muito mais simples aqui. Lembre-se que os backups em disco não devem jamais substituir os backups em fita. Em último caso, fique apenas com os backups em fita. Mas ter uma cópia do último backup físico em disco, pode acelerar muito o processo de recuperação de desastres.

- [Backup lógico](#)

O backup lógico não é uma estratégia recomendada para bases medias e grandes. Mesmo assim, você deve pensar em reservar um espaço em disco para os backups lógicos. O motivo é que a movimentação de dados entre as bases de produção, homologação e testes costumam ser frequentes. Seria decepcionante perceber que você não tem espaço em disco para isso. Aqui, o desempenho e a segurança não costumam ser algo fundamental.

Particionamento

É possível utilizar apenas uma partição para todo o servidor? Sim. Isto é bom? Não. Existem 3 bons motivos para você separar as coisas:

1. Segurança contra perda de dados: se você tiver os dados do seu servidor em um grupo de discos, o WAL e seus arquivamentos em outro e possuir um backup físico em algum lugar, a perda completa do grupo de discos onde os tablespaces estão não implicarão em perda de dados. Separar os tablespaces do WAL e seus archives e possuir um backup físico em algum outro lugar (em outro servidor, outro disco local, outra fita, etc) são uma política muito segura para se evitar a perda de dados e consequentemente a perda de emprego...
2. Segurança contra indisponibilidade: se você tiver apenas uma partição e um erro qualquer acontecer no servidor ou mesmo numa aplicação que acessa o banco de dados, ele pode começar a exigir repentinamente um montante enorme de espaço em disco até que ele fique completamente lotado. Se você tiver uma única partição no seu servidor, você terá não apenas o seu banco de dados travado, como o seu SO também. Separando os tipos de arquivos em partições distintas, você limita drasticamente o impacto deste tipo de situação;
3. Administração: Fica mais fácil detectar áreas que estão crescendo demais se elas estiverem guardadas em caixas separadas. Dividir os tipos de arquivos em diferentes partições permite que com um único comando no SO (um 'df' no caso do Linux) seja possível verificar como andam se comportando diferentes tipos de arquivos;
4. Desempenho: Se você separar diferentes tipos de arquivos em diferentes partições, poderá utilizar diferentes sistemas de arquivos em cada um deles. Além disso, discos, controladores e sistemas de arquivos possuem diferentes configurações que podem ajudar a aumentar o desempenho em algumas áreas críticas.

Ter tipos de arquivos diferentes em partições diferentes, nem sempre significa utilizar discos ou grupos de discos em RAID distintos. Se você tiver apenas 2 discos, não será possível fazer muita coisa. Ainda assim, separar algumas coisas como: SO, logs, tablespace padrão outros tablespaces, WAL, archives e backups em partições distintas, costuma fazer muito sentido. Um problema com as partições é que elas tem tamanho fixo, obrigando você a prever quanto espaço será necessário para cada partição. Em geral, prever isso costuma ser um dever do DBA, mas em bases novas isto pode ser mais difícil. Em todo caso, os storages e controladoras modernas permitem o uso de LUNs que mudam de tamanho dinamicamente. Se você não tiver acesso a este tipo de tecnologia, pode usar também o [LVM](#) que funciona muito bem, apesar que causar uma pouco de perda de desempenho no acesso a disco.

Como distribuir as partições nos discos existentes

Vamos imaginar aqui diferentes números de discos no servidor e possíveis configurações que poderiam ser utilizadas. Aqui estamos fazendo um chute grosso, você pode mudar um pouco as configurações dos discos e partições de acordo com algumas das dicas acima entre outras coisas, inclusive por exigências contratuais ou SLA.

- Um disco

Bom, esta é a pior situação que você pode encontrar pois não é possível montar um RAID. A sua segurança contra falhas de discos é nula e o seu desempenho também será pobre. Neste caso, você deve pelo menos adorar uma política de backups freqüentes para minimizar a possível perda de dados a qual você está sujeito. Tenha certeza de alertar sobre os riscos por escrito aos seus superiores.

Mesmo tendo apenas um disco, você deve ter ao menos ter as seguintes partições:

/boot para o kernel do linux (100MB a 200MB)

/ para o restante do SO (algo entre 5 e 10GB)

/var/log ou outro local destinado aos logs (algo entre 1 e 10 GB costumam ser valores razoáveis)

swap (o mesmo tamanho do tamanho da memória RAM)

/var/lib ou /postgres outro local destinado aos dados (cerca da metade do espaço em disco restante)

/backup ou outro local para os backups lógicos e/ou físicos (o restante do espaço em disco)

- Dois discos

A não ser que você tenha dois discos de tamanhos diferentes, use RAID 1 e mantenha o esquema de particionamento descrito acima;

- Três discos

Se você se preocupa com disponibilidade, use um RAID 1 com dois discos e deixe o disco restante como hot spare. Se você está precisando desesperadamente de mais espaço em disco, você pode montar um RAID 1 com dois discos e utilizar o 3º disco ser RAID para guardar os seus backups

- Quatro discos

Se a sua prioridade for segurança, você pode montar dois RAID 1. No primeiro RAID 1 coloque o seu SO, os logs, o WALL, o archive e os backups físicos. No segundo RAID 1 coloque os tablespaces e os backups lógicos. Você também pode montar um RAID 1 com 2 discos, mais um disco de hot spare e utilizar o 4º disco para guardar seus backups.

Se a sua prioridade for desempenho, coloque tudo em um RAID 10 com os 4 discos.

- Cinco discos

Use dois RAIDs 1 ou um RAID 10 e deixe um disco de hot spare.

- Seis discos

Use a mesma configuração de 5 discos, mas reserve um disco só para backup lógico e/ou físico

- Sete discos

Um disco ficará para hot spare, os 6 discos sobrando podem ser divididos em um RAID 1 com dois discos e um RAID 10 com 4 discos (opção recomendada para aumentar a segurança) ou montar um RAID 10 com 6 discos (opção recomendada para privilegiar o desempenho).

Você verá que com um maior número de discos você pode separar mais as coisas em pelo menos 3 grupos:

- Logs de transação e archives
- Datafiles
- Backups

Se você tiver algo como 15 ou mais discos, poderá começar a separar mais as coisas, optar ocasionalmente por um RAID 5 ou 6 para arquivos menos críticos e coisas do tipo.

Obs.: Este texto não é de minha autoria, mas como não encontro mais o mesmo na internet, o mantendo aqui.

Discos no PostgreSQL

Resumo do artigo PostgreSQL, discos & Cia do Fábio Telles.

Tipo de RAID a usar

Quanto mais discos colocar no RAID, mais rápido será o acesso para todas as operações.

RAID 0

Aproveitamento máximo dos discos. O ganho de desempenho é exatamente igual ao número de discos utilizados. Para 2 discos, 2 vezes mais rápido; 5 discos, 5 vezes mais rápido e assim por diante. Só um problema, se um único disco falhar todos os seus dados em todos os discos se perderão.

RAID 5

Se você tem um grande volume de dados estáticos (muita leitura e pouca gravação) o RAID 5 pode ser indicado para você. Enquanto que no RAID 0 não temos nenhuma segurança, no RAID 5 podemos perder até um disco inteiro e manter a segurança.

Se você se preocupa com segurança use RAID 5.

Caso se preocupe com desempenho só use RAID 5 com pelo menos 5 discos.

RAID 1

É o mais simples e o mais caro tipo de RAID.

RAID 10

Seu único problema é o custo. Com 4 discos de 300 (1200) você só aproveita 600GB.

A recomendação é sempre usar RAID, escolhendo de acordo com sua estrutura e condições o 10, 1, 5 ou 6. Existe uma tendência atual de usar sempre RAID 1 ou 10.

Discos

Existem no mercado 3 tipos de interfaces para discos: fiber channel, SAS e SCSI. Se estiver interessado em um desempenho e segurança mínimos esqueça a SATA. Os de fiber channel são usados apenas em storages externos de alto desempenho.

A recomendação é que se compre uma controladora decente e discos dedicados a servidores e não a desktops.

Pode começar com 2 discos e ir aumentando aos pares. Isso possibilita RAID 1 (aceitável) e ir aumentando para 4 ou 6 com RAID 10.

Ao comprar servidores em baias evite os 1U, que somente comporta 2 discos. Prefira os 2U, que comportam 6 discos ou superior.

Uma boa controladora faz toda a diferença.

Para medir estatísticas de discos instale:

sudo apt install sysstat

Execute
iostat

```
ribafs@ribalinux ~ $ iostat
Linux 4.4.0-21-generic (ribalinux) 01/08/2016 _x86_64_ (4 CPU)
```

```
avg-cpu: %user %nice %system %iowait %steal %idle
      5,49   0,05  1,40  7,03  0,00  86,03
```

Device:	tps	kB_read/s	kB_wrtn/s	kB_read	kB_wrtn
---------	-----	-----------	-----------	---------	---------

sda	18,38	297,77	1923,27	1722855	11127677
sdf	0,02	0,51	0,00	2932	16

A coluna tps mostra a velocidade.

Storages Externos

Caso tenha uma grande necessidade de espaço e segurança deve pensar em adquirir um storage externo de alto desempenho. Este storage pode ser compartilhado com vários servidores. Lembrar que eles geralmente são vendidos com algum software de backup e com outros bons recursos.

Os mais indicados são os com fiber channel.

Vacuum - <https://www.techonthenet.com/postgresql/vacuum.php>

Autovacuum - <https://www.techonthenet.com/postgresql/autovacuum.php>

51 - Monitorando as Atividades do Servidor do PostgreSQL

- 1 – Habilitando e monitorando o autovacuum
- 2 – Entendendo as estatísticas do PostgreSQL
- 3 – Rotina de reindexação
- 4 – Monitorando o espaço em disco e tamanho de objetos
- 5 – Monitorando o arquivo de registros (logs)

Estão disponíveis várias **ferramentas para monitorar** a atividade do banco de dados e analisar o desempenho. Não se deve desprezar os programas regulares de monitoramento do Unix, como **ps**, **top**, **iostat** e **vmstat**. Também, uma vez que tenha sido identificado um comando com baixo desempenho, podem ser necessárias outras investigações utilizando o comando **EXPLAIN** do PostgreSQL.

Das ferramentas citadas, apenas **iostat** não acompanha o Ubuntu. Podemos instalar com:

```
sudo apt-get install sysstat
```

Para monitorar o PostgreSQL no Windows, como também no Linux, usar o PGAdmin, em Ferramentas – Status do Servidor.

1 – Habilitando e monitorando o autovacuum

A rotina de manutenção vacuum (limpeza do disco rígido) é algo tão importante, que passou a vir habilitado por default para rodar automaticamente na versão 8.3.

Na versão 8.2 para automatizar o vacuum habilitar os seguintes parâmetros no postgresql.conf:

```
autovacuum = on
stats_start_collector = on
stats_row_level = on
```

Na versão 8.3 e na versão 9.5:

```
autovacuum = on
track_counts = on
```

Monitorando o autovacuum

No Linux, uma forma simples de monitorar o autovacuum é executando o comando:

```
ps ax | grep postgres (Linux XUbuntu 7.10 com PostgreSQL 8.3)
```

Com isso serão mostrados:

6881 ?	S	0:00 /usr/local/pgsql/bin/postmaster -D /var/lib/pgsql/data
6928 ?	Ss	0:00 postgres: writer process
6929 ?	Ss	0:00 postgres: wal writer process
6930 ?	Ss	0:00 postgres: autovacuum launcher process
6931 ?	Ss	0:00 postgres: stats collector process

Através dos logs (ao final) podemos ter acesso a mais informações sobre o autovacuum.

No Windows usar o PGAdmin.

2 – Entendendo as estatísticas do PostgreSQL

O coletor de estatísticas

O coletor de estatísticas do PostgreSQL é um subsistema de apoio a coleta e relatório de **informações sobre as atividades do servidor**. Atualmente, o coletor pode contar acessos a tabelas e índices em termos de blocos de disco e linhas individuais.

Configuração da coleta de estatísticas

Uma vez que a coleta de estatísticas adiciona alguma sobrecarga à execução das consultas, o sistema pode ser configurado para coletar informações, ou não. Isto é controlado por parâmetros de configuração, normalmente definidos no arquivo **postgresql.conf**

O parâmetro **track_activities** deve ser definido como true para que o coletor de estatísticas seja ativado. Esta é a definição padrão e recomendada, mas pode ser desativada se não houver interesse nas estatísticas e for desejado eliminar até a última gota de sobrecarga (Entretanto, provavelmente o ganho será pequeno e talvez não compense). Deve ser observado que esta opção não pode ser mudada enquanto o servidor está executando.

track_activities (boolean)

Habilita a coleta de informações das consultas em execução atualmente de cada seção, como também o tempo que cada consulta iniciou sua execução. Este parâmetro vem habilitado por default. Somente o superusuário e o usuário dono da sessão percebem essa informação e somente superusuários podem mudar essa configuração.

track_counts (boolean)

Habilita a coleta de informações das atividades dos bancos. Habilitado por default, pois o processo do autovacuum requer este parâmetro. Somente superusuário pode alterar.

update_process_title (boolean)

Habilita a atualização dos títulos dos processos cada vez que uma nova consulta SQL é recebida pelo servidor. O título do processo é tipicamente visualizado pelo comando **ps**, ou se em windows, pelo **Process Explorer**

[http://technet.microsoft.com/pt-br/sysinternals/bb896653\(en-us\).aspx](http://technet.microsoft.com/pt-br/sysinternals/bb896653(en-us).aspx)

Somente superusuário pode alterar esta configuração.

Monitorando Estatísticas

```
log_statement_stats (boolean)
log_parser_stats (boolean)
log_planner_stats (boolean)
log_executor_stats (boolean)
```

Para cada consulta, escreve as estatísticas de desempenho do respectivo módulo para o log (registro) do servidor. Este é um instrumento rústico. `log_statement_stats` reporta todas as declarações de estatística, enquanto os demais reportam as estatísticas por módulo. `log_statement_stats` não pode ser habilitado para nenhum com a opção por módulo.

Todos estes parâmetros vêm desabilitados por default e somente o superusuário pode alterar seus status (através do comando SET).

Mostra o PID de cada consulta e a própria consulta em execução atualmente:

```
SELECT pg_stat_get_backend_pid(s.backendid) AS procpid,
       pg_stat_get_backend_activity(s.backendid) AS current_query
  FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

Alerta: sempre ao trabalhar com estas funcionalidades e ainda outras é útil verificar a versão do PostgreSQL em uso para procurar a respectiva versão de documentação.

Utilização do comando EXPLAIN

EXPLAIN -- mostra o plano interno de execução de uma consulta.

Sintaxe:

`EXPLAIN [ANALYZE] [VERBOSE] consulta`

Este comando mostra o *plano de execução* gerado pelo planejador do PostgreSQL para a consulta fornecida. O plano de execução mostra como as tabelas referenciadas pelo comando serão varridas — por uma varredura seqüencial simples, varredura pelo índice, etc. — e, se forem referenciadas várias tabelas, quais algoritmos de junção serão utilizados para juntar as linhas requisitadas de cada uma das tabelas de entrada.

Usar o comando ANALYZE com o EXPLAIN para que assim o EXPLAIN tenha uma estimativa de custo mais realista.

Do que é mostrado, a parte mais importante é o custo estimado de execução do comando, que é a estimativa feita pelo planejador de quanto tempo vai demorar para executar o comando (medido em unidades de acesso às páginas do disco). Na verdade, são mostrados dois números: **o tempo inicial** para que a primeira linha possa ser retornada, e **o tempo total** para retornar todas as linhas. **Para a maior parte dos comandos o tempo total é o que importa**, mas em contextos como uma sub-seleção no EXISTS, o planejador escolhe o menor tempo inicial em vez do menor tempo total (porque o executor vai parar após ter obtido uma linha). Além disso, se for limitado o número de linhas retornadas usando a cláusula LIMIT, o planejador efetua uma interpolação apropriada entre estes custos para estimar qual é realmente o plano de menor custo.

ANALYZE faz o comando ser realmente executado, e não apenas planejado. O tempo total decorrido gasto em cada nó do plano (em milissegundos) e o número total de linhas realmente retornadas são adicionados ao que é mostrado. Esta opção é útil para ver se as estimativas do planejador estão próximas da realidade.

Se for desejado utilizar EXPLAIN ANALYZE para um comando INSERT, UPDATE, DELETE ou EXECUTE sem deixar o comando afetar os dados, deve ser utilizado o seguinte procedimento (dentro de uma transação):

```
BEGIN;
EXPLAIN ANALYZE INSERT INTO clientes (cod, nome) VALUES (1, 'João');
ROLLBACK;
```

VERBOSE

Mostra a representação interna completa da árvore do plano, em vez de apenas um resumo. Geralmente esta opção é útil apenas para finalidades especiais de depuração.

Consulta - Qualquer comando SELECT, INSERT, UPDATE, DELETE, EXECUTE ou DECLARE, cujo plano de execução se deseja ver.

O PostgreSQL concebe um *plano de consulta* para cada consulta recebida. A escolha do plano correto, correspondendo à estrutura do comando e às propriedades dos dados, é absolutamente crítico para o bom desempenho. Pode ser utilizado o comando [**EXPLAIN**](#) para ver o plano criado pelo sistema para qualquer comando. A leitura do plano é uma arte que merece um estudo aprofundado. Aqui são fornecidas apenas algumas informações básicas.

Os números apresentados atualmente pelo EXPLAIN são:

- O **custo inicial** estimado (O tempo gasto para poder começar a varrer a saída como, por exemplo, o tempo para fazer a classificação em um único nó de classificação).

- O **custo total** estimado (Se todas as linhas fossem buscadas, o que pode não acontecer: uma consulta contendo a cláusula LIMIT pária antes de gastar o custo total, por exemplo).
- **Número de linhas (rows)** de saída estimado para este nó do plano (Novamente, somente se for executado até o fim).
- **Largura média estimada (width)** (em bytes) das linhas de saída deste nó (fase) do plano.

Os custos são medidos em termos de **unidades de páginas de disco** buscadas (O esforço de CPU estimado é convertido em unidades de páginas de disco, utilizando fatores estipulados altamente arbitrários).

Linhos de saída é um pouco enganador, porque não é o número de linhas processadas/varridas pelo comando, geralmente é menos, refletindo a seletividade estimada de todas as condições da cláusula WHERE aplicadas a este nó. **Idealmente, a estimativa de linhas do nível superior estará próxima do número de linhas realmente retornadas, atualizadas ou excluídas pela consulta.**

Antes da realização de testes, devemos executar o comando:

VACUUM ANALYZE; -- Para atualizar as estatísticas internas (do banco atual)

Exemplos:

Testes no banco cep_brasil, após adicionar a chave primária na tabela cep_full.

Observe a saída destes comandos no psql ou através da query tool do PGAdmin.

EXPLAIN SELECT * FROM cep_full; (varredura sequencial, já que não usamos o índice e retornarão todos os registros).

EXPLAIN SELECT logradouro FROM cep_full WHERE cep = '60420440'; (varredura com o índice, pois retornamos apenas parte dos registros com a cláusula where).

Uma forma de ver outros planos é forçar o planejador a não considerar a estratégia que sairia vencedora, ativando e desativando sinalizadores de cada tipo de plano (Esta é uma forma deselegante, mas útil):

```
SET enable_nestloop = off;
EXPLAIN SELECT * FROM cep_full t1, cep_full_index t2 WHERE t1.cep < '60000' AND t1.cep = t2.cep;
```

Examinando Índices

É muito ruim usar conjuntos de registros muito pequenos. Enquanto que selecionando 1000 de um total de 100.000 registros pode ser um candidato para um índice, selecionando 1 de 100 deve dificilmente ser, por causa de que 100 deve provavelmente caber em uma simples página de disco e não existe nenhum plano que pode ser sequencialmente coletado em uma única página de disco.

Também seja cuidadoso quando criando dados de teste, que não estarão disponíveis quando a aplicação estiver em produção. Valores que são muito similares, completamente aleatórios ou inseridos em alguma ordem devem atrapalhar as estatísticas diferente do que teriam uma distribuição real de dados.

Então uma cópia dos registros em produção deve ser mais adequada para testes.

Quando índices não são usados é útil fazer testes para forçar seu uso. Existem parâmetros para run-time que podem desativar vários tipos de planos. Por exemplo, desabilitando o sequential scan (enable_seqscan) com:

Via SQL:

```
set enable_setscan to off;
```

No postgresql.conf:

```
enable_seqscan = off
```

e joins em loops aninhados (enable_nestloop), que são os planos mais básicos, irá forçar o sistema a usar planos diferentes. Caso o sistema continue selecionando um sequential scan ou nested-loop join (join com loops aninhados) então esta é provavelmente a mais fundamental razão de que o índice não é usado; por exemplo, a condição da consulta não corresponde ao índice.

Detalhes em: <http://www.postgresql.org/docs/8.6/interactive/runtime-config-query.html>

Controle do planejador com cláusulas JOIN explícitas

É possível ter algum controle sobre o planejador de comandos utilizando a sintaxe JOIN explícita. Para saber por que isto tem importância, primeiro é necessário ter algum conhecimento.

Em uma consulta de junção simples, como

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

o planejador está livre para fazer a junção das tabelas em qualquer ordem. Por exemplo, pode gerar um plano de comando que faz a junção de A com B utilizando a condição a.id = b.id do WHERE e, depois, fazer a junção de C com a tabela juntada utilizando a outra condição do WHERE. Poderia, também, fazer a junção de B com C e depois juntar A ao resultado. Ou, também, fazer a junção de A com C e depois juntar B, mas isto não seria eficiente, uma vez que deveria ser produzido o produto Cartesiano completo de A com C, porque não existe nenhuma condição aplicável na cláusula WHERE que permita a otimização desta junção (Todas as junções no executor do PostgreSQL acontecem entre duas tabelas de entrada sendo, portanto, necessário construir o resultado a partir de uma ou outra destas formas). **O ponto a ser destacado é que estas diferentes possibilidades de junção produzem resultados semanticamente equivalentes, mas podem ter custos de execução muito diferentes. Portanto, o planejador explora todas as possibilidades tentando encontrar o plano de consulta mais eficiente.**

Mais detalhes em: <http://pgdocptbr.sourceforge.net/pg80/explicit-joins.html> e

<http://pgdocptbr.sourceforge.net/pg80/catalog-pg-statistic.html>
<http://pgdocptbr.sourceforge.net/pg80/catalog-pg-class.html>

Estatísticas utilizadas pelo planejador

Conforme visto na seção anterior, o planejador de comandos precisa estimar o número de linhas buscadas pelo comando para poder fazer boas escolhas dos planos de comando.

Um dos componentes da estatística é o número total de entradas em cada tabela e índice, assim como o número de blocos de disco ocupados por cada tabela e índice. Esta informação é mantida nas colunas `reltuples` e `relpages` da tabela [pg_class](#), podendo ser vista utilizando consultas semelhantes à mostrada abaixo:

```
SELECT relname, relkind, reltuples, relpages FROM pg_class WHERE relname LIKE 'cep%';
```

Em: <http://pgdocptbr.sourceforge.net/pg80/planner-stats.html>

Mais detalhes em: <http://pgdocptbr.sourceforge.net/pg80/monitoring-stats.html>,
<http://www.postgresql.org/docs/8.3/interactive/monitoring-stats.html> e
<http://www.postgresql.org/docs/8.3/interactive/runtime-config-statistics.html#GUC-TRACK-ACTIVITIES>.

Entendendo os Planos internos do PostgreSQL para a Execução das Consultas

Após o servidor do PostgreSQL receber uma consulta de uma aplicação cliente, o texto da consulta é entregue ao parser (analisador). O analisador procura por toda a consulta e checa por erros de sintaxe. Caso a consulta esteja sintaticamente correta o parser deve transformar o texto da consulta em estrutura de árvore analisada. Um parse tree (gráfico em forma de árvore analisada) é uma estrutura de dados que representa o significado de sua consulta em um formulário formal e sem ambiguidades.

Tendo como base esta consulta:

```
SELECT cpf_cliente, valor FROM pedidos WHERE valor > 0 ORDER BY valor;
```

Após o parser ter completado a análise da consulta, o parse tree é entregue ao planejador/otimizador.

O planejador é responsável por percorrer o parse tree e procurar todos os possíveis planos de execução da consulta. O plano deve incluir um **sequential scan** através de toda a tabela e **index scan** se índices úteis forem definidos. Caso a consulta envolva duas ou mais tabelas o planejador pode sugerir um número de diferentes métodos para unir (join) as tabelas. Os planos de execução são desenvolvidos em termos de operadores de consulta. Cada operador de consulta transforma um ou mais conjuntos de entrada (input set) em um intermediário result set.

O operador seq scan, transforma um input set (tabela física) em um result set, filtrando qualquer registro que não satisfaça a constraint da consulta.

O operador sort produz um result set por ordenar o input set de acordo com uma ou mais chaves de ordenação.

Internamente complexas consultas são quebradas em consultas simples.

Quando todos os possíveis planos de execução forem gerados, o otimizador procurará pelo plano mais econômico. Para cada plano é atribuído um custo de execução. As estimativas de custo são medidos em unidades de disco I/O. **Um operador que lê um bloco simples de 8.192 bytes (8KB) de um disco tem o custo de uma unidade. O tempo de CPU também é medido em unidades de disco I/O mas usualmente como uma fração. Por exemplo, o total de tempo de CPU requerido para processar um simples registro é assumido como sendo 1% do simples disco I/O.** Podemos ajustar muitas das estimativas de custo. Cada operador de consulta tem uma diferente estimativa de custo.

Por exemplo, **o custo de um sequential scan de uma tabela completa é computado como o número de 8K blocos na tabela mais alguma sobrecarga de CPU.**

Após escolher o mais econômico (aparentemente) plano de execução o executor de consulta inicia no começo do plano e pede para o operador mais acima para produzir o result set. Cada operador transforma seu input set em um result set. O input set deve provir de outro operador mais abaixo na árvore/tree. Quando o operador mais acima completa sua transformação os results são retornados para a aplicação cliente.

Resumo das Operações Internas na Execução de uma Consulta

Cliente solicita uma consulta ->

Analisador recebe o texto da consulta ->

Analisador procura na consulta por erros de sintaxe ->

Analisador transforma texto da consulta (correta) em estrutura de árvore analisada (parse tree) ->

O planejador percorre o parse tree a procura todos os possíveis planos de execução ->

Internamente complexas consultas são quebradas em consultas simples ->

Para cada plano é atribuído um custo de execução ->

Após todos os possíveis planos de execução serem gerados, o otimizador procurará pelo mais econômico ->

Após escolher o plano mais econômico o executor de consulta vai ao começo do plano e pede para o operador mais acima para produzir o result set ->

Cada operador transforma seu input set em um result set ->

O input set deve provir de outro operador mais abaixo na árvore/tree ->

Quando o operador mais acima completa sua transformação os results são retornados para a aplicação cliente.

O explain é utilizado somente para analisar as consultas com SELECT, INSERT, DELETE, UPDATE e DECLARE... CURSOR.

Exemplo Simples de Uso do Explain:

```
cep_brasil=# explain analyze select * from cep_full_index;
                                         QUERY PLAN
-----
Seq Scan on cep_full_index (cost=0.00..31671.01 rows=633401 width=290)
                                         (actual time=0.032..10863.173 rows=633401 loops=1)
Total runtime: 12298.089 ms
(2 rows)
```

Vamos analisar o resultado.

Realmente o formato do plano de execução pode parecer um pouco misterioso no início. Em cada passo no plano de execução o EXPLAIN mostra as seguintes informações:

- O tipo de operação requerida
- O custo estimado para a execução
- Caso se especifique EXPLAIN ANALYZE, a consulta será de fato executada e mostrado o custo atual.

Em se omitindo ANALYZE a consulta será apenas planejada e não executada e o custo atual não é mostrado.

- Tipo de operação - **Seq Scan**, foi o tipo de operação que o PostgreSQL elegeu para a consulta

- Custo estimado - (**cost=0.00..31671.01 rows=633401 width=290**). É composto de três partes:

cost=0.00..31671.01, que nos informa quanto gastou a consulta. O gasto é medido em termos de leitura de disco. Dois números são mostrados:

0.00... – o primeiro nos informa a velocidade para que um registro do result set seja retornado pela operação.

31671.01 – o segundo, que normalmente é mais importante, representa o gasto para a operação completa.

rows=633401 – Este nos informa quantos registros o PostgreSQL espera que retorne na operação. Não é exatamente o número real de registros.

width=290 – Esta última parte do custo estimado. É uma estimativa do tamanho, em bytes, do registro médio, no result set. Para comprovar experimente mudar o exemplo para a tabela clientes do dba_projeto e veja como não bate.

Lembrando que os operadores de consulta...

Exemplo sem o Analyze:

```
cep_brasil=# explain select * from cep_full_index;
          QUERY PLAN
```

```
Seq Scan on cep_full_index (cost=0.00..31671.01 rows=633401 width=290)
(1 row)
```

Outro Exemplo:

```
cep_brasil=# explain select * from cep_full;
          QUERY PLAN
```

```
Seq Scan on cep_full (cost=0.00..31671.01 rows=633401 width=290)
(1 row)
```

Agora sem utilizar o Explain, ou seja, o atual (real):

```
\timing
select count(*) from cep_full_index;
count
-----
633401
Time: 10574.343 ms
```

Veja que o tempo é similar ao retornado no Exemplo Simples em: (actual time=0.032..10863.173).

Em consultas mais complexas o resultado é dividido em vários passos e os primeiros aparecem abaixo. Os últimos acima, como nestes exemplos:

Uma consulta contendo apenas 23 registros e ordenando por um campo que tem índice:

```
\c dba_projetodba_projeto=# explain analyze select * from clientes order by cpf;
          QUERY PLAN
```

```
Sort (cost=1.75..1.81 rows=23 width=64) (actual time=0.571..0.616 rows=23 loop
s=1)
```

```
  Sort Key: cpf
  Sort Method: quicksort Memory: 20kB
-> Seq Scan on clientes (cost=0.00..1.23 rows=23 width=64) (actual time=0.0
11..0.137 rows=23 loops=1)
Total runtime: 0.865 ms
```

Agora uma tabela com uma grande quantidade de registros (mais de meio milhão), mas sem um índice no campo pelo qual está ordenando:

```
\c cep_brasil
cep_brasil=# explain analyze select * from cep_full order by cep;
               QUERY PLAN
-----
Sort  (cost=352509.44..354092.94 rows=633401 width=290) (actual time=33675.802..53168.029 rows=633401 loops=1)
  Sort Key: cep
  Sort Method: external sort Disk: 189376kB
    -> Seq Scan on cep_full  (cost=0.00..31671.01 rows=633401 width=290) (actual time=0.027..17997.280 rows=633401 loops=1)
Total runtime: 54728.569 ms
```

Em uma tabela com uma grande quantidade de registros (mais de meio milhão), mas agora com um índice no campo pelo qual está ordenando:

```
explain analyze select * from cep_full_index order by cep;
cep_brasil=# explain analyze select * from cep_full_index order by cep;
               QUERY PLAN
-----
```

```
Index Scan using cep_pk on cep_full_index
(cost=0.00..44605.36 rows=633401 width=290)
(actual time=90.159..18145.128 rows=633401 loops=1)
Total runtime: 19699.895 ms
```

Outro exemplo, agora usando a cláusula WHERE na PK:

```
cep_brasil=# explain select * from cep_full_index where cep='60420440';
               QUERY PLAN
```

```
-----
```

```
Index Scan using cep_pk on cep_full_index  (cost=0.00..8.35 rows=1 width=290)
  Index Cond: (cep = '60420440'::bpchar)
```

O Sec Scan ocorrerá primeiro e depois dele o Sort. ***Veja que quando temos um índice E uma grande quantidade de registros no campo pelo qual estamos ordenando, o planejador/otimizador do PostgreSQL decide usar o operador Index Scan.***

Após o Index Scan finalizar a construção do seu result set intermediário ele será então colocado no próximo passo do plano. O passo final deste plano será a operação Sort, que é requerida para satisfazer a cláusula order by. Este operador reordena o result set produzido pelo Sec Scan e retorna o final result set para a aplicação cliente. Observe que quando o Index Scan é eleito, a ordenação é feita pelo próprio índice e não pelo Sort.

Alguns dos Operadores de Consultas

Seq Scan – é o mais básico operador de consultas. O Seq Scan trabalha assim:

- iniciando no começo da tabela e
- varrendo até o final da tabela
- para cada linha da tabela, Seq Scan testa a constraint da consulta (no caso, where). Caso a constraint seja satisfeita, as colunas requeridas são retornadas para o result set.

Registros Mortos

Uma tabela pode ainda conter os registros excluídos, que não mais são visíveis. O Seq Scan não inclui registros mortos no result set, mas ele precisa ler esses registros o que pode ser custoso em uma tabela com pesada atualização.

Seq Scan não lerá a tabela completa antes de retornar o primeiro registro.

Já o operador Sort lerá todos os registros do input set antes de retornar o primeiro registro.

O planejador/otimizador escolherá o Seq Scan quando nenhum índice possa ser usado para satisfazer a consulta ou quando a consulta retorna todos os registros (neste caso o índice não será usado).

Index Scan – funciona percorrendo uma estrutura índice. Caso especifiquemos um valor inicial para uma coluna índice, como por exemplo, WHERE codigo >= 100, o Index Scan deve iniciar pelo valor apropriado. Caso especifiquemos um valor final, como WHERE codigo <= 200, então o Index Scan deverá terminar tão logo encontre um valor maior que 200.

O Index Scan tem duas vantagens sobre o operador Seq Scan: primeiro, o Seq Scan precisa ler todos os registros da tabela e somente pode remover registros do result set avaliando a cláusula WHERE para cada registro. Index Scan não precisa ler todos os registros se indicarmos um valor inicial ou final. Segundo: o Seq Scan retorna os registros na ordem da tabela e não em uma sorteada ordem. O Index Scan deve retornar os registros na ordem do índice.

Nem todos os tipos de índices podem ser utilizados no Index Scan, somente: *B-Tree*, *R-Tree* e *GIST* podem mas o tipo *Hash* não pode.

O planejador/otimizador usa um operador Index Scan quando ele pode reduzir o tamanho do result set percorrendo a faixa de valores do índice ou quando ele pode evitar uma ordenação porcauxa da implícita ordenação oferecida pelo índice.

Sort – impõe uma ordenação no result set. Podemos ajustar o PostgreSQL através do parâmetro de runtima *sort_men*. Caso o result set caiba no *sort_men* * 1024 bytes, o Sort será feito na memória, usando o algoritmo Qsort.

O Sort nunca reduz o tamanho do result set e também não remove registros ou campos.

Sort lerá todos os registros do input set antes de retornar o primeiro registro. Obviamente um Sort pode ser utilizado pela cláusula ORDER BY.

Para os demais operadores veja este capítulo do Livro sugerido abaixo, que inclusive pode ser lido online: <http://www.iphelp.ru/faq/15/ch04lev1sec3.html>

Referências:

<http://www.postgresql.org/docs/8.3/interactive/using-explain.html>
<http://www.postgresql.org/docs/8.3/interactive/sql-explain.html>
<http://www.postgresql.org/docs/8.3/interactive/row-estimation-examples.html>
<http://pgdocptbr.sourceforge.net/pg80/sql-explain.html>

Livro: The comprehensive guide to building, programming, and administering PostgreSQL databases, Second Edition, Capítulo 4: Understanding How PostgreSQL Executes a Query, disponível como e-book em:

<http://www.iphelp.ru/faq/15/ch04lev1sec3.html>

Introdução ao otimizador de consultas do PostgreSQL

Walter Rodrigo de Sá Cruz Criado em Wed Aug 30 11:48:04 2006 em:
<http://artigos.waltercruz.com/postgresql/otimizador>

O caminho de uma consulta

Antes de entrarmos no assunto da otimização propriamente dito, precisamos rever qual é o caminho de uma consulta no banco de dados.

(A seção a seguir é adaptada da documentação do PostgreSQL)

O programa aplicativo transmite um comando para o servidor, e aguarda para receber de volta os resultados transmitidos pelo servidor.

1. O estágio de análise verifica o comando transmitido pelo programa aplicativo com relação à correção da sintaxe, e cria a árvore de comando.
2. O sistema de reescrita recebe a árvore de comando criada pelo estágio de análise, e procura por alguma regra (armazenada nos catálogos do sistema) a ser aplicada na árvore de comando. Realiza as transformações especificadas no corpo das regras. Uma das aplicações do sistema de reescrita é a criação de visões. Sempre que é executado um comando em uma visão (ou seja, uma tabela virtual), o sistema de reescrita reescreve o comando do usuário como um comando acessando as tabelas base especificadas na definição da visão, em vez da visão.
3. O planejador/otimizador recebe a árvore de comando (reescrita), e cria o plano de comando que será a entrada do executor. Isto é feito criando primeiro todos os caminhos possíveis que levam ao mesmo resultado. Por exemplo, se existe um índice em uma relação a ser varrido, existem dois caminhos para a varredura. Uma possibilidade é uma varredura seqüencial simples, e a outra possibilidade é utilizar o índice. Em seguida é estimado o custo de execução de cada um dos caminhos, e

escolhido o mais barato. O caminho mais barato é expandido em um plano completo para que o executor possa utilizá-lo.

4. O executor caminha recursivamente através da árvore do plano, e traz as linhas no caminho representado pelo plano. O executor faz uso do sistema de armazenamento ao varrer as relações, realiza classificações e junções, avalia as qualificações e, por fim, envia de volta as linhas derivadas.

O papel do otimizador

(Adaptado de http://en.wikipedia.org/wiki/Query_optimizer)

O otimizador de consultas é o componente do banco de dados que tenta determinar o modo mais eficiente de executar uma consulta.

O PostgreSQL usa algumas estatísticas mantidas em tabelas do sistema para direcionar o otimizador. Se essas estatísticas estiverem desatualizadas, você provavelmente não obterá o melhor plano para a consulta.

Para atualizar as estatísticas, devemos executar o comando ANALYZE.

É possível vermos o plano que o otimizador de consultas do PostgreSQL gerou, usando a cláusula EXPLAIN antes da consulta.

Adicionando a cláusula EXPLAIN ANALYZE antes da consulta, ela é de fato executada, de forma que possamos ter a medida do tempo.

Exemplos de bancos de dados

Pagila

Vamos usar o banco de dados exemplo pagila, disponível em:

<http://pgfoundry.org/projects/dbsamples/>, que é um exemplo de banco de dados de uma locadora.

Vamos começar com a configuração padrão do otimizador do Postgres. Todas as variáveis no postgresql.conf estão comentadas, o que significa que elas usarão os valores padrão.

Como teste, criei dois bancos iguais, o pagila e o pagila2, com a diferença que não rodei o ANALYZE no pagila2.

A configuração do banco é a padrão do Debian.

A consulta a seguir retorna os filmes e os atores associados a ela:

```
SELECT f.title, a.first_name FROM film f
INNER JOIN film_actor ON film_actor.film_id=f.film_id
INNER JOIN actor a on film_actor.actor_id = a.actor_id
```

Aqui está o plano gerado para essa consulta no pagila2 (o que não recebeu ainda o ANALYZE):

```

Merge Join  (cost=708.95..795.88 rows=5462 width=185) (actual
time=56.274..85.883 rows=5462 loops=1)
  Merge Cond: ("outer".film_id = "inner".film_id)
    -> Sort  (cost=94.83..97.33 rows=1000 width=149) (actual time=8.489..9.926
rows=1000 loops=1)
      Sort Key: f.film_id
        -> Seq Scan on film f  (cost=0.00..45.00 rows=1000 width=149) (actual
time=0.010..4.517 rows=1000 loops=1)
        -> Sort  (cost=614.12..627.77 rows=5462 width=42) (actual time=47.775..56.152
rows=5462 loops=1)
          Sort Key: film_actor.film_id
            -> Merge Join  (cost=0.00..275.06 rows=5462 width=42) (actual
time=0.027..33.019 rows=5462 loops=1)
              Merge Cond: ("outer".actor_id = "inner".actor_id)
                -> Index Scan using actor_pkey on actor a  (cost=0.00..12.20
rows=200 width=44) (actual time=0.010..0.450 rows=200 loops=1)
                -> Index Scan using film_actor_pkey on film_actor
(cost=0.00..194.08 rows=5462 width=4) (actual time=0.006..13.315 rows=5462
loops=1)
Total runtime: 94.576 ms

```

Sem entrar nos detalhes, vamos entender esse plano. Primeiro, temos que saber olhar de baixo pra cima.

1. Foi feito um index scan na tabela film_actor (a última linha)
2. Foi feito um index table scan na tabela actor
3. Essas duas tabelas foram unidas usando o algoritmo Merge Join
4. O resultado foi ordenado pela coluna fil,_actor.film_id
5. Foi feito um table scan na tabela film, e o resultado foi ordenado pela coluna film id
6. O resultado do scan na tabela film foi juntado com o resultado anterior, usando o algoritmo Merge Join

E aqui está o plano gerado pelo pgAdmin (o banco no qual foi rodado o ANALYZE):

```

Merge Join  (cost=562.49..697.92 rows=5462 width=27) (actual time=47.691..78.064
rows=5462 loops=1)
  Merge Cond: ("outer".film_id = "inner".film_id)
    -> Index Scan using film_pkey on film f  (cost=0.00..51.00 rows=1000
width=22) (actual time=0.011..2.442 rows=1000 loops=1)
    -> Sort  (cost=562.49..576.15 rows=5462 width=11) (actual time=47.668..56.011
rows=5462 loops=1)
      Sort Key: film_actor.film_id
        -> Merge Join  (cost=0.00..223.43 rows=5462 width=11) (actual
time=0.025..32.304 rows=5462 loops=1)
          Merge Cond: ("outer".actor_id = "inner".actor_id)
            -> Index Scan using actor_pkey on actor a  (cost=0.00..6.20
rows=200 width=13) (actual time=0.007..0.459 rows=200 loops=1)
            -> Index Scan using film_actor_pkey on film_actor
(cost=0.00..148.46 rows=5462 width=4) (actual time=0.008..12.557 rows=5462
loops=1)
Total runtime: 86.610 ms

```

Vamos agora analizar esse plano:

1. Foi feito um scan na tabela film_actor pelo índice film_actor_pkey (a chave primária da tabela)
2. Agora, um scan na tabela actor pelo índice
3. As tabelas foram JOINadas pelo actor_id, e o resultado foi ordenado por film_actor.actor_id
4. A tabela film foi varrida pelo índice.
5. O resultado do JOIN anterior foi juntado com a tabela film.

Procurando clientes e filiais:

```
SELECT 'Filial ' || filial.store_id as filial, c.first_name || c.last_name as
client FROM customer c
INNER JOIN store filial on filial.store_id = c.store_id WHERE c.active = 1
```

Aqui, o explain do banco sem ANALYZE:

```
Nested Loop (cost=1.02..17.65 rows=1 width=84) (actual time=0.042..15.491
rows=584 loops=1)
  Join Filter: ("inner".store_id = "outer".store_id)
    -> Seq Scan on customer c (cost=0.00..16.49 rows=3 width=82) (actual
       time=0.013..1.621 rows=584 loops=1)
        Filter: (active = 1)
    -> Materialize (cost=1.02..1.04 rows=2 width=4) (actual time=0.002..0.006
       rows=2 loops=584)
      -> Seq Scan on store filial (cost=0.00..1.02 rows=2 width=4) (actual
         time=0.004..0.011 rows=2 loops=1)
Total runtime: 16.480 ms
```

Repare que foi usado um materialize = um plano foi salvo num arquivo temporário.

E agora, o explain do banco com ANALYZE:

```
Nested Loop (cost=4.05..46.50 rows=584 width=23) (actual time=0.175..6.097
rows=584 loops=1)
  -> Seq Scan on store filial (cost=0.00..1.02 rows=2 width=4) (actual
     time=0.008..0.015 rows=2 loops=1)
  -> Bitmap Heap Scan on customer c (cost=4.05..16.80 rows=300 width=21)
     (actual time=0.136..1.248 rows=292 loops=2)
    Recheck Cond: ("outer".store_id = c.store_id)
    Filter: (active = 1)
    -> Bitmap Index Scan on idx_fk_store_id (cost=0.00..4.05 rows=300
       width=0) (actual time=0.122..0.122 rows=300 loops=2)
      Index Cond: ("outer".store_id = c.store_id)
Total runtime: 7.160 ms
```

Nesse caso, como as estatísticas já estavam atualizadas, o banco optou por uma otimização e usou o Bitmap Index Scan e o Bitmap Heap Scan.

O Bitmap Index Scan é usado em colunas que tem pouca variação (colunas de baixa cardinalidade). No caso, para cada cliente, eu tenho apenas duas opções de filias às quais ele pode estar associado (1 e 2). Então, o banco cria um mapa de bits para cada um desses valores. Esse mapa é carregado em memória, e é juntado com o resultado do table scan em store, que tem apenas dois valores possíveis.

No último select, ainda assim foi feito um Table Scan na tabela store, onde alguém poderia argumentar que seria melhor um index scan.

Random Page Cost

Existe um parâmetro de configuração, chamado **random_page_cost** que determina qual o peso que o PostgreSQL dá a leituras não sequenciais no disco. Aumentar esse valor favorece o uso de table scans, abaixá-lo favorece o uso de índices. O valor padrão é 4.0.

Para um pequeno teste, vamos baixá-lo para 2.0 e executar a query no banco pagila.

```
Merge Join  (cost=0.00..42.34 rows=584 width=23) (actual time=0.128..6.012
rows=584 loops=1)
  Merge Cond: ("outer".store_id = "inner".store_id)
    -> Index Scan using store_pkey on store filial  (cost=0.00..3.02 rows=2
width=4) (actual time=0.092..0.098 rows=2 loops=1)
    -> Index Scan using idx_fk_store_id on customer c  (cost=0.00..27.64 rows=584
width=21) (actual time=0.015..2.150 rows=584 loops=1)
      Filter: (active = 1)
Total runtime: 6.996 ms
```

Como o custo do índice estava mais baixo, o otimizador preferiu usar o índice do que gerar o índice bitmap em memória.

Agora, no pagila2, o plano gerado é semelhante ao plano do banco que está com as estatísticas atualizadas:

```
Nested Loop  (cost=2.01..12.74 rows=1 width=84) (actual time=0.264..5.829
rows=584 loops=1)
  -> Seq Scan on store filial  (cost=0.00..1.02 rows=2 width=4) (actual
time=0.012..0.019 rows=2 loops=1)
    -> Bitmap Heap Scan on customer c  (cost=2.01..5.82 rows=3 width=82) (actual
time=0.182..1.236 rows=292 loops=2)
      Recheck Cond: ("outer".store_id = c.store_id)
      Filter: (active = 1)
        -> Bitmap Index Scan on idx_fk_store_id  (cost=0.00..2.01 rows=3
width=0) (actual time=0.165..0.165 rows=300 loops=2)
          Index Cond: ("outer".store_id = c.store_id)
Total runtime: 6.861 ms
```

Executada no pagila2, a query gera o mesmo plano que havia sido gerado para **random_page_cost** = 4.0. E, com **random_page_cost** = 2.0, a primeira query do nosso teste passa a usar o índice, mesmo sem o **analyze**. Mas os custos não são os mesmos - isso porque o pagila2 ainda não teve as estatísticas atualizadas.

Configurações do otimizador

O arquivo de configuração do postgresql (postgresql.conf) contém várias opções que tratam da configuração do otimizador e dos vários métodos de consulta. São eles:

enable_bitmapscan
enable_hashagg
enable_hashjoin
enable_indexscan
enable_mergejoin
enable_nestloop
enable_seqscan
enable_sort
enable_tidscan

Todos eles podem ser configurados para on ou off, e por padrão todos vêm habilitados.

Alguns pontos a serem notados:

enable_seqscan = off não desabilita de fato o scan sequencial, já que essa pode ser a única forma de executar certas consultas. O que esse parâmetro faz na verdade é aumentar o custo de um table scan. O mesmo vale para enable_nestloop (algumas vezes não há forma de resolver uma consulta, exceto por esse tipo de loop) e enable_sort.

Esses valores podem ser alterados em tempo de execução para uma sessão em particular, usando o comando SET.

Esse é apenas um resumo. As descrições completas das opções relativas ao otimizador são encontradas em: <http://www.postgresql.org/docs/8.0/static/runtime-config.html>.

Listando atores e filmes

A primeira consulta, embora trouxesse a lista de filmes e atores, não trazia o resultado agrupado pelos filmes, com uma lista de todos os atores.

Vamos fazer essa consulta então.

Existem duas sintaxes que podem ser usadas. A primeira:

```
select
    film.title AS title,
    array_to_string(array_accum(actor.first_name || ' ' ||
actor.last_name),',') AS actors
from
    film
    inner join film_actor on film.film_id = film_actor.film_id
    inner join actor on film_actor.actor_id = actor.actor_id
GROUP BY film.title
ORDER BY film.title
```

E seu explain:

```

Sort  (cost=768.81..771.31 rows=1000 width=37) (actual time=151.996..153.650
rows=997 loops=1)
  Sort Key: film.title
    -> HashAggregate  (cost=698.98..718.98 rows=1000 width=37) (actual
time=132.534..142.902 rows=997 loops=1)
      -> Merge Join  (cost=536.24..671.67 rows=5462 width=37) (actual
time=53.739..97.676 rows=5462 loops=1)
        Merge Cond: ("outer".film_id = "inner".film_id)
          -> Index Scan using film_pkey on film  (cost=0.00..51.00
rows=1000 width=22) (actual time=0.012..3.780 rows=1000 loops=1)
            -> Sort  (cost=536.24..549.90 rows=5462 width=21) (actual
time=53.716..63.214 rows=5462 loops=1)
              Sort Key: film_actor.film_id
                -> Merge Join  (cost=0.00..197.18 rows=5462 width=21)
(actual time=0.026..36.636 rows=5462 loops=1)
                  Merge Cond: ("outer".actor_id = "inner".actor_id)
                    -> Index Scan using actor_pkey on actor
(cost=0.00..6.20 rows=200 width=23) (actual time=0.007..0.536 rows=200 loops=1)
                      -> Index Scan using film_actor_pkey on film_actor
(cost=0.00..122.21 rows=5462 width=4) (actual time=0.009..14.883 rows=5462
loops=1)
Total runtime: 159.766 ms

```

Uma outra forma possível de fazer essa consulta é essa:

```

SELECT f.title,
array_to_string(array(select first_name FROM film_actor fa
join actor a ON fa.actor_id = a.actor_id and fa.film_id = f.film_id
),', ') as film_actor
FROM film f

```

Será que essa forma, além de ser menos compacta, é mais rápida também?

```

Seq Scan on film f  (cost=0.00..16382.69 rows=1000 width=22) (actual
time=1.214..712.919 rows=1000 loops=1)
  SubPlan
    -> Merge Join  (cost=9.56..16.34 rows=5 width=9) (actual time=0.199..0.687
rows=5 loops=1000)
      Merge Cond: ("outer".actor_id = "inner".actor_id)
        -> Index Scan using actor_pkey on actor a  (cost=0.00..6.20 rows=200
width=13) (actual time=0.005..0.347 rows=165 loops=1000)
          -> Sort  (cost=9.56..9.57 rows=5 width=2) (actual time=0.056..0.069
rows=5 loops=1000)
            Sort Key: fa.actor_id
              -> Bitmap Heap Scan on film_actor fa  (cost=2.02..9.50 rows=5
width=2) (actual time=0.020..0.033 rows=5 loops=1000)
                Recheck Cond: (film_id = $0)
                  -> Bitmap Index Scan on idx_fk_film_id  (cost=0.00..2.02
rows=5 width=0) (actual time=0.013..0.013 rows=5 loops=1000)
                    Index Cond: (film_id = $0)
Total runtime: 714.655 ms

```

Parece que não.

Vemos que na consulta, o maior tempo gasto está no merge join (de 0.199 até 0.687).

Vamos desabilitá-lo então. Isso é feito com o comando: set enable_mergejoin = off.

Vejamos o novo plano:

```

Seq Scan on film f  (cost=0.00..24679.64 rows=1000 width=22) (actual
time=0.368..162.552 rows=1000 loops=1)
  SubPlan
    -> Nested Loop  (cost=2.02..24.63 rows=5 width=9) (actual time=0.037..0.136
rows=5 loops=1000)
      -> Bitmap Heap Scan on film_actor fa  (cost=2.02..9.50 rows=5
width=2) (actual time=0.020..0.039 rows=5 loops=1000)
        Recheck Cond: (film_id = $0)
        -> Bitmap Index Scan on idx_fk_film_id  (cost=0.00..2.02 rows=5
width=0) (actual time=0.014..0.014 rows=5 loops=1000)
          Index Cond: (film_id = $0)
        -> Index Scan using actor_pkey on actor a  (cost=0.00..3.01 rows=1
width=13) (actual time=0.007..0.009 rows=1 loops=5462)
          Index Cond: ("outer".actor_id = a.actor_id)
Total runtime: 164.212 ms

```

Fiz um pequeno teste com o beta do PostgreSQL 8.2, e para minha surpresa, o plano gerado para essa consulta é o mesmo que é gerado no 8.1 usando enable_mergejoin = off (com random_page_cost = 2.0).

Entendendo os ícones usados no pgadmin

Escrevi uma pequena descrição para os ícones usados no pgadmin, que está disponível em: <http://artigos.waltercruz.com/postgresql/explain/>

Outra coisa interessante pra se notar é que a grossura das setas é proporcional ao custo das operações.

3 – Rotina de reindexação

Em algumas situações vale a pena reconstruir índices periodicamente utilizando o comando REINDEX. Existe, também, o aplicativo contrib/reindexdb que pode reindexar todo o banco de dados. **Entretanto, a partir da versão 7.4 o PostgreSQL reduziu de forma substancial a necessidade desta atividade se comparado às versões anteriores.**

REINDEX -- reconstrói índices

Sintaxe

```
REINDEX { INDEX | TABLE | DATABASE | SYSTEM } nome [ FORCE ]
```

Exemplos

Reconstruir um único índice:

```
REINDEX INDEX meu_indice;
```

Reconstruir os índices da tabela minha_tabela:

```
REINDEX TABLE minha_tabela;
```

Reconstruir todos os índices de um determinado banco de dados, sem confiar que os índices do sistema estejam válidos:

```
$ export PGOPTIONS="-P"
$ psql bd_danificado
...
bd_danificado=> REINDEX DATABASE bd_danificado;
bd_danificado=> \q
```

Compatibilidade

Não existe o comando REINDEX no padrão SQL.

Existe também o contrib reindexdb, que reindexa todo um banco de dados:

Exemplos:

Para reindexar todos os índices do banco de dados teste:

```
$ reindexdb -U postgres teste
```

Para reindexar o índice clientes_idx da tabela clientes do banco de dados dba_projeto:

```
$ reindexdb -U postgres --table clientes --index clientes_idx dba_projeto
```

Mais detalhes em: <http://pgdocptbr.sourceforge.net/pg82/sql-reindex.html>

4 – Monitorando o espaço em disco e tamanho de objetos

Cada tabela possui um arquivo em disco, onde a maior parte dos dados são armazenados. Se a tabela possuir alguma coluna com valor potencialmente longo, também existirá um arquivo TOAST associado à tabela, utilizado para armazenar os valores muito longos para caber confortavelmente na tabela principal. Haverá um índice para a tabela TOAST, caso esta esteja presente. Também podem haver índices associados à tabela base. **Cada tabela e índice é armazenado em um arquivo em disco separado — possivelmente mais de um arquivo, se o arquivo exceder um gigabyte.**

O espaço em disco pode ser monitorado a partir de três lugares: do psql utilizando as informações do VACUUM, do psql utilizando as ferramentas presentes em contrib/dbsize, e da linha de comando utilizando as ferramentas presentes em contrib/oid2name. Utilizando o psql em um banco de dados onde o comando VACUUM ou ANALYZE foi executado recentemente, podem ser efetuadas consultas para ver a utilização do espaço em disco de qualquer tabela:

```
\c cep_brasil
VACUUM ANALYZE;

SELECT relname, relfilenode, relpages
FROM pg_class
WHERE relname LIKE 'cep%'
```

```
ORDER BY relname;
```

Ou:

```
SELECT relfilenode, relpages FROM pg_class WHERE relname = 'cep_full_index';
```

Ou:

```
SELECT relfilenode as tabela_inode, relpages*8*1024/(1024*1024) AS "Espaço em MB" FROM pg_class WHERE relname = 'cep_full_index';
```

Ou:

```
SELECT relfilenode as tabela_inode, relpages*8/1024 AS "Espaço em MB" FROM pg_class WHERE relname = 'cep_full_index';
```

Cada página possui, normalmente, 8 kilobytes (Lembre-se, relpages somente é atualizado por VACUUM, ANALYZE e uns poucos comandos de DDL como CREATE INDEX). O valor de relfilenode possui interesse caso se deseje examinar diretamente o arquivo em disco da tabela.

Para ver o espaço utilizado pelas tabelas TOAST deve ser utilizada uma consulta como a mostrada abaixo:

```
-- Ver as tabelas TOAST da tabela pg_rewrite
SELECT relname, relpages
  FROM pg_class,
       (SELECT reltoastrelid FROM pg_class
        WHERE relname = 'pg_rewrite') ss
 WHERE oid = ss.reltoastrelid
   OR oid = (SELECT reltoastidxid FROM pg_class
             WHERE oid = ss.reltoastrelid)
 ORDER BY relname;
```

Os tamanhos dos índices também podem ser facilmente exibidos:

```
SELECT c2.relname, c2.relpages
  FROM pg_class c, pg_class c2, pg_index i
 WHERE c.relname = 'cep_full_index'
   AND c.oid = i.indrelid
   AND c2.oid = i.indexrelid
 ORDER BY c2.relname;
```

Adaptação de: <http://pgdocptbr.sourceforge.net/pg80/diskusage.html>

Falha de disco cheio

A tarefa mais importante de monitoramento de disco do administrador de banco de dados é garantir que o disco não vai ficar cheio. Um disco de dados cheio não resulta em corrupção dos dados, mas pode impedir que ocorram atividades úteis. Se o disco que contém os arquivos do WAL ficar cheio, pode acontecer do servidor de banco de dados entrar na condição de pânico e encerrar a execução.

Se não for possível liberar espaço adicional no disco removendo outros arquivos, podem ser movidos alguns arquivos do banco de dados para outros sistemas de arquivos utilizando espaços de tabelas (tablespaces).

Dica: Alguns sistemas de arquivos têm desempenho degradado quando estão praticamente cheios, portanto não se deve esperar o disco ficar cheio para agir.

Se o sistema possuir cotas de disco por usuário, então naturalmente o banco de dados estará sujeito a cota atribuída para o usuário sob o qual o sistema de banco de dados executa. Exceder a cota ocasiona os mesmos malefícios de ficar totalmente sem espaço.

De: <http://pgdocptbr.sourceforge.net/pg80/disk-full.html>

5 – Monitorando o arquivo de registros (logs)

Estes arquivos guardam informações detalhadas de todas as operações do PostgreSQL, tanto as operações bem sucedidas quanto às más sucedidas, erros e alertas.

São arquivos com nomes no seguinte formato:

postgresql-2008-03-16_072122.log

postgresql – data – horadaprimeiraocorrência.log

E internamente exibe os registros em linhas assim:

```
2008-03-16 08:56:46 GMT ERROR: relation "wlw" does not exist
2008-03-16 08:56:46 GMT STATEMENT: select * from wlw;
```

Uma linha para o erro e a seguinte para a entrada que causou o erro.

Recomenda-se que o DBA mantenha sempre um olho nos logs.

Cada vez que o PostgreSQL é iniciado é criado um arquivo de log. Neste arquivo são registradas todas as ocorrências a partir daí e criado um novo arquivo na próxima vez que o servidor iniciar ou quando chegar a hora de rolar (configurado no postgresql.conf logrotate).

No postgresql.conf estão as regras:

```
#log_rotation_age = 1d
#log_rotation_size = 10MB
```

A cada dia ou quando chegar a 10MB um novo arquivo será criado.

Fique atento também para o volume que os logs ocupam em disco, pois podem comprometer o servidor.

No Windows localizan-se em:

C:\Program Files\PostgreSQL\8.3\data\pg_log

No Linux varia de acordo com a distribuição ou forma de instalação do PostgreSQL.

No postgresql.conf estão parâmetros que controlam e configuram os logs:

```
#log_directory = 'pg_log'  
#log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'
```

E muitas outras configurações dos logs.

É uma boa prática, periodicamente, remover os arquivos mais antigos. Se será feito backup dos mesmos ou não, depende da importância dos mesmos.

O diretório pg_xlog contém os arquivos de logs das transações. Tamanhos fixos de 16MB.

O diretório pg_clog guarda o status das transações.

pg_subtrans - status das subtransações

pg_tblspc - links simbolicos de tablespace

Ativando os Logs no PostgreSQL 8.3 do Linux Ubuntu 7.10

Editar o script postgresql.conf

```
sudo gedit /etc/postgresql/8.3/main/postgresql.conf
```

E alterar a linha:

```
#logging_collector = off
```

Copiando-a para uma linha abaixo:

```
logging_collector = on
```

Restartar o servidor:

```
sudo /etc/init.d/postgresql-8.3 restart
```

Depois disso podemos acessar os logs como usuário postgres:

```
su – postgres
```

```
cd /var/lib/postgresql/8.3/main/pg_log
```

Sobre as configurações dos logs veja:

<http://pgdocptbr.sourceforge.net/pg80/runtime-config.html#RUNTIME-CONFIG-LOGGING-WHERE>

Referências

<http://pgdocptbr.sourceforge.net/pg80/diskusage.html>

<https://www.postgresql.org/docs/9.6/static/monitoring.html>

<https://www.postgresql.org/docs/9.6/static/diskusage.html>

Explain



dex scan: Foi feito uma varredura no índice da tabela



Table scan: Foi feito um table scan na tabela



Aggregate(ou Hash Aggregate): Indica uma função agregada. (COUNT SUM MIN MAX AVG STDDEV VARIANCE)



Append: Disparado por UNIONs e por herança. Seu custo é simplesmente a soma de todas as entradas.



Bitmap Heap Scan: É usado um bitmap (mapa de bits). Usado em colunas de baixa cardinalidade.



Bitmap Index Scan: É usado um bitmap (mapa de bits) a partir do índice. Usado em colunas de baixa cardinalidade.



Agrupamento: Usado na cláusula group by



Hash: Denota uma função hash executada na tabela (para ser usada posteriormente em um hash join ou hash left join)



Hash join (ou Hash Left Join):²



Limit:

Cláusula de limitação



Materialize: É usado com sub-selects e merge-joins. O resultado é colocado numa tabela temporária.



Merge Join: O³ Merge Join é um operador binário. Os lados esquerdo e direito são os fluxo de dados externo e interno, respectivamente. Ambos fluxos devem ser organizados de acordo com a chave do merge-join. Primeiro, uma linha do fluxo exterior é lido. Isso inicializa os valores das chaves do merge join. Então, linhas do fluxo interno são lidas até que uma linha tenha um valor que case ou seja maior (exceto se a coluna chave é descendente) seja encontrada. Se as chaves da junção casam, então a coluna qualificadora é passada adiante para processamento adicional, e uma próxima chamada ao operador merge-join continua lendo do fluxo de dados atual. Se os novos valores são maiores que a chave de comparação, então esses valores são usados como a chave de

junção, enquanto a traz linhas dos outros fluxos de dados. Esse processo continua até um dos fluxo de dados exaurir.



Nested Loop (ou Nested Loop Left Join): Loop Aninhado¹: A relação à direita é escaneada uma vez para cada linha encontrada na relação à esquerda. Essa estratégia é fácil de implementar, mas pode ser uma grande consumidora de tempo. (Porém, se a relação direita pode ser varrida usando um índicescan, essa pode ser uma boa estratégia. É possível usar valores da linha atual da relação esquerda como chaves para a varredura de índice da relação direita.)



Function: Resultado de uma função



Explicação



SetOp Intersect, SetOp Intersect All, SetOp Except, SetOp Except All : Encontra grupos de linhas duplicadas. Usado para INTERSECT, INTERSECT ALL, EXCEPT, EXCEPT ALL.



Sort: Ordenação



Subquery: Usado com UNIONs



Varredura TID: Id da coluna da tupla. É usado apenas quando o cláusula "ctid =" aparece na sua query. Muito raro, e muito rápido.



Unique: Usado quando queremos obter dados únicos da tabela (por exemplo, com DISTINCT)



Desconhecido: Desconhecido

1. Retirado da documentação do PostgreSQL

Explain

<https://www.postgresql.org/docs/9.6/static/using-explain.html>

<https://www.postgresql.org/docs/9.6/static/sql-explain.html>

<https://www.postgresql.org/docs/9.6/static/auto-explain.html>

Ver as estatísticas coletadas

Estão disponíveis diversas visões pré-definidas para mostrar os resultados das estatísticas coletadas, conforme listado na Tabela 23-1. Como alternativa, podem ser construídas visões personalizadas utilizando as funções de estatísticas subjacentes.

Ao se utilizar as estatísticas para monitorar a atividade corrente, é importante ter em mente que as informações não são atualizadas instantaneamente. Cada processo servidor individual transmite os novos contadores de acesso a bloco e a linha para o coletor logo antes de ficar ocioso; portanto, um comando ou transação ainda em progresso não afeta os totais exibidos. Também, o próprio coletor emite um novo relatório no máximo uma vez a cada pgstat_stat_interval milissegundos (500 por padrão). Portanto, as informações mostradas são anteriores à atividade corrente. A informação do comando corrente é enviada para o coletor imediatamente, mas ainda está sujeita ao retardo de pgstat_stat_interval antes de se tornar visível.

Outro ponto importante é que, quando se solicita a um processo servidor para mostrar uma destas estatísticas, primeiro este busca os relatórios mais recentes emitidos pelo processo coletor, e depois continua utilizando este instantâneo para todas as visões e funções de estatística até o término da transação corrente. Portanto, as estatísticas parecem não mudar enquanto se permanece na transação corrente. Isto é uma característica, e não um erro, porque permite realizar várias consultas às estatísticas e correlacionar os resultados sem se preocupar com números variando por baixo. Se desejar ver novos resultados a cada consulta, certifique-se que as consultas estão fora de qualquer bloco de transação.

Visões de estatísticas padrão

Nome da visão	Descrição
pg_stat_activity	Uma linha por processo servidor, mostrando o ID do processo, o banco de dados, o usuário, o comando corrente e a hora em que o comando corrente começou a executar. As colunas que mostram os dados do comando corrente somente estão disponíveis quando o parâmetro stats_command_string está ativado. Além disso, estas colunas mostram o valor nulo a menos que o usuário consultando a visão seja um superusuário, ou o mesmo usuário dono do processo sendo mostrado (Deve ser observado que devido ao retardo do que é informado pelo coletor, o comando corrente somente será mostrado no caso dos comandos com longo tempo de execução).
pg_stat_database	Uma linha por banco de dados, mostrando o número de processos servidor ativos, total de transações efetivadas e total de transações canceladas neste banco de dados, total de blocos de disco lidos e total de acertos no buffer (ou seja, solicitações de leitura de bloco evitadas por encontrar o bloco no cache do buffer).
pg_stat_all_tables	Para cada tabela do banco de dados corrente, o número total de: varreduras seqüenciais e de índice; linhas retornadas por cada tipo de varredura; linhas inseridas, atualizadas e excluídas.
pg_stat_sys_tables	O mesmo que pg_stat_all_tables, exceto que somente são mostradas as tabelas do sistema.
pg_stat_user_tables	O mesmo que pg_stat_all_tables, exceto que somente são mostradas as tabelas de usuário.

`pg_stat_all_indexes` Para cada índice do banco de dados corrente, o total de varreduras de índice que utilizaram este índice, o número de linhas do índice lidas, e o número de linhas da tabela buscadas com sucesso (Pode ser menor quando existem entradas do índice apontando para linhas da tabela expiradas)

`pg_stat_sys_indexes` O mesmo que `pg_stat_all_indexes`, exceto que somente são mostrados os índices das tabelas do sistema.

`pg_stat_user_indexes` O mesmo que `pg_stat_all_indexes`, exceto que somente são mostrados os índices das tabelas de usuário.

`pg_statio_all_tables` Para cada tabela do banco de dados corrente, o número total de blocos de disco da tabela lidos, o número de acertos no buffer, o número de blocos de disco lidos e acertos no buffer para todos os índices da tabela, o número de blocos de disco lidos e acertos no buffer para a tabela auxiliar TOAST da tabela (se houver), e o número de blocos de disco lidos e acertos no buffer para o índice da tabela TOAST.
`pg_statio_sys_tables` O mesmo que `pg_statio_all_tables`, exceto que somente são mostradas as tabelas do sistema.

`pg_statio_user_tables` O mesmo que `pg_statio_all_tables`, exceto que somente são mostradas as tabelas de usuário.

`pg_statio_all_indexes` Para cada índice do banco de dados corrente, o número de blocos de disco lidos e de acertos no buffer para o índice.

`pg_statio_sys_indexes` O mesmo que `pg_statio_all_indexes`, exceto que somente são mostrados os índices das tabelas do sistema.

`pg_statio_user_indexes` O mesmo que `pg_statio_all_indexes`, exceto que somente são mostrados os índices das tabelas de usuário.

`pg_statio_all_sequences` Para cada objeto de seqüência do banco de dados corrente, o número de blocos de disco lidos e de acertos no buffer para a seqüência.

`pg_statio_sys_sequences` O mesmo que `pg_statio_all_sequences`, exceto que somente são mostradas as seqüências do sistema (Atualmente não está definida nenhuma seqüência do sistema e, portanto, esta visão está sempre vazia).

`pg_statio_user_sequences` O mesmo que `pg_statio_all_sequences`, exceto que somente são mostradas as seqüências de usuário.

As estatísticas por índice são particularmente úteis para determinar quais índices estão sendo utilizados e quão efetivos são.

As visões `pg_statio_` são úteis, principalmente, para determinar a efetividade do cache do buffer. Quando o número de leituras físicas no disco é muito menor do que o número de acertos no buffer, então o cache está respondendo à maioria das solicitações de leitura, evitando chamadas ao núcleo. Entretanto, estas estatísticas não fornecem toda a história: devido à forma como o PostgreSQL trata a E/S em disco, dados que não estão no cache do buffer do PostgreSQL podem estar no cache de E/S do núcleo e, portanto, podem ser lidos sem que haja necessidade de uma leitura física. Os usuários interessados em obter informações mais detalhadas sobre o comportamento de E/S do PostgreSQL, são aconselhados a utilizar o coletor de estatísticas do PostgreSQL em combinação com os utilitários do sistema operacional que permitem analisar o tratamento da E/S pelo núcleo.

Podem ser criadas outras formas de ver as estatísticas, escrevendo consultas que utilizam as mesmas funções subjacentes de acesso às estatísticas utilizadas pelas visões padrão. Esta funções estão listadas na Tabela 23-2. As funções de acesso por banco de dados, recebem como argumento o OID do banco de dados que identifica para qual banco de dados é o relatório. As funções por tabela e por índice recebem o OID da tabela

ou do índice, respectivamente (Deve ser observado que somente podem ser vistos por estas funções as tabelas e índices presentes no banco de dados corrente). As funções de acesso por processo servidor recebem o número de ID do processo servidor, que varia de um ao número de processos servidor ativos no momento.

Funções de acesso às estatísticas

Função	Tipo retornado	Descrição
pg_stat_get_db_numbackends(oid)	integer	Número de processos servidor ativos conectados ao banco de dados
pg_stat_get_db_xact_commit(oid)	bigint	Transações efetivadas no banco de dados
pg_stat_get_db_xact_rollback(oid)	bigint	Transações canceladas no banco de dados
pg_stat_get_db_blocks_fetched(oid)	bigint	Número de solicitações de busca de blocos de disco para o banco de dados
pg_stat_get_db_blocks_hit(oid)	bigint	Número de solicitações de busca de blocos de disco para o banco de dados encontradas no cache
pg_stat_get_numscans(oid)	bigint	Número de varreduras seqüenciais realizadas quando o argumento é uma tabela, ou o número de varreduras de índice quando o argumento é um índice
pg_stat_get_tuples_returned(oid)	bigint	Número de linhas lidas por varreduras seqüenciais quando o argumento é uma tabela, ou o número de linhas do índice lidas quando o argumento é um índice
pg_stat_get_tuples_fetched(oid)	bigint	Número de linhas válidas (não expiradas) da tabela buscadas por varreduras seqüenciais quando o argumento é uma tabela, ou buscadas por varreduras de índice, utilizando este índice, quando o argumento é um índice
pg_stat_get_tuples_inserted(oid)	bigint	Número de linhas inseridas na tabela
pg_stat_get_tuples_updated(oid)	bigint	Número de linhas atualizadas na tabela
pg_stat_get_tuples_deleted(oid)	bigint	Número de linhas excluídas da tabela
pg_stat_get_blocks_fetched(oid)	bigint	Número de solicitações de busca de bloco de disco para a tabela ou índice
pg_stat_get_blocks_hit(oid)	bigint	Número de solicitações de busca de bloco de disco encontradas no cache para a tabela ou o índice
pg_stat_get_backend_idset()	conjunto de integer	Conjunto de IDs de processos servidor ativos no momento (de 1 ao número de processos servidor ativos). Veja o exemplo de utilização no texto.
pg_backend_pid()	integer	ID de processo do processo servidor conectado à sessão corrente
pg_stat_get_backend_pid(integer)	integer	ID de processo do processo servidor especificado
pg_stat_get_backend_dbid(integer)	oid	ID de banco de dados do processo servidor especificado
pg_stat_get_backend_userid(integer)	oid	ID de usuário do processo servidor especificado
pg_stat_get_backend_activity(integer)	text	Comando ativo do processo servidor especificado (nulo se o usuário corrente não for um superusuário nem o mesmo usuário da sessão sendo consultada, ou se stats_command_string não estiver ativo)
pg_stat_get_backend_activity_start(integer)	timestamp with time zone	A hora em que o comando executando no momento, no processo servidor especificado, começou (nulo

se o usuário corrente não for um superusuário nem o mesmo usuário da sessão sendo consultada, ou se stats_command_string não estiver ativo)
pg_stat_reset() boolean Reinicia todas as estatísticas atualmente coletadas

Nota: pg_stat_get_db_blocks_fetched menos pg_stat_get_db_blocks_hit fornece o número de chamadas à função read() do núcleo feitas para a tabela, índice ou banco de dados; mas o número verdadeiro de leituras físicas é geralmente menor por causa da "buferização" no nível do núcleo.

A função pg_stat_get_backend_idset fornece uma maneira conveniente de gerar uma linha para cada processo servidor ativo. Por exemplo, para mostrar o PID e o comando corrente de todos os processos servidor:

```
SELECT pg_stat_get_backend_pid(s.backendid) AS procpid,
       pg_stat_get_backend_activity(s.backendid) AS current_query
  FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

Características de um bom hardware para servidor

Quando escolhemos uma máquina para ser o servidor do PostgreSQL devemos atentar para alguns detalhes:

- Escolher o sistema operacional de acordo com a segurança desejada, com a experiência dos administradores, com características desejadas (tablespace requer link simbólico) (ou treinar administradores para o SO escolhido, ou ainda contratar administradores com experiência);
- Escolher o hardware adequado para ser o servidor;
- É importante que o servidor tenha o melhor desempenho possível, portanto devemos optar:
 - pela ausência de ambiente gráfico (se presente, que seja um magro)
 - uso de kernel otimizado para o PostgreSQL
 - Instalar somente os serviços essenciais
 - otimizar as configurações do PostgreSQL
 - otimizar o hardware
- Devemos fazer uma boa inspeção em tudo que se relaciona à segurança:
 - senhas e privilégios de usuários
 - acesso físico dos administradores
 - backups regulares
 - mídias de boa qualidade
 - local adequado para armazenamento da mídia
 - sanidade do servidor (proteção antes de chegar ao mesmo, através de firewall, anti-virus, etc)
 - etc.
- O Ubuntu e outras distribuições contam com uma versão de suas distribuições específica para servidores
- O Windows 2008 server agora vem com a opção de se instalar apenas o core.

Bom tutorial de instalação do Ubuntu-server:
"Instalar un servidor con ubuntu-server"

Particionamento de Tabelas no PostgreSQL

Adaptado do artigo
<https://savepoint.blog.br/2013/01/09/particionamento-de-tabelas-no-postgres-como/>
Do Fábio Telles

52 - TableSpaces

Antes de criar as tabelas, criaremos o banco, os tablespaces e somente então criaremos as tabelas e seus índices nos respectivos tablespaces.

No caso criarei o banco estoque e nele todos os exercícios abaixo:

Criaremos 4 tablespaces:

- Um para a tabela pedidos
- Um para os índices da tabela pedidos
- Um para a tabela pedido_itens
- Um para os índices da tabela pedido_itens

```
# Criando diretórios para novos TABLESPACES  
sudo su
```

```
mkdir -p /data/pedidos  
mkdir -p /data/pedido_itens  
mkdir -p /index/pedidos  
mkdir -p /index/pedido_itens  
chown -R postgres: /data  
chown -R postgres: /index
```

Acessar o psql e entrar no banco:

```
su - postgres  
psql
```

```
CREATE DATABASE estoque;
```

```
\c estoque
```

```
-- Criar os tablespaces antes de criar as tabelas  
-- Tablespaces novos nos diretórios criados anteriormente  
CREATE TABLESPACE pedidos LOCATION '/data/pedidos';
```

```
CREATE TABLESPACE pedido_itens LOCATION '/data/pedido_itens';
CREATE TABLESPACE pedidos_idx LOCATION '/index/pedidos';
CREATE TABLESPACE pedido_itens_idx LOCATION '/index/pedido_itens';
```

-- Criação de usuários app e esquema. Os usuários app serão donos das tabelas
 CREATE ROLE app PASSWORD 'app' LOGIN;
 CREATE SCHEMA app;
 GRANT ALL ON SCHEMA app TO app;

```
CREATE ROLE client PASSWORD 'app' LOGIN;
GRANT USAGE ON SCHEMA app TO client;
```

-- Tabelas não particionadas

```
CREATE TABLE app.cliente (
    id_cliente INTEGER,
    nome_cliente VARCHAR(100),
    CONSTRAINT cliente_pk PRIMARY KEY (id_cliente)
);
ALTER TABLE app.cliente OWNER TO app;
```

```
CREATE TABLE app.vendedor (
    id_vendedor INTEGER,
    nome_vendedor VARCHAR(100),
    CONSTRAINT vendedor_pk PRIMARY KEY (id_vendedor)
);
ALTER TABLE app.vendedor OWNER TO app;
```

-- Criação da tabela PEDIDO

```
CREATE TABLE app.pedidos (
    ano_pedido SMALLINT,
    id_pedido INTEGER,
    data_pedido TIMESTAMP(2) NOT NULL DEFAULT now(),
    id_cliente INTEGER NOT NULL,
    id_vendedor INTEGER NOT NULL,
    status CHAR(1),
    observacao TEXT,
    CONSTRAINT pedido_pk PRIMARY KEY (ano_pedido, id_pedido)
        WITH (fillfactor=100) USING INDEX TABLESPACE pedidos_idx,
    CONSTRAINT pedido_cliente_fk FOREIGN KEY (id_cliente)
        REFERENCES app.cliente(id_cliente),
    CONSTRAINT pedido_vendedor_fk FOREIGN KEY (id_vendedor)
        REFERENCES app.vendedor(id_vendedor),
    CONSTRAINT pedido_status_ck CHECK (status IN ('A', 'B', 'I', 'Z')),
    CONSTRAINT pedido_null_ck CHECK (ano_pedido IS NULL) NO INHERIT -- Só
funciona a partir do PG 9.2
) WITH (autovacuum_vacuum_scale_factor=0.1,fillfactor=70) TABLESPACE pedidos;
```

```
CREATE INDEX pedido_data_pedido_index ON app.pedidos (data_pedido) WITH
(fillfactor=95) TABLESPACE pedidos_idx;
```

```

CREATE INDEX pedido_id_cliente_index ON app_pedidos (id_cliente) WITH (fillfactor=95)
TABLESPACE pedidos_idx;
CREATE INDEX pedido_id_vendedor_index ON app_pedidos (id_vendedor) WITH
(fillfactor=95) TABLESPACE pedidos_idx;

```

```

COMMENT ON TABLE app_pedidos IS 'Tabela de pedidos foo';
COMMENT ON COLUMN app_pedidos.ano_pedido IS 'Campo chave do particionamento';
COMMENT ON CONSTRAINT pedido_null_ck ON app_pedidos IS 'Restrição para impedir
registros na tabela mãe';
COMMENT ON INDEX app_pedido_data_pedido_index IS 'Índice from hell';

```

```

ALTER TABLE app_pedidos OWNER TO app;
GRANT SELECT, INSERT, UPDATE ON TABLE app_pedidos TO client;

```

```

-- Criação da tabela PEDIDO_DETALHE
CREATE TABLE app_pedido_itens (
    ano_pedido SMALLINT,
    id_pedido INTEGER,
    id_produto INTEGER,
    valor_unidade NUMERIC(10,2),
    desconto NUMERIC(10,2),
    quantidade INTEGER,
    CONSTRAINT pedido_detalhe_pk PRIMARY KEY (id_pedido, ano_pedido, id_produto)
        WITH (fillfactor = 95) USING INDEX TABLESPACE pedido_itens_idx,
    CONSTRAINT pedido_detalhe_fk FOREIGN KEY (id_pedido, ano_pedido)
        REFERENCES app_pedidos (id_pedido, ano_pedido)
) WITH (fillfactor = 100)
TABLESPACE pedido_itens;

```

```

COMMENT ON TABLE app_pedido_itens IS 'Itens do pedido';
CREATE INDEX pedido_detalhe_id_produto_index ON app_pedido_itens (id_produto)
    WITH (fillfactor = 95) TABLESPACE pedido_itens_idx;

```

```

ALTER TABLE app_pedido_itens OWNER TO app;
GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE app_pedido_itens TO client;

```

-- Um DBA experiente costuma olhar com atenção para tabelas grandes, as que geralmente particionamos.

-- Criação de Tabelas Filhas

-- Para criar tabelas filhas usamos herança, assim:

```

-- INCLUDING COMMENTS só funciona a partir do PG 9.0
-- INCLUDING STORAGE só funciona a partir do PG 9.0
-- INCLUDING INDEXES só funciona a partir do PG 8.3
-- INCLUDING CONSTRAINTS só funciona a partir do PG 8.2
-- INCLUDING DEFAULT só funciona a partir do PG 7.4

```

-- Criar Todas as Partições

-- INCLUDING ALL só funciona a partir do PG 9.0

```

CREATE TABLE app_pedidos_2008 (LIKE app_pedidos INCLUDING ALL) INHERITS
(app_pedidos);
CREATE TABLE app_pedidos_2009 (LIKE app_pedidos INCLUDING ALL) INHERITS
(app_pedidos);
CREATE TABLE app_pedidos_2010 (LIKE app_pedidos INCLUDING ALL) INHERITS
(app_pedidos);
CREATE TABLE app_pedidos_2011 (LIKE app_pedidos INCLUDING ALL) INHERITS
(app_pedidos);
CREATE TABLE app_pedidos_2012 (LIKE app_pedidos INCLUDING ALL) INHERITS
(app_pedidos);
CREATE TABLE app_pedidos_2013 (LIKE app_pedidos INCLUDING ALL) INHERITS
(app_pedidos);

ALTER TABLE app_pedidos_2008 DROP CONSTRAINT pedido_null_ck;
ALTER TABLE app_pedidos_2009 DROP CONSTRAINT pedido_null_ck;
ALTER TABLE app_pedidos_2010 DROP CONSTRAINT pedido_null_ck;
ALTER TABLE app_pedidos_2011 DROP CONSTRAINT pedido_null_ck;
ALTER TABLE app_pedidos_2012 DROP CONSTRAINT pedido_null_ck;
ALTER TABLE app_pedidos_2013 DROP CONSTRAINT pedido_null_ck;

ALTER TABLE app_pedidos_2008 ADD CONSTRAINT pedido_cliente_fk_2008 FOREIGN
KEY (id_cliente) REFERENCES app_cliente(id_cliente);
ALTER TABLE app_pedidos_2009 ADD CONSTRAINT pedido_cliente_fk_2009 FOREIGN
KEY (id_cliente) REFERENCES app_cliente(id_cliente);
ALTER TABLE app_pedidos_2010 ADD CONSTRAINT pedido_cliente_fk_2010 FOREIGN
KEY (id_cliente) REFERENCES app_cliente(id_cliente);
ALTER TABLE app_pedidos_2011 ADD CONSTRAINT pedido_cliente_fk_2011 FOREIGN
KEY (id_cliente) REFERENCES app_cliente(id_cliente);
ALTER TABLE app_pedidos_2012 ADD CONSTRAINT pedido_cliente_fk_2012 FOREIGN
KEY (id_cliente) REFERENCES app_cliente(id_cliente);
ALTER TABLE app_pedidos_2013 ADD CONSTRAINT pedido_cliente_fk_2013 FOREIGN
KEY (id_cliente) REFERENCES app_cliente(id_cliente);

ALTER TABLE app_pedidos_2008 ADD CONSTRAINT pedido_vendedor_fk_2008 FOREIGN
KEY (id_vendedor) REFERENCES app_vendedor(id_vendedor);
ALTER TABLE app_pedidos_2009 ADD CONSTRAINT pedido_vendedor_fk_2009 FOREIGN
KEY (id_vendedor) REFERENCES app_vendedor(id_vendedor);
ALTER TABLE app_pedidos_2010 ADD CONSTRAINT pedido_vendedor_fk_2010 FOREIGN
KEY (id_vendedor) REFERENCES app_vendedor(id_vendedor);
ALTER TABLE app_pedidos_2011 ADD CONSTRAINT pedido_vendedor_fk_2011 FOREIGN
KEY (id_vendedor) REFERENCES app_vendedor(id_vendedor);
ALTER TABLE app_pedidos_2012 ADD CONSTRAINT pedido_vendedor_fk_2012 FOREIGN
KEY (id_vendedor) REFERENCES app_vendedor(id_vendedor);
ALTER TABLE app_pedidos_2013 ADD CONSTRAINT pedido_vendedor_fk_2013 FOREIGN
KEY (id_vendedor) REFERENCES app_vendedor(id_vendedor);

ALTER TABLE app_pedidos_2008 SET
(autovacuum_vacuum_scale_factor=0.1,fillfactor=70);

```

```

ALTER TABLE app_pedidos_2009 SET
(autovacuum_vacuum_scale_factor=0.1,fillfactor=70);
ALTER TABLE app_pedidos_2010 SET
(autovacuum_vacuum_scale_factor=0.1,fillfactor=70);
ALTER TABLE app_pedidos_2011 SET
(autovacuum_vacuum_scale_factor=0.1,fillfactor=70);
ALTER TABLE app_pedidos_2012 SET
(autovacuum_vacuum_scale_factor=0.1,fillfactor=70);
ALTER TABLE app_pedidos_2013 SET
(autovacuum_vacuum_scale_factor=0.1,fillfactor=70);

```

```

ALTER TABLE app_pedidos_2008 OWNER TO app;
ALTER TABLE app_pedidos_2009 OWNER TO app;
ALTER TABLE app_pedidos_2010 OWNER TO app;
ALTER TABLE app_pedidos_2011 OWNER TO app;
ALTER TABLE app_pedidos_2012 OWNER TO app;
ALTER TABLE app_pedidos_2013 OWNER TO app;

```

```

COMMENT ON TABLE app_pedidos_2008 IS 'Tabela de pedidos foo';
COMMENT ON TABLE app_pedidos_2009 IS 'Tabela de pedidos foo';
COMMENT ON TABLE app_pedidos_2010 IS 'Tabela de pedidos foo';
COMMENT ON TABLE app_pedidos_2011 IS 'Tabela de pedidos foo';
COMMENT ON TABLE app_pedidos_2012 IS 'Tabela de pedidos foo';
COMMENT ON TABLE app_pedidos_2013 IS 'Tabela de pedidos foo';

```

-- Criar restrição:

```

ALTER TABLE app_pedidos_2008 ADD CONSTRAINT pedido_check_2008 CHECK
(ano_pedido = 2008);
ALTER TABLE app_pedidos_2009 ADD CONSTRAINT pedido_check_2009 CHECK
(ano_pedido = 2009);
ALTER TABLE app_pedidos_2010 ADD CONSTRAINT pedido_check_2010 CHECK
(ano_pedido = 2010);
ALTER TABLE app_pedidos_2011 ADD CONSTRAINT pedido_check_2011 CHECK
(ano_pedido = 2011);
ALTER TABLE app_pedidos_2012 ADD CONSTRAINT pedido_check_2012 CHECK
(ano_pedido = 2012);
ALTER TABLE app_pedidos_2013 ADD CONSTRAINT pedido_check_2013 CHECK
(ano_pedido = 2013);

```

-- Gatilho de Insert

```

CREATE OR REPLACE FUNCTION app_pedido_trigger()
RETURNS TRIGGER AS $$
BEGIN
  IF NEW.ano_pedido = 2008 THEN
    INSERT INTO app_pedidos_2008 VALUES (NEW.*);
  ELSIF NEW.ano_pedido = 2009 THEN
    INSERT INTO app_pedidos_2009 VALUES (NEW.*);
  ELSIF NEW.ano_pedido = 2010 THEN

```

```

INSERT INTO app.pedido_2010 VALUES (NEW.*);
ELSIF NEW.ano_pedido = 2011 THEN
    INSERT INTO app.pedido_2011 VALUES (NEW.*);
ELSIF NEW.ano_pedido = 2012 THEN
    INSERT INTO app.pedido_2012 VALUES (NEW.*);
ELSIF NEW.ano_pedido = 2013 THEN
    INSERT INTO app.pedido_2013 VALUES (NEW.*);
ELSE
    RAISE EXCEPTION 'Data fora do intervalo permitido.';
END IF;
RETURN NULL;
END;
$$
LANGUAGE plpgsql;

```

```

CREATE TRIGGER insert_pedido_trigger
BEFORE INSERT ON app_pedidos
FOR EACH ROW EXECUTE PROCEDURE app_pedido_trigger();

```

-- Regra de Insert

```

CREATE RULE pedido_2008_insert AS ON INSERT TO app_pedidos
WHERE (ano_pedido = 2008)
DO INSTEAD INSERT INTO app_pedidos_2008 VALUES (NEW.*);

```

```

CREATE RULE pedido_2009_insert AS ON INSERT TO app_pedidos
WHERE (ano_pedido = 2009)
DO INSTEAD INSERT INTO app_pedidos_2009 VALUES (NEW.*);

```

```

CREATE RULE pedido_2010_insert AS ON INSERT TO app_pedidos
WHERE (ano_pedido = 2010)
DO INSTEAD INSERT INTO app_pedidos_2010 VALUES (NEW.*);

```

```

CREATE RULE pedido_2011_insert AS ON INSERT TO app_pedidos
WHERE (ano_pedido = 2011)
DO INSTEAD INSERT INTO app_pedidos_2011 VALUES (NEW.*);

```

```

CREATE RULE pedido_2012_insert AS ON INSERT TO app_pedidos
WHERE (ano_pedido = 2012)
DO INSTEAD INSERT INTO app_pedidos_2012 VALUES (NEW.*);

```

```

CREATE RULE pedido_2013_insert AS ON INSERT TO app_pedidos
WHERE (ano_pedido = 2013)
DO INSTEAD INSERT INTO app_pedidos_2013 VALUES (NEW.*);

```

-- Regra X Gatilho

-- Utilizando REGRA

```
\timing
```

```
INSERT INTO app.cliente (id_cliente, nome_cliente) VALUES (1, 'Riba');
INSERT INTO app.vendedor (id_vendedor, nome_vendedor) VALUES (1, 'João');
```

```
INSERT INTO "pedidos_2008" VALUES (2017, 1, '2017-09-22 11:09:32.57', 1, 1, 'A',
'sdsadas');
```

Time: 3819,447 ms

-- Utilizando GATILHO

```
INSERT INTO app_pedidos SELECT 2008, nextval('pedido_id_pedido_seq'), s.a, 1, 1
FROM generate_series('2008-01-01'::timestamp, '2008-12-31'::timestamp, '2
minutes') AS s(a);
```

INSERT 0 0

Time: 10869,557 ms

-- Usando Explain

```
EXPLAIN ANALYZE SELECT * FROM app_pedidos;
QUERY PLAN
```

```
Append (cost=0.00..114.00 rows=5401 width=62) (actual time=0.006..0.007 rows=1
loops=1)
```

```
    -> Seq Scan on pedidos (cost=0.00..0.00 rows=1 width=62) (actual time=0.001..0.001
rows=0 loops=1)
```

```
        -> Seq Scan on pedidos_2009 (cost=0.00..19.00 rows=900 width=62) (actual
time=0.000..0.000 rows=0 loops=1)
```

```
        -> Seq Scan on pedidos_2010 (cost=0.00..19.00 rows=900 width=62) (actual
time=0.001..0.001 rows=0 loops=1)
```

```
        -> Seq Scan on pedidos_2011 (cost=0.00..19.00 rows=900 width=62) (actual
time=0.000..0.000 rows=0 loops=1)
```

```
        -> Seq Scan on pedidos_2012 (cost=0.00..19.00 rows=900 width=62) (actual
time=0.000..0.000 rows=0 loops=1)
```

```
        -> Seq Scan on pedidos_2013 (cost=0.00..19.00 rows=900 width=62) (actual
time=0.000..0.000 rows=0 loops=1)
```

```
        -> Seq Scan on pedidos_2008 (cost=0.00..19.00 rows=900 width=62) (actual
time=0.004..0.005 rows=1 loops=1)
```

Planning time: 0.400 ms

Execution time: 0.057 ms

(10 rows)

Time: 0,821 ms

Referências:

<https://savepoint.blog.br/2013/01/09/particionamento-de-tabelas-no-postgres-como/> do Telles

Trabalhando com TableSpace no PostgreSQL

Na criação do cluster inicial são criados dois tablespaces, um para o catálogo (pg_global) e outro para os bancos de dados e demais objetos (pg_default).

Os tablespaces são muito usados para balancear cargas, separando objetos em unidades de discos diferentes.

TableSpaces no PostgreSQL permitem aos administrador definir locações no sistema de arquivos para armazenar os objetos de bancos de dados.

O administrador pode controlar o layout do disco. Um tablespace pode ser criado em uma partição diferente e em um disco diferente. Permite ao administrador otimizar a performance. Um índice que é usado de forma pesada pode ser alojado em um disco mais rápido ou com alta disponibilidade. Já tabelas com pouco uso e relevância pode ficar em disco mais barato.

Vale lembrar que se perder o tablespace perderá o acesso a todos os dados dos bancos dele.

```
CREATE TABLESPACE nome_do_espaco_de_tabelas [ OWNER nome_do_usuario ]
LOCATION 'diretório';
```

```
CREATE TABLESPACE indexspace OWNER genevieve LOCATION '/data/indexes';
```

A locação indicada na criação do tablespace precisa existir, precisamos criar o respectivo diretório antes e ainda precisa também ter o super usuário que a criou como seu dono. Todos os objetos criados no tablespace devem ser armazenados em diretórios dentro do diretório indicado no location.

A locação não pode ser em discos removíveis nem em storages transitórios.

Somente super usuários do postgresql podem criar tablespaces. Para que usuários possam acessar precisarão do privilégio CREATE.

Idealmente devemos ter um disco apenas para dados e um disco apenas para índices para evitar concorrência.

Criar tabespace

```
CREATE TABLESPACE ts_tables location '/usr/local/postgresql/ts_tables';
```

Criar Banco em Tablespace

```
CREATE DATABASE intranet TEMPLATE=template0 TABLESPACE=ts_tables;
```

Criar tabela em certo tablespace:

```
CREATE TABLE lotes (id serial) TABLESPACE ts_tables;
```

Setar ts como default

```
SET default_tablespace = space1;
```

Criar tabela no ts default

```
CREATE TABLE foo(i int);
```

1 - Recuperação do identificador de objeto e dos dados dos tablespaces do servidor.

```
SELECT oid, * FROM pg_tablespace;
```

2 - Recuperando o Tablespace e o usuário que tem permissão de owner do tablespace

```
SELECT T.oid, T.spcname, T.spcowner, U.usename
FROM pg_tablespace T, pg_user U
WHERE T.spcowner = U.usesysid;
```

— verifica se as tablespaces foram criadas

```
SELECT spcname AS "Tablespace",
pg_size.pretty(pg_tablespace_size(spcname)) AS "Tamanho",
spclocation as "Caminho"
FROM pg_tableSpace;
```

— Gera Script para alterar tabelas

```
SELECT 'ALTER TABLE' ,n.nspname AS schemaname,'.', c.relname AS tablename, 'SET
TABLESPACE banco_data;'
FROM pg_class c
LEFT JOIN pg_namespace n ON n.oid = c.relnamespace
LEFT JOIN pg_tablespace t ON t.oid = c.reltablespace
WHERE c.relkind = 'r'::"char"
AND nspname NOT IN
('dbatest','information_schema','pg_catalog','pg_temp_1','pg_toast','postgres','publico','pu
blic')
ORDER BY n.nspname
```

— Confere alteracao das tabelas

```
SELECT n.nspname AS schemaname, c.relname AS tablename, t.spcname AS
"Tablespace"
FROM pg_class c
LEFT JOIN pg_namespace n ON n.oid = c.relnamespace
LEFT JOIN pg_tablespace t ON t.oid = c.reltablespace
WHERE c.relkind = 'r'::"char"
AND nspname NOT IN
('dbatest','information_schema','pg_catalog','pg_temp_1','pg_toast','postgres','publico','pu
blic')
ORDER BY n.nspname, c.relname
```

— Verifica tabelas sem tablespace

```

SELECT n.nspname AS schemaname, c.relname AS tablename, t.spcname AS
"Tablespace"
FROM pg_class c
LEFT JOIN pg_namespace n ON n.oid = c.relnamespace
LEFT JOIN pg_tablespace t ON t.oid = c.reltablespace
WHERE c.relkind = 'r'::"char"
AND nspname NOT IN
('dbateste','information_schema','pg_catalog','pg_temp_1','pg_toast','postgres','publico','pu
blic')
AND t.spcname IS NULL
ORDER BY t.spcname DESC

```

— Verifica tamanho da tablespace

```

SELECT spcname AS "Tablespace",
pg_size.pretty(pg_tablespace_size(spcname)) AS "Tamanho",
spclocation as "Caminho"
FROM pg_tableSpace;
ALTERANDO OS INDICES

```

— Cria TableSpace

```

CREATE TABLESPACE "banco_idx" OWNER postgres LOCATION
'/postgres/pg825/dados/pg_tblspc/banco_idx';

```

— verifica se as tablespaces foram criadas

```

SELECT spcname AS "Tablespace",
pg_size.pretty(pg_tablespace_size(spcname)) AS "Tamanho",
spclocation as "Caminho"
FROM pg_tableSpace;

```

— Verifica quais sao os indices (Nao primarios) e o tamanho

```

SELECT n.nspname AS schemaname,c.relname AS tablename,
c.relpages::numeric * 4.096 / 1024::numeric AS espaco_mb
FROM pg_class c
LEFT JOIN pg_namespace n ON n.oid = c.relnamespace
LEFT JOIN pg_tablespace t ON t.oid = c.reltablespace
LEFT JOIN pg_index x ON x.indexrelid = c.oid
WHERE c.relkind = 'i'::"char"
AND x.indisprimary != 't'
AND x.indisunique != 't'
AND nspname NOT IN
('dbateste','information_schema','pg_catalog','pg_temp_1','pg_toast','postgres','publico','pu
blic')
ORDER BY n.nspname

```

— Gera Script para alterar indices

```

SELECT 'ALTER INDEX', n.nspname AS schemaname , '.' ,c.relname AS tablename, 'SET
TABLESPACE banco_idx;'
FROM pg_class c
LEFT JOIN pg_namespace n ON n.oid = c.relnamespace
LEFT JOIN pg_tablespace t ON t.oid = c.reltablespace

```

```

LEFT JOIN pg_index x ON x.indexrelid = c.oid
WHERE c.relkind = 'i'::"char"
AND x.indisprimary != 't'
AND x.indisunique != 't'
AND nspname NOT IN
('dbateste','information_schema','pg_catalog','pg_temp_1','pg_toast','postgres','publico','public')
ORDER BY n.nspname

```

— Confere alteracao dos indices

```

SELECT n.nspname AS schemaname ,c.relname AS tablename,t.spcname AS
"Tablespace"
FROM pg_class c
LEFT JOIN pg_namespace n ON n.oid = c.relnamespace
LEFT JOIN pg_tablespace t ON t.oid = c.reltablespace
LEFT JOIN pg_index x ON x.indexrelid = c.oid
WHERE c.relkind = 'i'::"char"
AND x.indisprimary != 't'
AND x.indisunique != 't'
AND nspname NOT IN
('dbateste','information_schema','pg_catalog','pg_temp_1','pg_toast','postgres','publico','public')
ORDER BY n.nspname

```

— Verifica indice sem tablespace

```

SELECT n.nspname AS schemaname ,c.relname AS tablename,t.spcname AS
"Tablespace"
FROM pg_class c
LEFT JOIN pg_namespace n ON n.oid = c.relnamespace
LEFT JOIN pg_tablespace t ON t.oid = c.reltablespace
LEFT JOIN pg_index x ON x.indexrelid = c.oid
WHERE c.relkind = 'i'::"char"
AND x.indisprimary != 't'
AND x.indisunique != 't'
AND nspname NOT IN
('dbateste','information_schema','pg_catalog','pg_temp_1','pg_toast','postgres','publico','public')
AND t.spcname IS NULL
ORDER BY t.spcname DESC

```

— Verifica tamanho da tablespace

```

SELECT spcname AS "Tablespace",
pg_size_pretty(pg_tablespace_size (spcname)) AS "Tamanho",
spclocation as "Caminho"
FROM pg_tableSpace;

```

Espero que tenha ajudado

Kenia Milene

Verificação das tablespaces existentes:

```
SELECT spcname AS Tablespace, pg_size.pretty(pg_tablespace_size(spcname)) AS
Tamanho, spclocation AS Caminho
FROM pg_tableSpace;
```

— Criando Banco de dados vinculando os tablespace de armazenamento do usuário —
— criando banco de dados com collate 'UTF8'

```
CREATE DATABASE zeus
WITH OWNER = zeus
ENCODING = 'UTF8'
TABLESPACE = tbs_zeustab
CONNECTION LIMIT = -1;
```

— Criando Schemas no PostgreSQL —

```
— Schema: sisimobiliaria
— DROP SCHEMA sisimobiliaria;
CREATE SCHEMA sisimobiliaria
AUTHORIZATION zeus;
GRANT ALL ON SCHEMA sisimobiliaria TO zeus;
```

— Schema: public

```
— DROP SCHEMA public;
CREATE SCHEMA public
AUTHORIZATION zeus;
GRANT ALL ON SCHEMA public TO zeus;
GRANT ALL ON SCHEMA public TO public;
COMMENT ON SCHEMA public
IS 'standard public schema';
```

— Definido os tablespaces:

— Para definir o tablespace, você deve procurar dois pontos importantes no seu dump ou criação: o ponto imediatamente anterior antes de criar as tabelas e o ponto imediatamente anterior a criação dos índices e constraints.

Antes da criação das tabelas coloque a seguinte linha:

```
SET default_tablespace = 'tbs_zeustab';
```

Antes da criação de índices e constraints, coloque a seguinte linha:

```
SET default_tablespace = 'tbs_zeusindx';
```

Listar os tablespaces do cluster

```
select spcname from pg_tablespace;
```

O PostgreSQL faz uso dos links simbólicos para simplificar a implementação dos tablespaces. Isso implica que tablespaces somente podem ser implementados em SO com suporte a links simbólicos.

O diretório \$PGDATA/pg_tblspc contém links simbólicos que apontam para cada um dos tablespaces não nativos definidos no cluster. Não é recomendado mas é possível ajustar manualmente esses links em caso de alterações em diretórios. No PostgreSQL 9.1 e anteriores devemos também necessitar de atualizar o catálogo pg_tablespaces com as novas locações.

Criar banco de dados anexado a tablespace:

```
create database banco with owner usuario tablespace = tbs_dados  
connection limit = -1;
```

Criar esquema definindo outro dono:

```
create schema nomeesquema authorization usuario;
```

Conceder privilégios:

```
grant all on schema nomeesquema to usuario;
```

53 - Criação de grupos de usuários no PostgreSQL 9.6.3

Usando PostgreSQL 9.5.7 no Linux Mint 18.1

Para que um determinado usuário tenha acesso a certo banco, de um certo IP usando um certo método de autenticação precisamos efetuar várias configurações:

- postgresql.conf
- pg_hba.conf (banco, user, ip/rede, método)
- tipo de senha
- Grant e revoke

Por conta disso este é um dos recursos mais trabalhosos do PostgreSQL e que gera mais dúvidas e perguntas e me motivou a criar este guia.

Criarei 6 grupos cada um com um perfil diferente em termos de privilégios

Sendo:

super - com todos os privilégios em todos os bancos do SGBD

admin - com todos os privilégios apenas sobre o banco_um, mas sem privilégio de criar o banco, que será criado por um usuário do grupo super

devel - com todos os privilégios apenas sobre o esquema_um do banco banco_um, mas somente localmente

devel_remote - com todos os privilégios apenas sobre o esquema_um do banco banco_um remotamente (de outros IPs)

manager - com todos os privilégios mas apenas na tabela tabela_um do esquema_um do banco_um

user - apenas com privilégio de consultar/select a tabela_um do esquema_um do banco_um

Estes nomes não são bons, pois não representam uma realidade. É bom quando usamos nomes que representam a realidade de uma empresa ou organização.

Grupos: grp_super, grp_admin, grp_engenharia, grp_comptabilidade, etc

Usuários: user_super1, user_admin1, user_engenharia1, user_comptabilidade1, etc

ou

Usuários: user_joao.brigido, user_pedro.carlos, etc.

No caso, sempre precisarmos anotar a relação de usuários e suas funções/privilégios.

Observações importantes:

- Esquema public - todo banco criado no PostgreSQL tem um esquema chamado public, que dá a acesso a qualquer usuário e inclusive permite que criem objetos no mesmo.

- Acesso remoto - para garantir acesso remoto do banco para certos usuários usando certo IP é feito no pg_hba.conf, especificamente na linha:

```
host    all        all      127.0.0.1/32      md5
        banco     usuário   IP           método de acesso
```

Onde devemos indicar o banco, o usuário, o IP e o método de acesso.

- O parâmetro INHERIT é o default e não precisa ser indicado
- Quando criamos um usuário com "create role", por default o PostgreSQL não concede direito de logar. Se quizermos este direito usemos o alter role.
- Todos os privilégios devem ser concedidos ao grupo e não diretamente ao usuário, assim como também para remover devemos remover do grupo. Os usuários membros do grupos herdarão e assim é realmente mais coerente.
- Alguns poucos privilégios não são herdados, como é o caso de LOGIN e SUPERUSER.

Criar um grupo de super usuários chamado "super"

```
sudo su
su - postgres
psql
CREATE ROLE super WITH LOGIN SUPERUSER PASSWORD 'super';
```

Listar usuários

```
\du
ou
select username from pg_user;
```

Sair do psql

```
\q
```

Tentar logar como super

psql -U super

Reclama que o banco super não existe. Ele tenta conectar com um banco igual ao nome do usuário.

Conectar ao banco postgres

psql -U super -d postgres

psql: FATAL: Peer authentication failed for user "super"

Então editei o pg_hba.conf com o postgres

nano /etc/postgresql/9.5/main/pg_hba.con

Mudei o método da linha abaixo de peer para md5

local	all	all	md5
-------	-----	-----	-----

pg_hba.conf ficou assim:

local	all	postgres	peer
-------	-----	----------	------

# TYPE	DATABASE	USER	ADDRESS	METHOD
--------	----------	------	---------	--------

"local" is for Unix domain socket connections only

```

local  all      all          md5
# IPv4 local connections:
host   all      all  127.0.0.1/32    md5
# IPv6 local connections:
host   all      all ::1/128       md5

exit

/etc/init.d/postgresql reload
su - postgres

```

Tentei novamente:

psql -U super -d postgres
 Agora ele pede a senha e consegue efetuar o login:

Criar usuário super1 pertencente ao grupo super usando o próprio usuário super

```

\q
psql -U super -d postgres
create role super1 in role super password 'super1';

```

\dg
 Veja que super1 mesmo pertencendo ao grupo/role super, não pode fazer login

alter role super1 login; -- O WITH é opcional

```

\dg
Agora ele pode fazer login. Então testemos:
\q

```

psql -U super1 -d postgres
 Consegue logar, mas veja que ele não é super usuário (o prompt é este =>), mesmo tendo sido criado no grupo do super não herdou este priviléio.

Então usemos o super para transformá-lo em super usuário

```

\q
psql -U super -d postgres

```

alter role super1 superuser;
\dg

Agora ele é super user.

```

\q
psql -U super1 -d postgres
Agora sim, ele é super usuário, com todos os poderes (prompt =#, ao invés de =>).

```

Agora vou conectar através do adminer via web, que é um cliente para vários SGBDs, inclusive PostgreSQL:

<http://localhost/adminer.php>

System - PostgreSQL
Server - localhost
Username - super1
Password - super1

Conecta com sucesso. Inclusive removi um banco teste que eu tinha e criei novamente.

Veja no pg_hba que não existe nenhum comando liberando acesso remoto, nenhum IP, exceto o 127.0.0.1, portanto o super1 não deve acessar remotamente, mesmo que o outro computador esteja na mesma rede deste. Vamos testar. Estou em uma rede WIFI e vou testar em outra máquina da mesma.

Tentei acessar esta máquina (192.168.25.11 da máquina 192.168.25.10). Não consegue conectar, conexão recusada. Pergunta:

"O serve está rodando no host 192.168.25.11 e aceita conexão tipo TCP/IP na porta 5432?"

Realmente não, só aceita conexão socket unix (via psql no terminal) ou via localhost/127.0.0.1.

Alerta: é uma iniciativa arriscada em termos de segurança liberar o acesso remoto para um suer usuário. Para acesso remoto liberemos apenas usuários com privilégios restritos, apenas os privilégios necessários e simples como consultas ao banco. Evitemos algo como criar objetos e usuários, apenas manipular registros e ainda assim de forma otimizada, somente o que o usuário realmente precisa. Se podemos fazer assim, por que não fazer?

Obs.: por conta da segurança devemos evitar gerenciar o SGBD com o super usuário ou com um super usuário. É mais seguro para isso criar um usuário com algumas restrições como:

```
CREATE ROLE administrador WITH LOGIN CREATEROLE CREATEDB PASSWORD 'administrador';
```

Criar o segundo grupo, o admin

```
psql -U super1 -d postgres
create role admin with INHERIT LOGIN password 'admin';
\q
psql -U admin -d postgres
```

Consegue logar pelo psql

Veja pelo prompt (=>) que é usuário comum. Não é super usuário.

Tentemos conectar pelo adminer.

Consegue também. Mas tentei remover o banco testes e não consegui. Mas consegui criar uma tabela no esquema public.

Esquema public

Vale lembrar que qualquer usuário pode criar objetos no esquema public, por padrão. Criei uma tabela num banco teste que havia no servidor.

Isso indica que não é interessante manter o esquema public nos bancos que são usados por equipes especialmente, mas não somente.

Eu gosto de remover o esquema public para evitar problemas e confusões.

Logado como super1 remover o esquema public do banco teste existente:

```
psql -U super1 -d postgres
\c testes
drop schema public cascade;
```

Uma alternativa é, ao invés de remover o esquema public, remover todos os privilégios para o público do esquema public:

```
revoke all privileges on database banco_um from public;
```

Tentei criar a tabela novamente com o admin mas agora não consegui.

Testes do super1, se realmente é super usuário: criação de banco e de usuário.

Criar um usuário apenas para teste e um banco também.

```
create role teste1;
create database teste1;
Criou os dois.
```

Agora remover:

```
drop role teste1;
drop database teste1;
```

Removeu ambos, o que caracteriza um super usuário.

Criar o usuário admin1 pertencendo ao grupo admin usando o usuário super1.

```
\q
psql -U super1 -d postgres
```

```
create role admin1 in role admin login password 'admin1';
\du
```

Remover usuário de grupo

```
revoke grupo from usuario;
```

Criar banco_um e tornar o grupo admin seu dono

Este usuário precisa ter todos os poderes sobre o banco_um. Vou criar o banco_um com o usuário super1 e dar todos os privilégios ao admin:

```
create database banco_um owner admin;
\q
psql -U admin1 -d banco_um
```

Conectou e pode criar uma tabela no esquema public.

```
create schema esquema1;
create table esquema1.tabela1(id int);
\d esquema1.tabela1
```

Também conecta pelo adminer e pode criar um esquema e removê-lo. Pode até apagar o banco_um, mas não apaga o testes.

Tentei criar novamente o banco_um mas ele não tem permissão.

Tentei criar uma role mas ele não tem permissão. Tá coerente.

Criar o grupo devel -- com todos os privilégios apenas sobre o esquema_um do banco banco_um, mas somente localmente

```
\q
psql -U super1 -d postgres
create role devel with login password 'devel';
\du
\c banco_um -- conectar ao banco um
create schema IF NOT EXISTS esquema_um AUTHORIZATION devel;

ou
alter schema esquema_um OWNER TO devel;
```

Listar os esquemas criados

```
\dn
```

Para excluir recursivamente:

```
drop schema cascade nome_esquema;
```

Nosso esquema está pronto para ser usado pelo usuário devel1.

Caso criemos a tabela sem usar o prefixo (esquema_um.) ela seria criada no esquema default, que é o public. Como removemos não será criada. Me parece um bom motivo para remover o esquema public.

Removendo o esquema public e tentando criar a tabela dará erro, dizendo que nenhum esquema foi selecionado, o que é bem melhor do que criar e não ser onde desejamos. Uma alternativa é remover privilégios de remoção para o público (PUBLIC) no esquema public:

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

Vou remover o esquema public do banco_um:

```
drop schema public;
```

Criar o usuário devel1 no grupo devel para gerenciar o esquema_um do banco_um.

```
create role devel1 in role devel login password 'devel1';
\du
\q
psql -U devel1 -d banco_um
```

Tentemos criar uma tabela com apenas um campo:

```
create table tabela1(id int);
```

Não consegue pois o esquema public foi removido e não especificamos o esquema_um nem o configuramos como o atual.

Tentemos assim:

```
create table esquema_um.tabela1(id int);
```

Logicamente ele cria.

```
select * from tabela1;
Reclama que tabela1 não existe.
```

```
select * from esquema_um.tabela1;
\d
```

Agora assim:

```
SET search_path TO esquema_um;
create table tabela2(id int);
select * from tabela2;
```

Também cria e consulta, como era de se esperar.

Agora um teste de conexão pelo localhost via web com o adminer.
Conectou sem problema. Cria tabela no esquema_um do banco_um e no esquema public de outros bancos mas não em outros esquemas.

Criar o grupo devel_remote - com todos os privilégios apenas sobre o esquema_umremoto do banco banco_um remotamente (de outros IPs)

```
\q
psql -U super1 -d postgres
create role devel_remote with login password 'devel_remote';
\du
```

```
\c banco_um
create schema esquema_umremoto authorization devel_remote;
ou
alter schema esquema_umremoto owner to devel_remote;
```

Agora precisamos permitir que `devel_remote` acesse o esquema `umremoto` do banco `um` remotamente, ou seja de outro computador/IP.

Mas antes vou criar um usuário no grupo `devel_remote` e dar a este usuário o poder de se conectar remotamente:

Criar o usuário `devel_remote1` no grupo `devel_remote`

```
create role devel_remote1 in role devel_remote login password 'devel_remote1';
\du
\q
psql -U devel_remote1 -d banco_um
create table esquema_umremoto.tabela1(id int);
```

Criou

```
\d esquema_umremoto.tabela1
```

Testando conexão no adminer.

Conectou. Não pode mexer em outros esquemas mas pode criar tabela no esquema `umremoto`, mas não conseguiu criar no esquema `um`.

Habilitar o Acesso Remoto

Para permitir o acesso remoto configuramos o `postgresql.conf` e o `pg_hba.conf`:

No `postgresql.conf` mudar apenas `listen_address`:
`\q`
`nano /etc/postgresql/9.5/main/postgresql.conf`

Onde tem:

```
#listen_addresses = 'localhost'      # what IP address(es) to listen on
```

Mudar para
`listen_addresses = '*' # what IP address(es) to listen on`

Por default ele somente permite acesso no localhost. Veja que a porta usada também está neste arquivo assim como muitas outras configurações.

```
nano /etc/postgresql/9.5/main/pg_hba.conf
exit
/etc/init.d/postgresql restart
su - postgres
```

Observe que a linha atual é:

```
host    all        all        127.0.0.1/32      md
```

Permite todos os usuários acessarem todos os bancos mas apenas localmente.

Vamos adicionar esta linha, logo abaixo da linha acima:

```
host banco_um      devel_remote1      192.168.25.47/32      md5
```

Ficarão assim as duas linhas no pg_hba.conf:

```
host all      all      127.0.0.1/32      md
host banco_um      devel_remote1      192.168.25.47/32      md5
```

Quero que o usuário devel_remote1 possa acessar o banco_um do IP 192.168.25.47 usando autenticação md5.

Ao tentar conectar pelo computador com IP 192.168.25.47 recebo a mensagem:

"Connection refused. Is the server running on host 192.168.25.47 and accepting tcp/ip on port 5432."

Eu estava reiniciando ou recarregando os scripts com o comando
sudo service postgresql reload ou restart

Parece que não estava realizando o que deveria, então restartei o postgresql assim:
/etc/init.d/postgresql restart

Agora ele reclamou que o usuário devel_remote1 no host 192.168.25.47 não tem acesso ao banco postgres no pg_bha.conf.

Então ao criar a conexão com o PgAdmin na máquina remota usei o banco_um ao invés do postgres.

Conectou normalmente.

Liberando acesso remoto para toda uma rede no pg_hba.conf

192.168.25.0/24

\q

Criação do grupo manager - com todos os privilégios mas apenas na tabela tabela_um do esquema_um do banco_um

```
\q
psql -U super1 -d postgres
create role manager login password 'manager';
\du
\c banco_um
create table esquema_um.clientes(id serial primary key, nome char(50) not null, email
char(50), endereco char(100));
\d esquema_um.clientes
```

Criar o usuário manager1 no grupo manager, que terá direitos somente de mexer na tabela clientes

```
create role manager1 in role manager login password 'manager1';
\du
```

Dar permissão total na tabela esquema_um.clientes para o usuário manager1.

```
\c banco_um
```

Antes precisamos dar permissão de acesso ao esquema_um

```
GRANT USAGE ON SCHEMA esquema_um TO manager1;
GRANT ALL ON esquema_um.clientes TO manager1;
```

Testando:

```
\q
psql -U manager1 -d banco_um
\l esquema_um.clientes
Lista a estrutura da tabela
```

Vamos tentar inserir um registro na tabela clientes:

```
insert into esquema_um.clientes values (1, 'Ribamar FS', 'ribafs@gmail.com', 'Rua
Vasco');
```

Inseriu sem problemas.

```
select * from esquema_um.clientes;
```

Algo importante é remover os privilégios default de todos os bancos para que usuários não autenticados não possam acessar.

Uma das opções do postgresql para isso é usar o comando REVOKE e outra é a exclusão do esquema public. Devemos usar ambas, de acordo com o caso.

Criar o grupo usuario - apenas com privilégio de consultar/select a tabela_um do esquema_um do banco_um

Não usei "user", pois é uma palavra reservada do postgresql, que não aceita em nomes de roles.

Assim ele somente acessará o esquema public dos bancos. Precisamos que acesse a tabela_um, do esquema_um, do banco_um.

Vamos criar a tabela_um, no esquema_um, no banco_um usando o super1:

```
\q
psql -U super1 -d banco_um
```

```
create role usuario with login password 'usuario';
\du
\c banco_um
create table esquema_um.tabela_um (codigo int primary key, nome char(50) not null,
endereco char(100));
```

Conceder permissão de acesso ao esquema_um:

```
GRANT USAGE ON SCHEMA esquema_um TO usuario;
```

Agora dar privilégio somente de select nesta tabela. Nada de insert nem outros.

```
GRANT SELECT ON esquema_um.tabela_um TO usuario;
```

```
\q
psql -U usuario -d banco_um
select * from esquema_um.tabela_um;
```

Funcionou.

Agora tentarei inserir um registro:

```
insert into esquema_um.tabela_um values (1, 'Ribamar', 'Rua Vasoco');
```

Permissão negada. Sem a permissão na tabela_um, pois somente o privilégio SELECT foi concedido.

Criar o usuário usuario1 no grupo usuario

```
\q
psql -U super1 -d banco_um
create role usuario1 in role usuario login password 'usuario1';
\du
```

```
revoke all on schema esquema_um from usuario1;
```

Testando

```
\q
psql -U usuario1 -d banco_um
select * from esquema_um.tabela_um;
```

Consultou. Então não basta remover dele a permissão.

Agora para testar, vou remover o privilégio do grupo usuario de SELECT no tabela_um para ver.

```
\q
psql -U super1 -d banco_um
```

```
REVOKE ALL PRIVILEGES ON esquema_um.tabela_um FROM usuario;
```

```
\z esquema_um
\z esquema_um.tabela_um
Veja que o acesso é somente para super1.
```

Testemos:

```
\q
psql -U usuario1 -d banco_um
select * from esquema_um.tabela_um;
```

Sem acesso à tabela_um.

Vamos devolver seu privilégio de SELECT na tabela_um, mas para o grupo usuario, de quem usuario1 herda.

```
\q
psql -U super1 -d banco_um
grant select on esquema_um.tabela_um to usuario;
```

```
\q
psql -U usuario1 -d banco_um
select * from esquema_um.tabela_um;
```

Agora conseguiu.

Isso é bom, precisamos apenas setar as permissões para o grupo e todos os seus membros herdam as mesmas.

Permissões

```
rolename=xxxx -- privileges granted to a role
=xxxx -- privileges granted to PUBLIC
```

```
r -- SELECT ("read")
w -- UPDATE ("write")
a -- INSERT ("append")
d -- DELETE
D -- TRUNCATE
x -- REFERENCES
t -- TRIGGER
X -- EXECUTE
U -- USAGE
C -- CREATE
c -- CONNECT
T -- TEMPORARY
```

```
arwdDxt -- ALL PRIVILEGES (for tables, varies for other objects)
```

```
* -- grant option for preceding privilege
```

```
/yyyy -- role that granted this privilege
```

SET

```
psql
ALTER DATABASE test SET enable_indexscan TO off;

SET search_path TO esquema_um, public;

SET datestyle TO banco_um, dmy;

SET TIME ZONE 'America/Fortaleza';

RESET timezone;
```

SHOW

Mostrar parâmetros em tempo de execução

Mostrar estilo de dadas
SHOW DateStyle;

Mostrar todos os parâmetros do SGBD
SHOW ALL;

\h show

\h set

Referências:

Documentação do PostgreSQL 8 em português - <http://pgdocptbr.sourceforge.net/pg80/>

<http://pgdocptbr.sourceforge.net/pg80/app-psql.html>
<https://www.postgresql.org/docs/9.5/static/app-psql.html>

<http://pgdocptbr.sourceforge.net/pg80/sql-createuser.html>
<https://www.postgresql.org/docs/9.5/static/sql-createrole.html> (Create Role aparece na versão 8.1)

<http://pgdocptbr.sourceforge.net/pg80/sql-alterdatabase.html>
<https://www.postgresql.org/docs/9.5/static/sql-alterdatabase.html>

<http://pgdocptbr.sourceforge.net/pg80/sql-grant.html>
<https://www.postgresql.org/docs/9.5/static/sql-grant.html>

<http://pgdocptbr.sourceforge.net/pg80/sql-revoke.html>
<https://www.postgresql.org/docs/9.5/static/sql-revoke.html>

<http://pgdocptbr.sourceforge.net/pg80/sql-set.html>
<https://www.postgresql.org/docs/9.5/static/sql-set.html>

<http://pgdocptbr.sourceforge.net/pg80/sql-show.html>
<https://www.postgresql.org/docs/9.5/static/sql-show.html>

Dicas Extras:

Usuários são Globais em todo o Agrupamento de Bancos de Dados

Todos os usuários são globais para todo o agrupamento de bancos de dados. Um usuário pode ter acesso a qualquer banco de dados.

Superusuários

Não estão sujeitos à verificação de permissão. Tem direito de fazer o que bem entender em qualquer banco de dados. Somente um superusuário pode criar usuários. Para criar um superusuário usamos o comando:

`create role nome_user createrole;`

Para que um usuário tenha privilégio de criar bancos de dados devemos conceder assim:

`create role nome_user createdb;`

Grupos

São uma forma lógica de juntar usuários para facilitar o gerenciamento de privilégios.

Neste caso concedemos ou revogamos privilégios para todo o grupo, que fica mais prático. Criar um usuário em um certo grupo:

`create role user in role grupo password 'user';`

GRANT e REVOKE

O comando GRANT concede privilégios específicos para um objeto (tabela, visão, seqüência, banco de dados, função, linguagem procedural, esquema ou espaço de tabelas) para um ou mais usuários ou grupos de usuários. Estes privilégios são adicionados aos já concedidos, se existirem.

A palavra chave PUBLIC indica que os privilégios devem ser concedido para todos os usuários, inclusive aos que vierem a ser criados posteriormente.

Se for especificado WITH GRANT OPTION quem receber o privilégio poderá, por sua vez, conceder o privilégio a terceiros.

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

Listar grupos:

`select groname from pg_group;`

ou

`\dg`

Donos dos Objetos

Quando um objeto do banco de dados é criado é atribuído um dono ao mesmo. O dono é o usuário que executou o comando de criação do objeto. Por padrão somente o dono pode fazer qualquer coisa com o objeto. Para que outros usuários tenham acesso ao objeto o dono precisa conceder os privilégios usando o comando GRANT.

- Para maior segurança, sempre antes de conceder somente os privilégios necessários para um usuário sobre um objeto, remova todos os privilégios sobre o objeto do usuário.

Exemplo:

```
REVOKE ALL ON tabela FROM usuario;
GRANT PRIVILÉGIOS ON tabela TO usuario;
```

- Ao conceder privilégios sobre uma tabela que contém um campo do tipo serial, também precisamos conceder privilégios para a sequência gerada pelo serial.

O comando abaixo mostra a sequência:

```
\d
```

- Remover privilégios de acesso a um esquema para todos os usuários:
revoke create on schema public from public;

- Nenhum usuário tenha acesso a uma tabela:
revoke all on tabela from public;

- Funções e Gatilhos

As funções e os gatilhos permitem que usuários insiram código no servidor que outros usuários podem executar sem conhecer. A única proteção real é um controle rígido sobre quem pode definir funções. Estas funções podem burlar qualquer sistema de controle de acesso. As linguagens de função que permitem este tipo de acesso são consideradas "não confiáveis" (untrusted), e o PostgreSQL somente permite que superusuários criem funções escritas nestas linguagens.

Alguns Privilégios:

CREATE

Para bancos de dados, permite a criação de novos esquemas no banco de dados.

Para esquemas, permite a criação de novos objetos no esquema. Para mudar o nome de um objeto existente é necessário ser o dono do objeto e possuir este privilégio no esquema que o contém.

Para tablespaces, permite a criação de tabelas e índices no espaço de tabelas, e permite a criação de bancos de dados possuindo este espaço de tabelas como seu espaço de tabelas padrão.

CONNECT

Permite ao usuário se conectar ao banco de dados especificados, também serão verificadas as restrições impostas pelo postgresql.conf e pelo pg_hba.conf.

USAGE

Para as linguagens procedurais, permite o uso da linguagem especificada para criar funções nesta linguagem. Este é o único tipo de privilégio aplicável às linguagens procedurais.

Para os esquemas, permite acessar os objetos contidos no esquema especificado (assumindo que os privilégios requeridos para os próprios objetos estejam atendidos). Essencialmente, concede a quem recebe o direito de "procurar" por objetos dentro do esquema.

Para as sequências permite usar o nextval e currval

EXECUTE

Este é o único tipo de privilégio aplicável às funções

ALL PRIVILEGES

Concede todos os privilégios disponíveis de uma só vez. A palavra chave PRIVILEGES é opcional no PostgreSQL, embora seja requerida pelo SQL estrito.

Os privilégios aplicáveis a um determinado tipo de objeto variam de acordo com o tipo de objeto, como pode ser visto acima.

Exemplo de Uso de Usuários e Privilégios

Criação do usuário us_dnocs - super usuário para administrar o banco db_intranet

ssh ribamar@10.10.0.60

sudo su
su - postgres
psql

CREATE ROLE us_dnocs WITH LOGIN SUPERUSER PASSWORD 'abcd1029@';

Listar usuários
\du

Sair do psql
\q

Conectar ao banco postgres (precisamos indicar um banco, pois não existe o banco user_dnocs)

psql -U us_dnocs -d postgres

Agora ele pede a senha e consegue efetuar o login

Agora irei criar um grupo de usuários/role chamado "gr_intranet" que se destinará a criação dos usuários para cada esquema do db_intranet

Será criado com o usuário us_dnocs e não precisamos dar senha a ele, pois não faremos login, apenas com os que serão criados através dele

```
\q
```

```
psql -U us_dnocs -d postgres
```

```
create role gr_intranet WITH LOGIN;
```

```
\q
```

Criação do banco de dados db_intranet pelo usuário us_dnocs e tornando o grupo gr_intranet seu dono;

```
psql -U us_dnocs -d postgres
```

```
create database db_intranet owner gr_intranet;
```

```
\l
```

Acessar o db_intranet com gr_intranet

```
psql -U gr_intranet -d db_intranet
```

```
\q
```

Este user pode criar qualquer objeto no banco db_intranet.

Remover o esquema public do db_intranet para evitar confusões e tornar o mesmo mais seguro.

```
psql -U us_dnocs -d db_intranet
```

```
drop schema public;
```

Criar um usuário chamado us_testes para gerenciar o esquema sc_testes e ser seu dono. Este usuário pertencerá ao grupo gr_intranet e será criado pelo usuário us_dnocs

```
psql -U us_dnocs -d postgres
```

```
create role us_testes in role gr_intranet login password 'senhaforte';
```

```
\du
```

```
\q
```

Tornar o us_testes o dono do sc_testes

```
psql -U us_dnocs -d db_intranet
```

```
create schema sc_testes AUTHORIZATION us_testes;
```

```
GRANT ALL ON SCHEMA sc_testes TO us_testes;
```

```
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA sc_testes TO us_testes;
```

```
GRANT USAGE ON ALL SEQUENCES IN SCHEMA sc_testes TO us_testes;
```

Ficar atento, pois talvez seja necessário executar o último comando após importar o script.

Caso ao executar algum comando do cake para cadastrar algo no banco e aparecer a mensagem de que o banco não tem nenhuma tabela execute os dois últimos comandos acima.

Acessar o sc_testes com us_testes

```
psql -U us_testes -d db_intranet
```

Tornar o schema sc_testes o default.

Podemos criar objetos aqui mas especificando o esquema como prefixo. Ex:

```
create table sc_testes.tabela1(campo1 int);
```

```
SET search_path TO sc_testes;
```

Agora podemos criar e excluir sem especificar o esquema, pois estamos nele:

```
create testes.tabela1(campo1 int);
```

Liberando, como root, acesso para o servidor de arquivos, que está no IP 172.16.5.15

```
\q
exit
nano /var/lib/pgsql/9.4/data/pg_hba.conf
```

Precisamos ter uma linha assim:

```
host db_intranet all 172.16.5.15/32 md5
```

E reiniciar o postgresql

```
service postgresql-9.4 restart
```

Tornei meu usuário ribamar_sousa superuser para poder acessar todos os bancos do meu desktop.

alter role ribamar superuser;

54 - Conectividade

O PostgreSQL oferece conectividade com as principais linguagens e ferramentas que utilizam SGBDs.

Existem somente duas interfaces clientes incluídas na distribuição do PostgreSQL:

- libpq – é a interface primária para a linguagem C. Muitas outras interfaces são construídas sobre esta (<http://www.postgresql.org/docs/8.3/interactive/libpq.html>).
- ecpg – SQL incorporado ao PostgreSQL.

Artigo sobre ecpg - <http://www.vivaolinux.com.br/artigos/impressora.php?codigo=4652>

Todas as outras interfaces são projetos externos e são distribuídos separadamente.

Interfaces Clientes Mantidos Externos

Name	Language	Comments	Website
DBD::Pg	Perl	Perl DBI driver	http://search.cpan.org/dist/DBD-Pg/
JDBC	JDBC	Type 4 JDBC driver	http://jdbc.postgresql.org/
libpqxx	C++	New-style C++ interface	http://pqxx.org/
Npgsql	.NET	.NET data provider	http://npgsql.projects.postgresql.org/
ODBCng	ODBC	An alternative ODBC driver	http://projects.commandprompt.com/public/odbcng/
pgtclng	Tcl		http://pgfoundry.org/projects/pgtclng/
psqlODBC	ODBC	The most commonly-used ODBC driver	http://psqlodbc.projects.postgresql.org/
psycopg	Python	DB API 2.0-compliant	http://www.initd.org/

- 1) PHP
- 2) Java
- 3) Windows ODBC
- 4) Visual C++
- 5) Visual BASIC
- 6) C#
- 7) VB.NET

12.1) Integração com o PHP

Para a integração do PHP com o PostgreSQL não há necessidade de se instalar nenhum driver externo, ela acontece usando a libpq do PostgreSQL. A não ser que estejamos usando no Windows poderá acontecer da conexão não estar habilitada, então basta descomentar no php.ini a linha:

```
extension=php_pgsql.dll
```

A conexão da linguagem PHP com o SGBD é efetuada através da função:

`pg_connect()` - http://www.php.net/manual/pt_BR/function.pg-connect.php

Exemplo:

```
<?php
$dbh = new PDO("pgsql:dbname=$dbname;host=$host", $dbuser, $dbpass);
?>
```

http://php.net/manual/pt_BR/ref pdo-pgsql.connection.php

Obs.: Lembrando que uma conexão ao PostgreSQL é realizada com um único banco.

Após a conexão podemos manipular a codificação de caracteres com as funções:

```
pg_set_client_encoding("UNICODE");
echo pg_client_encoding();
echo pg_set_client_encoding();
```

2) Integração com Java (jdbc)

Configuração do Java em Windows

No prompt:

doskey.com /insert

No painel de controle

Adicionar ao path:

c:\jsdk\bin;c:\jsdk\jre\bin;%PATH%

Criar a variável chasspath

.;c:\jsdk\lib\tools.jar;c:\jsdk\lib\dt.jar;c:\jsdk\lib\htmlconverter.jar;c:\jsdk\jre\lib;c:\jsdk\jre\lib\rt.jar;c:\jsdk\jre\lib\ext\postgresql-8.3-603.jdbc3

No Linux (Ubuntu)

Para saber qual versão está usando, qual seu path e alterar, se for o caso:

sudo update-alternatives --config java

Então copie o driver jdbc para o diretório, por exemplo (postgresql-8.3-603.jdbc3.jar):

/usr/lib/jvm/java-6-sun/jre/lib/ext

O driver para integrar aplicações em Java ao PostgreSQL é o JDBC.

Para selecionar a versão correta do JDBC devemos ter em mãos a versão do PostgreSQL e a versão do Java a usar e visitar o site

<http://jdbc.postgresql.org/download.html#jdbcselection>

A versão for Windows já traz o respectivo JDBC. Na dúvida:

- JDK 1.1 - JDBC 1. Note que com a versão 8.0 o suporte ao JDBC 1 foi removido
 - JDK 1.2, 1.3 - JDBC 2.
 - JDK 1.3 + J2EE - JDBC 2 EE. Contém adicional suporte para classes javax.sql.
 - JDK 1.4, 1.5 - JDBC 3. Contém suporte para SSL e javax.sql, mas não requer J2EE
 - JDK 1.6 - JDBC4. Suporte para métodos JDBC4 é limitado.

Para a Versão Atual (8.3) do PostgreSQL

[JDBC3 Postgresql Driver, Version 8.3-603](#) (preferir este, mais maduro)

[JDBC4 Postgresql Driver, Version 8.3-603](#)

Pequenos exemplos em Java acessando banco PostgreSQL através do JDBC:

Para rodar os exemplos abaixo precisamos ter o J2SE (JDK) instalado.

Comando para compilar:

```
javac nome.java
```

Executando:

```
java nome.class
```

Este exemplo apenas testa a conexão com o servidor e o banco de dados.

Apenas copie o driver JDBC para o diretório `\jre\lib\ext` do Java, crie um arquivo chamado `jdbc_teste.java` com este conteúdo:

```
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.SQLException;
public class jdbc_teste2 {
    public static void main(String[] argv) {
        System.out.println("Checando se o Driver está registrado com DriverManager.");
        try {
            Class.forName("org.postgresql.Driver");
        } catch (ClassNotFoundException cnfe) {
            System.out.println("Driver não encontrado!");
            System.out.println("Deixe mostrar o relatório do que encontrei e sair.");
            cnfe.printStackTrace();
            System.exit(1);
        }
        System.out.println("Driver Registrado corretamente, tentarei uma connection.");
    }
}
```

```

Connection c = null;

try {
    // The second and third arguments are the username and password,
    // respectively. They should be whatever is necessary to connect
    // to the database.

    c = DriverManager.getConnection("jdbc:postgresql://localhost:5433/dba_projeto",
"postgres", "postgres");

} catch (SQLException se) {
    System.out.println("Erro ao conectar: imprimindo o resultado e saindo.");
    se.printStackTrace();
    System.exit(1);
}

if (c != null)
    System.out.println("Conexão bem sucedida ao banco!");
else
    System.out.println("Nunca deve ver isso.");
}
}

```

Agora outro arquivo que mostra os registros de uma tabela

Crie um arquivo jdbc_teste2.java com o conteúdo abaixo:

```

import java.sql.*;

public class jdbc_teste {
    public static void main(String args[]) {
        String url = "jdbc:postgresql://localhost:5433/dba_projeto";
        Connection con;
        String query = "select * from clientes";
        Statement stmt;
    }
}

```

```
try {  
    Class.forName("org.postgresql.Driver");  
} catch(java.lang.ClassNotFoundException e) {  
    System.err.print("ClassNotFoundException: ");  
    System.err.println(e.getMessage());  
}  
  
try {  
    con = DriverManager.getConnection(url,"postgres", "postgres");  
    stmt = con.createStatement();  
    ResultSet rs = stmt.executeQuery(query);  
    ResultSetMetaData rsmd = rs.getMetaData();  
    int numberOfColumns = rsmd.getColumnCount();  
    int rowCount = 1;  
    while (rs.next()) {  
        System.out.println("Cliente " + rowCount + ": ");  
        for (int i = 1; i <= numberOfColumns; i++) {  
            System.out.print(" Campo " + i + ": ");  
            System.out.println(rs.getString(i));  
        }  
        System.out.println("");  
        rowCount++;  
    }  
    stmt.close();  
    con.close();  
}  
  
} catch(SQLException ex) {  
    System.err.print("SQLException: ");  
    System.err.println(ex.getMessage());  
}  
}
```

```
}
```

Mais um pequeno exemplo, consultando uma tabela:

Crie um arquivo com nome jdbc_teste3.java, com o conteúdo das duas páginas seguintes:

```
import java.sql.*;
public class jdbc_teste3 {
    public static void main(String args[])
    {
        String url = "jdbc:postgresql://localhost:5433/dba_projeto";
        System.out.println("-----");
        System.out.println("Esta é a URL: " + url);
        Connection db;
        ResultSet rs;
        //Statement sq_stmt;
        try
        {
            Class.forName( "org.postgresql.Driver" );
        }
        catch ( java.lang.ClassNotFoundException e )
        {
            System.err.print( "ClassNotFoundException: " );
            System.err.println( e.getMessage () );
        }

        System.out.println("Driver do PostgreSQL selecionado. ");

        try
        {
            db = DriverManager.getConnection( url, "postgres", "postgres" );
        }
        catch ( SQLException ex )
```

```
{  
    System.err.println( "SQLException: " + ex.getMessage() );  
}  
  
System.out.println("Conexão aberta.");  
try  
{  
    db = DriverManager.getConnection( url, "postgres", "postgres" );  
    Statement sq_stmt = db.createStatement();  
    String sql_str = "SELECT * FROM clientes";  
  
    rs = sq_stmt.executeQuery(sql_str);  
    while (rs.next())  
    {  
        System.out.println("-----");  
        String cpf = rs.getString("cpf");  
        String nome = rs.getString("nome");  
        String email = rs.getString("email");  
        System.out.println("CPF: " + cpf);  
        System.out.println("Nome: " + nome);  
        System.out.println("Email: " + email + " ");  
    }  
    System.out.println("-----");  
    System.out.println("-----");  
}  
catch ( SQLException ex )  
{  
    System.err.println( "SQLException: " + ex.getMessage() );  
}
```

```
System.out.println("Consulta efetuada. ");
System.out.println("Conexão fechada. ");
System.out.println("-----");
}
}
```

3) Windows ODBC

Integração com MS Access

Para aplicações usadas por um único usuário, o MS Access atende geralmente. Ele tem problemas para diversos usuários simultâneos.

A conexão do MS Access e de muitos outros aplicativos for Windows com o PostgreSQL se dá através do psqlODBC, que pode ser encontrado em:

<http://www.postgresql.org/ftp/odbc/versions/msi/>

Baixar a última versão, descompactar e instalar usando o arquivo psqlodbc.msi ou atualizar uma versão existente com o arquivo upgrade.bat.

Após a instalação teremos um novo driver no ODBC do Windows.

Vamos criar uma Conexão ODBC

Adicionar um banco do PostgreSQL no ODBC

Iniciar – Painel de Controle – Ferramentas Administrativas – ODBC

Adicionar – PostgreSQL ANSI ou UNICODE (depende da codificação do banco)

Database – dba_projeto

Server – localhost

Username – postgres

Password – postgres

Então clique em Test

Observe que em Options existem diversas opções extra.

Clique em Save para guardar e OK.

Criar o Banco

Agora abra o Access e crie um banco de dados em branco chamado dba_projeto.

Clicar com o botão direito na janela bancos de dados (área livre)

Importar

Arquivos do tipo (Selecionar o último - Bancos de dados ODBC())

Clique na Aba acima - Fontes de dados de máquina

Selecione a PostgreSQL30W ou outro nome dado, a que criamos e clique em OK

Selecione uma das tabelas para importar e clique em OK

caso apenas vinculemos, as tabelas continuam no PostgreSQL e podemos usá-la e alterá-las, mas seus registros continuarão na tabela que está no PostgreSQL.

Já importando estamos trazendo uma cópia para o Access, uma cópia estática.

4) Integração com Visual C++

E também .NET e outros for Windows na parte III do Livro "PostgreSQL 8 for Windows".

Também o Capítulo 13 do PostgreSQL Prático:

http://pt.wikibooks.org/wiki/PostgreSQL_Pr%C3%A1tico/Conectividade

5) Conectando com Visual BASIC

CurrentProject.Connection.Execute strSql2

If not linked tables then use something like

Dim cnn as new ADODB.Connection

cnn.Open "DSN=my_dbs_dsn_name" 'or a full PostgreSQL connection string

cnn.Execute strSql2

Outro exemplo:

Criar um DSN ODBC "pgresearch" via ADO e use:

```
Dim gcnResearch As ADODB.Connection
```

```
Dim rsUId As ADODB.Recordset
```

```
' open the database
Set gcnResearch =3D New ADODB.Connection
With gcnResearch
    .ConnectionString =3D "dsn=3Dpgresearch"
    .Properties("User ID") =3D txtUsername
    .Properties("Password") =3D txtPassword
    .Open
End With
```

Conexão com Visual Studio

<http://www.linhadecodigo.com.br/ArtigoImpressao.aspx?id=1687>

Conexão do PostgreSQL com o Java

No DETRAN-CE, os sistemas finalísticos foram desenvolvidos em Java, para testar a conexão do PostgreSQL com o Java podem ser utilizados inúmeros clientes de gerenciamento ou modelagem do PostgreSQL. No exemplo que vou mostrar abaixo, utilizei o driver JDBC. O [driver JDBC](#) a ser utilizado deve estar de acordo com a versão do PostgreSQL, entretanto temos instalado a versão 8.2.4 do banco e nos testes ela só funcionou com o driver [8.1-410.jdbc3](#), quando o correto seria utilizar a versão [8.2-506.jdbc4](#). Ainda estou realizando mais alguns uns testes para entender o que ocorreu. No exemplo abaixo criei uma tabela com dados de livros (id, nome, autor, editor, ano) e me conectei ao postgres para retornar uma consulta simples.

```
// início da aplicação
```

```
import java.sql.*;
public class SQLStatement {
    public static void main(String args[]) {
        String url = "jdbc:postgresql://host:5432/nomedobanco";
        Connection con;
        String query = "select * from nomedoesquema.nomedatabela";
        Statement stmt;
        try {
            Class.forName("org.postgresql.Driver");
        } catch(java.lang.ClassNotFoundException e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
        }
    }
}
```

```

try {
    con = DriverManager.getConnection(url, "login", "senha");
    stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery(query);
    ResultSetMetaData rsmd = rs.getMetaData();
    int numberofColumns = rsmd.getColumnCount();
    int rowCount = 1;
    System.out.println("Cadastro de Livros");
    while (rs.next()) {
        System.out.println("Livro " + rowCount);
        for (int i = 1; i <= numberofColumns; i++) {
            System.out.print(" Campo " + i + ": ");
            System.out.println(rs.getString(i));
        }
        System.out.println("");
        rowCount++;
    }
    stmt.close();
    con.close();
} catch(SQLException ex) {
    System.err.print("SQLException: ");
    System.err.println(ex.getMessage());
}
}
} // fim da aplicação

```

PostgreSQLDirect .NET Data Provider - From CoreLab (CoreLab.PostgreSql)

PostgreSQLDirect .NET is data provider to direct access to PostgreSQL database for the Microsoft .NET Framework and .NET Compact Framework. It completely based on ActiveX Data Objects for the .NET Framework (ADO.NET) technology. ADO.NET provides a rich set of components for creating distributed, data-sharing applications. It is an integral part of the .NET Framework, providing access to relational data, XML, and application data. PostgreSQLDirect .NET data provider can be used in the same way as the SQL Server .NET or the OLE DB .NET Data Provider.

6) Using C#

```

using CoreLab.PostgreSql;
...
PgSqlConnection oPgSqlConn = new PgSqlConnection();
oPgSqlConn.ConnectionString =
    "User ID=myUsername;" +
    "Password=myPassword;" +
    "Host=localhost;" +
    "Port=5432;" +
    "Database=myDatabaseName;" +
    "Pooling=true;" +

```

```

    "Min Pool Size=0;" +
    "Max Pool Size=100;" +
    "Connection Lifetime=0";
oPgSqlConn.Open();

```

7) Using VB.NET

```

Imports CoreLab.PostgreSql
...
Dim oPgSqlConn As PgSqlConnection = New PgSqlConnection()
oPgSqlConn.ConnectionString =
    "User ID=myUsername;" & _
    "Password=myPassword;" & _
    "Host=localhost;" & _
    "Port=5432;" & _
    "Database=myDatabaseName;" & _
    "Pooling=true;" & _
    "Min Pool Size=0;" & _
    "Max Pool Size=100;" & _
    "Connection Lifetime=0"
oPgSqlConn.Open()

```

For more information, see: [PostgreSQLDirect](#) .NET Data Provider. Download [here](#).
Support forms [here](#).

Conectando ao .NET com PostgreSQL

Vamos agora instalar o **.NET Data Provider para PostgreSQL** chamada **Npgsql**. O download do provedor pode ser feito no endereço:

<http://pgfoundry.org/frs/download.php/1408/Npgsql2-MS-Net-bin.zip>

Descompacte o arquivo zipado e abra a pasta principal , dentro dela você verá duas pastas , abra a pasta **bin** e copie os arquivos **Npgsql.dll** e **Mono.Security.dll** para o diretório do projeto de sua aplicação VB .NET.

Após copiar estes arquivos clique com o botão direito do mouse sobre o nome do projeto e selecione a opção **Add Reference**, navegue até o local onde a **Npgsql.dll** foi copiada selecione-a e clique em **OK**.

Agora já temos tudo pronto para usar o **PostgreSQL** no **VB 2005 Express**, e é isso que vou fazer no próximo artigo: [VB 2005 - Acessando o PostGreSQL II](#)

Veja o original para as capturas e mais detalhes:

http://www.macoratti.net/07/11/vbn5_pg1.htm
http://www.macoratti.net/07/11/vbn5_pg2.htm

Openoffice2 Base

Usando o OpenOffice para abrir, editar bancos de dados PostgreSQL, como também criar consultas, formulários e relatórios.

Uma das formas de conectar o OpenOffice ao PostgreSQL é usando um driver JDBC do PostgreSQL.

- Antes devemos ter instalado o OpenOffice com suporte a Java
- Baixe daqui:

<http://jdbc.postgresql.org/download.html#jars>

Para o PostgreSQL 8.1 podemos pegar o JDBC3 -

<http://jdbc.postgresql.org/download/postgresql-8.1-405.jdbc3.jar>

A outra é o driver do próprio OO:

<http://dba.openoffice.org/drivers/postgresql/index.html>

- Abrir o OpenOffice, pode ser até o Writer – Ferramentas – Opções – Java – Class Path – Adicionar Arquivo (indicar o arquivo postgresql-8.0-313.jdbc2.jar baixado) e OK.

- Abrir o OOBase
- Conectar a um banco de dados existente
- Selecionar JDBC - Próximo
- URL da fonte de dados:

jdbc:postgresql://127.0.0.1:5432/bdteste

Classe do driver JDBC:

org.postgresql.Driver

Nome do usuário - postgres

password required (marque, caso use senha)

Concluir

Digitar um nome para o banco do OOBase

Pronto. Agora todas as tabelas do banco bdteste estão disponíveis no banco criado no OOBase.

Também podemos agora criar consulta com assistentes, criar formulários e relatórios com facilidade.

Fonte: http://pt.wikibooks.org/wiki/PostgreSQL_Pr%C3%A1tico/Ferramentas/OpenOffice_Base

55 - Informações Introdutórias sobre Redes de Computadores

Destinadas a Administradores de Bancos de Dados com PostgreSQL

As informações aqui são bem resumidas, com o intuito de simplificar, portanto caso precise de mais detalhes deve procurar alguma publicação mais detalhada, como o Guia Foca Linux ou um bom livro.

Redes – é a ligação de dois ou mais computadores para compartilharem seus recursos: arquivos, periféricos (impressoras, scanners, etc), etc.

Principais Tipos de Redes:

- LAN (Local Area Network) e
- WAN (Wide Area Network)

As redes **LAN**, são redes locais, com dois ou mais computadores interligados localmente.

As redes **WAN** são formadas por duas ou mais redes LAN separadas ao redor do planeta, podendo estar em salas separadas ou em países distantes (como é o caso da Internet ou de redes de grandes multinacionais ao redor do planeta).

Vários são os elementos que fazem parte de uma rede. Vejamos alguns destes conceitos.

HUBs -Concentrador (também chamado **HUB**) em linguagem de [informática](#) é o aparelho que interliga diversas [máquinas](#) (computadores) que pode ligar externamente [redes TAN](#), [LAN](#), [MAN](#) e [WAN](#).

O Hub é indicado para redes com poucos terminais de rede, pois o mesmo não comporta um grande volume de informações passando por ele ao mesmo tempo devido sua metodologia de trabalho por [broadcast](#), que envia a mesma informação dentro de uma rede para todas as máquinas interligadas. Devido a isto, sua aplicação para uma rede maior é desaconselhada, pois geraria lentidão na troca de informações.

Switch - Um **switch**, que pode ser traduzido como **comutador**, é um dispositivo utilizado em [redes de computadores](#) para reencaminhar quadros (ou [tramas](#) em Portugal, e [frames](#) em inglês) entre os diversos nós. Possuem diversas portas, assim como os [concentradores](#) (*hubs*) e a principal diferença entre o comutador e o concentrador é que o comutador segmenta a rede internamente, sendo que a cada porta corresponde um segmento diferente, o que significa que não haverá colisões entre pacotes de segmentos diferentes — ao contrário dos [concentradores](#), cujas portas partilham o mesmo [domínio de colisão](#).

MAC - O **endereço MAC** (do inglês *Medium Access Control*) é o endereço físico da estação, ou melhor, da interface de rede. É um endereço de 48 bits, representado em hexadecimal. O protocolo é responsável pelo controle de acesso de cada estação à rede Ethernet. Este endereço é o utilizado na camada 2 (Enlace) do Modelo OSI.

Exemplo:

00:00:5E:00:01:03

NAT - Em redes de computadores, **NAT**, *Network Address Translation*, também conhecido como *masquerading* é uma técnica que consiste em reescrever os endereços IP de origem de um pacote que passam por um router ou firewall de maneira que um computador de uma rede interna tenha acesso ao exterior (rede pública).

Broadcast - **Broadcast** ou **Radiodifusão** é o processo pelo qual se transmite ou difunde determinada informação, tendo como principal característica que a mesma informação está sendo enviada para muitos receptores ao mesmo tempo. Este termo é utilizado em telecomunicações e em informática.

Em Redes de computadores, um endereço de *broadcast* é um endereço IP (e o seu endereço é sempre o último possível na rede) que permite que a informação seja enviada para todas as máquinas de uma LAN, MAN, WAN e TANS, redes de computadores e sub-redes. A RFC (Request for comments), RFC 919 é a RFC padrão que trata deste assunto.

Roteador (também chamado **router** ou **encaminhador**) é um equipamento usado para fazer a comutação de protocolos, a comunicação entre diferentes redes de computadores provendo a comunicação entre computadores distantes entre si.

Roteadores são dispositivos que operam na camada 3 do modelo OSI de referência. A principal característica desses equipamentos é selecionar a rota mais apropriada para repassar os pacotes recebidos. Ou seja, encaminhar os pacotes para o melhor caminho disponível para um determinado destino.

Tunelamento - Em informática, a definição de **Tunnelling** é a capacidade de criar túneis entre duas máquinas por onde certas informações passam.

Em se tratando de um ramo do protocolo TCP/IP, o SSH e o Telnet, pode-se criar uma conexão entre dois computadores, intermediada por um servidor remoto, fornecendo a capacidade de redirecionar pacotes de dados.

Rede ponto-a-ponto

É um tipo de configuração física de enlaces (links) de comunicação de dados, onde existem apenas dois pontos de dispositivos de comunicação em cada uma das extremidades dos enlaces. Geralmente é utilizado cabeamento Coaxial para realizar essas conexões.

Cliente-servidor é um modelo computacional que separa [clientes](#) e [servidores](#), sendo interligados entre si geralmente utilizando-se uma [rede de computadores](#). Cada instância de um cliente pode enviar requisições de dado para algum dos servidores conectados e esperar pela resposta. Por sua vez, algum dos servidores disponíveis pode aceitar tais requisições, processá-las e retornar o resultado para o cliente. Apesar do conceito ser aplicado em diversos usos e aplicações, a arquitetura é praticamente a mesma.

Uma comunicação é dita **half duplex** (também chamada semi-duplex) quando temos um dispositivo Transmissor e outro Receptor, sendo que ambos podem transmitir e receber dados, porém não simultaneamente, a transmissão tem sentido bidirecional. Durante uma transmissão half-duplex, em determinado instante um dispositivo A será transmissor e o outro B será receptor, em outro instante os papéis podem se inverter. Por exemplo, o dispositivo A poderia transmitir dados que B receberia; em seguida, o sentido da transmissão seria invertido e B transmitiria para A a informação se os dados foram corretamente recebidos ou se foram detectados erros de transmissão. A operação de troca de sentido de transmissão entre os dispositivos é chamada de turn-around e o tempo necessário para os dispositivos chavearem entre as funções de transmissor e receptor é chamado de turn-around time.

Exemplos

-Walk Talkie

-Transmissão de fibra ótica.

Uma comunicação é dita **full duplex** (também chamada apenas duplex) quando temos um dispositivo Transmissor e outro Receptor, sendo que os dois podem transmitir dados simultaneamente em ambos os sentidos (a transmissão é bidirecional). Poderíamos entender uma linha full-duplex como funcionalmente equivalente a duas linhas [simplex](#), uma em cada direção. Como as transmissões podem ser simultâneas em ambos os sentidos e não existe perda de tempo com turn-around (operação de troca de sentido de transmissão entre os dispositivos), uma linha full-duplex pode transmitir mais informações por unidade de tempo que uma linha half-duplex, considerando-se a mesma taxa de transmissão de dados.

Exemplo

Aparelho telefônico; Vídeo Conferência; PCI-Express; Protocolo TCP ([Transmission Control Protocol](#)).

O [cabeamento](#) por **par trançado** (Twisted pair) é um tipo de fiação na qual dois condutores são entrançados um ao redor do outro para cancelar [interferências](#) eletromagnéticas de fontes externas e interferências mútuas ([linha cruzada](#) ou, em inglês, crosstalk) entre cabos vizinhos. A taxa de giro (normalmente definida em termos de giros

por metro) é parte da especificação de certo tipo de cabo. Quanto maior o número de giros, mais o ruído é cancelado. Foi um sistema originalmente produzido para transmissão telefônica analógica que utilizou o sistema de transmissão por par de fios aproveita-se esta tecnologia que já é tradicional por causa do seu tempo de uso e do grande número de linhas instaladas.

Wireless - A tecnologia **wireless** (sem fios) permite a conexão entre diferentes pontos sem a necessidade do uso de cabos - seja ele telefônico, coaxial ou óptico - por meio de equipamentos que usam radiofrequência (comunicação via ondas de rádio) ou comunicação via infravermelho, como em dispositivos compatíveis com IrDA.

Wireless é uma tecnologia capaz de unir terminais eletrônicos, geralmente computadores, entre si devido às ondas de rádio ou infravermelho, sem necessidade de utilizar cabos de conexão entre eles. O uso da tecnologia wireless vai desde transceptores de rádio como walkie-talkies até satélites artificiais no espaço.

Classes de Redes

Classe	Máscara de Rede	Endereço da Rede		
A	255.0.0.0	0.0.0.0	-	127.255.255.255
B	255.255.0.0	128.0.0.0	-	191.255.255.255
C	255.255.255.0	192.0.0.0	-	223.255.255.255
Multicast	240.0.0.0	224.0.0.0	-	239.255.255.255

O tipo de endereço que você deve utilizar depende exatamente do que estiver fazendo.

Referência rápida de máscara de redes

A tabela abaixo faz referência as máscaras de rede mais comuns e a quantidade de máquinas máximas que ela atinge. Note que a especificação da máscara tem influência direta na classe de rede usada:

Máscara (Forma octal)	Máscara (Forma 32 bits)	Número Máximo de Máquinas
Classe A: /8	/255.0.0.0	16,777,215
Classe B: /16	/255.255.0.0	65,535
/17	/255.255.128.0	32,767
/18	/255.255.192.0	16,383
/19	/255.255.224.0	8,191
/20	/255.255.240.0	4,095
/21	/255.255.248.0	2,047
/22	/255.255.252.0	1,023

/23	/255.255.254.0	511
Classe C		
/24	/255.255.255.0	255
/25	/255.255.255.128	127
/26	/255.255.255.192	63
/27	/255.255.255.224	31
/28	/255.255.255.240	15
/29	/255.255.255.248	7
/30	/255.255.255.252	3
/32	/255.255.255.255	1

Qualquer outra máscara fora desta tabela (principalmente para a classe A), deverá ser redimensionada com uma calculadora de IP para chegar a um número aproximado de redes/máquinas aproximados que deseja.

Referência: Guia Foca Linux - <http://focalinux.cipsga.org.br/guia/avancado/ch-rede.html#s-rede-ip-classes>

Endereços reservados para uso em uma rede Privada

Se você estiver construindo uma rede privada que nunca será conectada a Internet, então você pode escolher qualquer endereço que quiser. No entanto, para sua segurança e padronização, existem alguns endereços IP's que foram reservados especificamente para este propósito. Eles estão especificados no RFC1597 e são os seguintes:

ENDEREÇOS RESERVADOS PARA REDES PRIVADAS			
Classe de Rede	Máscara de Rede	Endereço da Rede	
A	255.0.0.0	10.0.0.0	- 10.255.255.255
B	255.255.0.0	172.16.0.0	- 172.31.255.255
C	255.255.255.0	192.168.0.0	- 192.168.255.255

IP - O **endereço IP**, de forma genérica, pode ser considerado como um conjunto de números que representa o local de um determinado **equipamento** (normalmente **computadores**) em uma **rede privada** ou **pública**.

Máscara de subrede – também conhecida como **subnet mask** ou **netmask**. Uma **subrede** é uma divisão de uma rede de computadores.

Class e	Bits iniciais	Início	Fim	Máscara de Subrede padrão	Notação CIDR
A	0	1.0.0.1	126.255.255.254	255.0.0.0	/8
B	10	128.0.0.1	191.255.255.254	255.255.0.0	/16
C	110	192.0.0.1	223.255.255.25	255.255.255.0	/24

Uma rede “classful” é uma rede que possui uma máscara de subrede 255.0.0.0, 255.255.0.0 ou 255.255.255.0.

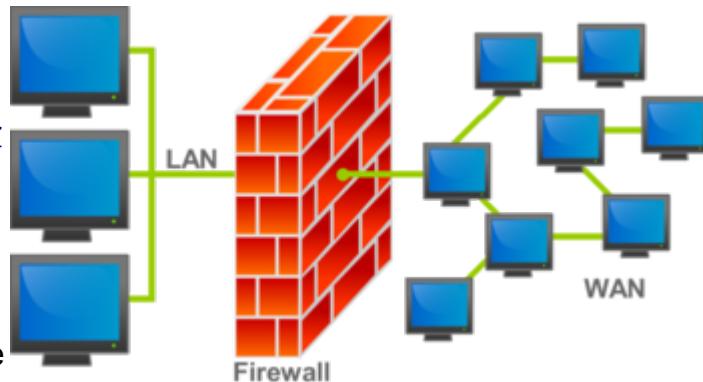
Gatway – Endereço IP do servidor.

DNS – Domain Name System (Sistema de Nomes de Domínios) é um sistema de gerenciamento de nomes hierárquico e distribuído operando segundo duas definições:

- Examinar e atualizar seu banco de dados.
- Resolver nomes de servidores em endereços de rede (IPs).

Firewall é o nome dado ao dispositivo de uma rede de computadores que tem por objetivo aplicar uma política de segurança a um determinado ponto de controle da rede. Sua função consiste em regular o tráfego de dados entre redes distintas e impedir a transmissão e/ou recepção de acessos nocivos ou não autorizados de uma rede para outra. Este conceito inclui os equipamentos de filtros de pacotes e de proxy de aplicações, comumente associados a redes TCP/IP.

Squid - o Squid é um popular servidor Proxy em software livre. Um dos melhores softwares para a função do mercado. Seu uso é variado, ele pode



esconder petições repetidas, esconder www, DNS, e outros recursos de rede compartilhados para um grupo de pessoas. É projetado principalmente para rodar em sistemas UNIX.

IPTables - O netfilter é um módulo que fornece ao sistema operacional Linux as funções de firewall, NAT e log de utilização de rede de computadores.

iptables é o nome da ferramenta do espaço do usuário que permite a criação de regras de firewall e NATs. Apesar de, tecnicamente, o iptables ser apenas uma ferramenta que controla o módulo netfilter, o nome "iptables" é frequentemente utilizado como referência ao conjunto completo de funcionalidades do netfilter. O iptables é parte de todas as distribuições modernas do Linux.

Serviço	Porta						
PostgreSQL	5432						
MySQL	3306						
SSH	22						
FTP	21 e 20						
POP	110						
IMAP	143						
SMTP	25						
TELNET	23						
HTTP	80						
HTTPS	443						

Velocidade de Redes Ethernet

10 MBPs TCP/IP

100 MBPs RFC

1 GBPs INTERNIC (<http://www.internic.com>)

10 GBPs UDP

Máscara Explicada

Apenas para iluminar um pouco, aquele número que vem depois da barra "/" significa o número de bits que ele vai utilizar na máscara. Vejamos.

Suponha que você deixe 189.0.0.0/24, o que vai acontecer?

- 1) Você vai tentar conectar no IP do seu servidor a partir de seu IP de origem IP
- 2) Seu servidor vai pegar o seu IP de origem e fazer um cálculo de máscara usando 24 bits, numa comparação XOR bit-a-bit que vai resultar em 189.22.33.0
- 3) Ele vai pegar o resultado do cálculo acima e comparar com seu arquivo

e vai identificar que 189.22.33.0 NÃO É IGUAL A 189.0.0.0.

Portanto você tem que utilizar 189.0.0.0/8, pois assim ele vai pegar o seu IP IP, vai fazer uma comparacao XOR bit-a-bit e vai ter como resultado 189.0.0.0, com esse resultado ele vai comprar com o 189.0.0.0 e vai reconhecer a IGUALDADE entre eles e vai aceitar.

Em resumo.

IP/8 => IP/255.0.0.0 = 192.0.0.0

IP/16 => IP/255.255.0.0 = 192.168.0.0

IP/24 => IP/255.255.255.0 = 192.168.1.0 - uma rede (256 máquinas)

IP/32 => IP/255.255.255.255 = 192.168.1.12 - uma única máquina

Outras mascaras são possíveis através de deslocamento de bit do parte da rede para o host, obtendo-se sub-redes, mas ai ja acabamos fungindo do escopo da lista.

--

Dickson S. Guedes

