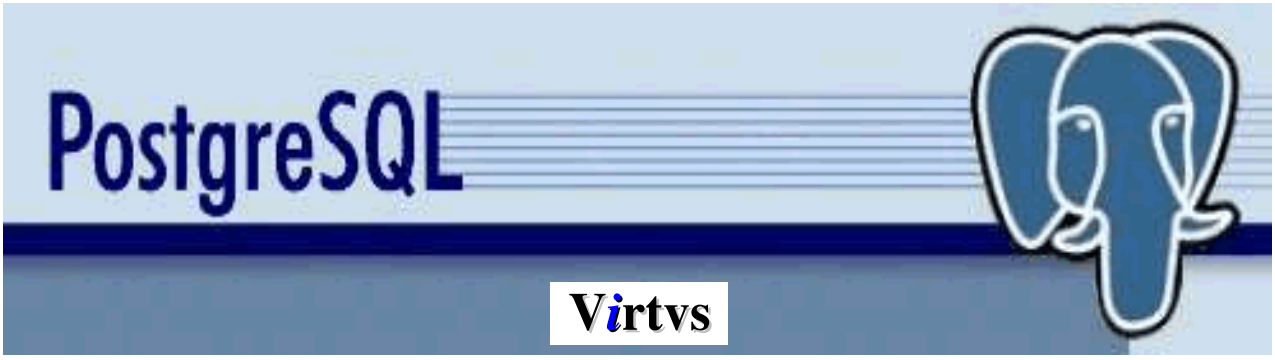
*e*Book



Eng. Adbeel Goes Filho

Edição 200507

.

Objetivo	Capacitar o leitor a instalar / configurar, fazer operações básicas do banco de dados e utilizar as características avançadas do PostgreSQL.
Pré-requisitos	Noções de sistemas operacionais Linux e Windows e banco de dados

Composto e editado eletrônicamente no Núcleo de Informática da
VIRTVS Engenharia e Informática Ltda.
Rua Cel. Ribeiro da Silva, 391. Monte Castelo.
CEP 60325-210. Fortaleza-Ceará.
Telefones +55 - xx - 85 - 32833124 / 32819997.
Email: adbeel@virtvs.com.br adbeel@virtvs.com.br

Este trabalho têm como referência o PostgreSQL versão 7.3.4.

Fonte Principal

The PostgreSQL Global Development Group

Modificações foram realizadas para uso educacional.

Revisões: 20050719, 20050721.

Sumário

Índice de Tabelas

Agradecimentos

- À todos os que trabalham em comunidade para o desenvolvimento do conhecimento humano.
- À todos que, cedendo horas preciosas de suas vidas, promovem a inclusão social.
- Àos diversos autores anônimos, cujos textos foram aqui transcritos.

CAPÍTULO 1. Introdução

O PostgreSQL é um sistema de gerenciamento de banco de dados objeto-relacional (SGBDOR) baseado no POSTGRES, Versão 4.2, desenvolvido no Departamento de Ciência da Computação da Universidade da Califórnia, em Berkeley. O projeto POSTGRES, liderado pelo Professor Michael Stonebraker, foi patrocinado pelas seguintes instituições: Defense Advanced Research Projects Agency (DARPA); Army Research Office (ARO); National Science Foundation (NSF); e ESL, Inc.

O PostgreSQL descende deste código original de Berkeley, possuindo o código fonte aberto. Fornece suporte às linguagens SQL92/SQL99, além de outras funcionalidades modernas.

O POSTGRES foi pioneiro em muitos conceitos objeto-relacionais que agora estão se tornando disponíveis em alguns bancos de dados comerciais. Os Sistemas de Gerenciamento de Bancos de Dados Relacionais (SGBDR) tradicionais suportam um modelo de dados composto por uma coleção de relações com nome, contendo atributos de um tipo específico. Nos sistemas comerciais em uso, os tipos possíveis incluem número de ponto flutuante, inteiro, cadeia de caracteres, monetário e data. É amplamente reconhecido que este modelo não é adequado para aplicações futuras de processamento de dados. O modelo relacional substituiu com sucesso os modelos anteriores em parte devido à sua "simplicidade Espartana". Entretanto, esta simplicidade tornou a implementação de certas aplicações muito difícil. O PostgreSQL oferece recursos adicionais pela incorporação dos conceitos mostrados abaixo, tornando possível os usuários estenderem o sistema facilmente:

- herança
- tipos de dado
- funções

Outras funcionalidades fornecem poder e flexibilidade adicionais:

- restrições
- gatilhos
- regras
- integridade da transação

Estas funcionalidades colocam o PostgreSQL dentro da categoria de banco de dados referida como objeto-relacional. Repare que é diferente daqueles referidos como orientados a objetos que, em geral, não são muito adequados para apoiar as linguagens tradicionais de banco de dados relacional. Portanto, embora o PostgreSQL possua algumas funcionalidades de orientação a objetos, está firmemente ligado ao mundo dos bancos de dados relacionais. Na verdade, alguns bancos de dados comerciais incorporaram recentemente funcionalidades nas quais o PostgreSQL foi pioneiro.

Este manual descreve a utilização da linguagem SQL no PostgreSQL. Começa descrevendo a sintaxe geral do SQL, depois explica como criar estruturas para conter dados, como carregar o banco de dados, e como consultá-lo. A parte intermediária mostra os tipos de dado e funções

disponíveis para uso nos comandos de dados do SQL. O restante do trabalho discorre sobre vários aspectos importantes para ajustar o banco de dados para um desempenho otimizado.

Neste trabalho as informações estão dispostas de tal maneira que um usuário novato poderá seguilas do princípio ao fim, tendo uma compreensão completa dos tópicos, sem que sejam feitas referências à frente muitas vezes. A idéia é que os capítulos sejam autocontidos, permitindo aos usuários avançados a leitura individual de cada capítulo conforme sua necessidade. As informações neste manual estão apresentadas sob forma de narrativa, sendo cada unidade um tópico. Os leitores procurando por uma descrição completa de um determinado comando devem consultar o Manual de Referência do PostgreSQL.

Os leitores deste trabalho devem saber como conectar ao banco de dados PostgreSQL e executar comandos SQL. Os leitores não familiarizados com estes procedimentos são encorajados a ler antes o Tutorial do PostgreSQL. Os comandos SQL são geralmente executados utilizando o terminal interativo do PostgreSQL chamado psql, mas outros programas com funcionalidades equivalentes também podem ser utilizados.

Este manual abrange o PostgreSQL 7.3.4 e observações sobre outras versões. Para obter informações mais detalhadas relacionadas com outras versões, por favor leia a documentação que acompanha a própria versão.

CAPÍTULO 2. Histórico

O ancestral do PostgreSQL foi o Ingres, desenvolvido na Universidade da California / EUA (1977-1985). O código fonte do Ingres depois foi atualizado pela Relational Technologies/Ingres Corporation que produziu o primeiro e bem sucedido servidor de banco de dados comercial. Depois, nesta universidade, o Professor Michael Stonebraker coordenou o projeto de seus alunos para a criação de um servidor de banco de dados orientado a objetos chamado Postgres (1986-1994). Mais tarde, dois estudantes graduados desta universidade, Jolly Chen e Andrew Yu, adicionaram as capacidades da linguagem SQL ao Postgres. O resultado do projeto foi chamado de Postgres95 (1994-1995) que foi mantido e atualizado através de uma lista de mensagens na Internet.

Em 1996 o aumento da demanda de um banco de dados SQL com características *open-source*, motivou a continuidade do desenvolvimento do Postgres95 por alunos de universidades em vários países (Eua, Canadá, Russia). O código fonte já tinha mais de 250 mil linhas escritas na universal "linguagem C". No final de 1996 resolveram mudar o nome do Postgres95 para PostgreSQL e a distribuição inicial foi restrita a programadores autorizados a realizar atualizações no código fonte.

Hoje, o código fonte do PostgreSQL está protegido e registrado como *open-source* (código aberto) pela Universidade da California/EUA, é mantido e atualizado por universitários e programadores experientes e por empresas como a PostgreSQL Inc - que entre muitas outras, oferece suporte comercial a este produto.

Com mais de uma década de desenvolvimento por trás, o PostgreSQL é o mais avançado banco de dados de código aberto disponível em qualquer lugar, oferecendo controle de concorrência multi-versão, suportando praticamente todas as construções do SQL (incluindo subconsultas, transações, tipos definidos pelo usuário e funções), e dispondo de um amplo conjunto de ligações com linguagens procedurais (incluindo C, C++, Java, Perl, Tcl e Python).

O projeto POSTGRES de Berkeley

A implementação do SGBD POSTGRES começou em 1986. Os conceitos iniciais para o sistema foram apresentados em The design of POSTGRES, e a definição do modelo de dados inicial apareceu em The POSTGRES data model. O projeto do sistema de regras nesta época foi descrito em The design of the POSTGRES rules system. Os princípios básicos e a arquitetura do gerenciador de armazenamento foram detalhados em The design of the POSTGRES storage system.

O Postgres passou por várias versões desde então. A primeira "versão de demonstração" do sistema ficou operacional em 1987, e foi exibida em 1988 na Conferência ACM-SIGMOD. A versão 1, descrita em The implementation of POSTGRES, foi liberada para alguns poucos usuários externos em junho de 1989. Em resposta à crítica ao primeiro sistema de regras (A commentary on the POSTGRES rules system), o sistema de regras foi re-projetado (On Rules, Procedures, Caching and Views in Database Systems) e a versão 2 foi liberada em junho de 1990, contendo um novo sistema de regras. A versão 3 surgiu em 1991 adicionando suporte a múltiplos gerenciadores de armazenamento, um executor de consultas melhorado, e um sistema de regras de reescrita reescrito.

Para a maior parte, as versões seguintes até o Postgres95 focaram a portabilidade e a confiabilidade.

O POSTGRES tem sido usado para implementar muitas aplicações diferentes de pesquisa e produção, incluindo: sistema de análise de dados financeiros, pacote de monitoramento de desempenho de turbina a jato, banco de dados de acompanhamento de asteróides, banco de dados de informações médicas, além de vários sistemas de informações geográficas. O POSTGRES também tem sido usado como ferramenta educacional em diversas universidades. Por fim, a Illustra Information Technologies (posteriormente incorporada pela Informix, que agora pertence à IBM) pegou o código e comercializou. O POSTGRES se tornou o gerenciador de dados principal do projeto de computação científica Sequoia 2000 no final de 1992.

O tamanho da comunidade de usuários externos praticamente dobrou durante o ano de 1993. Começou a ficar cada vez mais óbvio que a manutenção do código do protótipo e seu suporte estava consumindo grande parte do tempo que deveria ser dedicado às pesquisas sobre banco de dados. Em um esforço para reduzir esta sobrecarga de suporte, o projeto do POSTGRES de Berkeley terminou oficialmente com a versão 4.2.

O Postgres95

Em 1994, Andrew Yu e Jolly Chen adicionaram um interpretador da linguagem SQL ao POSTGRES. O Postgres95 foi em seguida liberado para a Web para encontrar seu caminho no mundo como descendente de código aberto do código original do POSTGRES de Berkeley.

O código do Postgres95 era totalmente escrito em ANSI C e reduzido em tamanho em 25%. Muitas mudanças internas melhoraram o desempenho e a facilidade de manutenção. O Postgres95 versão 1.0.x era 30-50% mais rápido que o POSTGRES versão 4.2, utilizando o Wisconsin Benchmark. Além da correção de erros, as principais melhorias foram as seguintes:

A linguagem de consultas PostQUEL foi substituída pelo SQL (implementado no servidor). As subconsultas não foram permitidas até o PostgreSQL, mas podiam ser simuladas no Postgres95 por meio de funções SQL definidas pelo usuário. As agregações foram re-implementadas. O suporte para a cláusula GROUP BY das consultas também foi adicionado. A interface da libpq permaneceu disponível para os programas escritos na linguagem C.

Além do programa monitor, um novo programa (psql) foi disponibilizado para consultas SQL interativas utilizando o Readline do GNU.

A nova biblioteca cliente libpgtcl suportava clientes baseados no Tcl. A shell pgtclsh permitia que os novos comandos Tcl interfaceassem as aplicações Tcl e o servidor Postgres95.

A interface para objetos grandes foi refeita. A Inversão de objetos grandes era o único mecanismo para armazená-los (O sistema de arquivos Inversão foi removido).

O sistema de regras no nível de instância foi removido. As regras ainda estavam disponíveis por meio de regras de reescrita.

Um breve tutorial introduzindo as funcionalidades regulares da linguagem SQL, assim como as do Postgres95, foi distribuído junto com o código fonte.

O utilitário make do GNU (no lugar do make do BSD) foi utilizado para a geração. Além disso, o Postgres95 podia ser compilado com o GCC sem correções (o alinhamento de dados para a precisão dupla foi corrigido).

O PostgreSQL

Em 1996 ficou claro que o nome "Postgres95" não resistiria ao teste do tempo. Foi escolhido então um novo nome, PostgreSQL, para refletir o relacionamento entre o POSTGRES original e as versões mais recentes com funcionalidade SQL. Ao mesmo tempo, foi mudado o número da versão para começar em 6.0, colocando os números na seqüência original começada pelo projeto POSTGRES de Berkeley.

A ênfase durante o desenvolvimento do Postgres95 era identificar e compreender os problemas existentes no código do servidor. Com o PostgreSQL, a ênfase foi mudada para a melhoria das funcionalidades e dos recursos, embora o trabalho continuasse em todas as áreas.

As principais melhorias no PostgreSQL incluem:

O bloqueio no nível de tabela foi substituído por um sistema de concorrência multi-versão, permitindo os que estão lendo continuarem a ler dados consistentes durante a atividade de escrita, possibilitando efetuar cópias de segurança utilizando o pg_dump enquanto o banco de dados se mantém disponível para consultas.

A implementação de funcionalidades importantes no servidor, incluindo subconsultas, padrões, restrições e gatilhos.

A incorporação de funcionalidades adicionais compatíveis com a linguagem SQL92, incluindo chaves primárias, identificadores entre aspas, conversão implícita de tipo de cadeias de caracteres literais, conversão explícita de tipos e inteiros binários e hexadecimais.

Os tipos nativos foram aperfeiçoados, incluindo vários tipos para data e hora, e suporte adicional para tipos geométricos.

A velocidade geral do código do servidor foi melhorada em aproximadamente 20-40%, e o tempo de inicialização do servidor foi reduzido em 80% desde que a versão 6.0 foi liberada.

Principais Recursos e Características

Futuro Definido

PostgreSQL - têm uma equipe distribuída de programadores. Grandes novidades são incorporadas ao banco de dados muito rapidamente. Defeitos são corrigidos em questão de horas. A licença BSD garante sua continuidade, e a possibilidade de implementações comerciais originadas do código oficial, o que permite o financiamento dos programadores que oferecem suporte ao banco de dados.

Estabilidade

PostgreSQL - utiliza o padrão ACID [1]. Suas transações são duráveis e atômicas, e não criam travamentos para usuários simultâneos. Em caso de *crash*, possui um avançado sistema de *logging* de transações, que permite que dados não sejam perdidos.

Flexibilidade

PostgreSQL permite flexibilidade na criação de tipos de dados e funções, nas opções de linguagens de programação de funções e stored procedures.

Suporte SQL

PostgreSQL - suporta totalmente o padrão ANSI SQL, incluindo as versões 89, 92 e 98, 2003. Pode executar queries complexas.

Constante Evolução

O PostgreSQL é um produto que está em constante evolução. Diferentemente do Interbase, que desde o ano passado não teve modificações, e ninguém sabe se vai ou não sair uma versão nova, ou se a Borland vai deixar de desenvolvê-lo. O futuro é incerto para o Interbase.

Suporte ao Desenvolvedor

O PostgreSQL possui excelente suporte para qualquer linguagem de programação atual. Possui excelente suporte a Delphi e Kylix, com componentes *open-source* e de código aberto. Possui driver ODBC open source e *runtime free*. Possui suporte às mais variadas linguagens, como C, C++, Delphi, Cobol, Flagship, Visual Basic, PHP, Perl, ASP, Zope, Python, Java, etc. Pode ser utilizada em qualquer sistema operacional que suporte o protocolo TCP/IP.

Portabilidade, Escalabilidade e Suporte a Multiplataforma

O PostgreSQL roda em qualquer plataforma compatível com Unix, como Linux, FreeBSD, OpenBSD, NetBSD, MacOS X, Solaris, HP UX, AIX, BeOS, entre outros. Teve seu código portado nativamente para o ambiente Windows pela dbExperts, com suporte à execução tanto como aplicativo tanto como serviço, com um licenciamento POR SERVIDOR, a um custo extremamente baixo.

O dbExperts PostgreSQL possui suporte à instalação nos sistemas operacionais Windows 95, 98, ME, NT e 2000, XP. Foram desenvolvidos componentes de sistema que permitem o correto funcionamento do PostgreSQL na plataforma Windows, além de melhorias internas e instalação automática. Além disso, foram implementadas rotinas que possibilitam a inicialização como um serviço nativo no Windows NT, 2000, XP e rotinas para a execução como um aplicativo em modo usuário, com controle a partir de um ícone na barra de tarefas no Windows 95/98/ME.

Todas estas melhorias tornam o processo de instalação, configuração e utilização simples e rápido. Não é necessário nenhum conhecimento específico de PostgreSQL para o uso.

Terminologia e notação

Administrador normalmente é a pessoa responsável pela instalação e funcionamento do servidor. **Usuário** pode ser qualquer um usando, ou querendo usar, qualquer parte do sistema PostgreSQL. Estes termos não devem ser interpretados ao pé da letra; este conjunto de documentos não estabelece premissas relativas aos procedimentos do administrador do sistema.

1 ACID (Atomicidade, Consistência, Isolamento, Durabilidade)

É utilizado /usr/local/pgsql/ como sendo o diretório raiz da instalação, e /usr/local/pgsql/data como o diretório contendo os arquivos dos banco de dados. Estes diretórios podem ser outros em sua máquina, os detalhes podem ser vistos no Guia do Administrador [2].

Na sinopse dos comandos os colchetes ([]) indicam uma frase ou palavra chave opcional. Qualquer coisa entre chaves ({}) contendo, também, barras verticais (|), indica que uma das alternativas deve ser escolhida.

Os exemplos mostram comandos executados a partir de várias contas e programas. Os comandos executados a partir de uma "shell" do Unix podem estar precedidos por um caractere de cifrão ("$"). Os comandos executados a partir da conta de um usuário, tal como root ou postgres, são especialmente sinalizados e explicados. Os comandos SQL podem estar precedidos por "=>" ou não ter nada precedendo, dependendo do contexto.

Nota: A notação para sinalizadores de comandos não é totalmente consistente através do conjunto de documentos. Por favor relate os problemas encontrados para a lista de discussão da documentação em <pgsql-docs@postgresql.org>.

Guia para relatar erros

Quando é encontrado algum erro no PostgreSQL nós desejamos conhecê-lo. Seus relatórios de erro são parte importante para tornar o PostgreSQL mais confiável, porque mesmo o mais extremo cuidado não pode garantir que todas as partes do PostgreSQL vão funcionar em todas as plataformas sob qualquer circunstância.

As sugestões abaixo têm por objetivo ajudá-lo a preparar seu relatório de erro, permitindo que este possa ser tratado de uma forma efetiva. Ninguém é obrigado a segui-las, mas são feitas para serem vantajosas para todos.

Nós não podemos prometer corrigir todos os erros imediatamente. Se o erro for óbvio, crítico, ou afetar muitos usuários, existe uma boa chance de alguém olhá-lo. Pode acontecer, também, nós solicitarmos a atualização para uma nova versão, para ver se o erro também acontece na nova versão. Também podemos decidir que o erro não poderá ser corrigido antes de uma grande reescrita que planejamos fazer. Ou, talvez, é simplesmente muito difícil corrigi-lo e existem assuntos mais importantes na agenda. Se for necessária ajuda imediata, deve ser levada em consideração a contratação de um suporte comercial.

Identificando erros

Antes de relatar um erro, por favor leia e releia a documentação para verificar se realmente pode ser feito o que está se tentando fazer. Se não está claro na documentação que algo pode ou não ser feito, por favor informe isto também; é um erro da documentação. Se acontecer do programa fazer algo diferente do que está escrito na documentação, isto também é um erro. Pode incluir, mas não está restrito às seguintes circunstâncias:

- O programa termina com um erro fatal, ou com uma mensagem de erro do sistema operacional

2 /var/lib/pgsql/ e /var/lib/pgsql/data no RedHat e no Mandrake. (N.T.)

que aponta para um erro no programa (um exemplo oposto seria uma mensagem de "disco cheio", porque o próprio usuário deve corrigir este problema).
- O programa produz uma saída errada para uma entrada específica.
- O programa se recusa a aceitar uma entrada válida (conforme definido na documentação).
- O programa aceita uma entrada inválida sem enviar uma mensagem de erro. Porém, tenha em mente que a sua idéia de entrada errada pode ser a nossa idéia de uma extensão, ou a compatibilidade com a prática tradicional.
- O PostgreSQL falha ao compilar, montar ou instalar de acordo com as instruções, em uma plataforma suportada.

Aqui "programa" se refere a qualquer executável, e não apenas ao processo servidor.

Estar lento ou consumir muitos recursos não é necessariamente um erro. Leia a documentação ou faça perguntas em alguma lista de discussão pedindo ajuda para ajustar suas aplicações. Não agir de acordo com o padrão SQL também não é necessariamente um erro, a não ser que a aderência com a funcionalidade específica esteja explicitamente declarada.

Antes de prosseguir, verifique a lista TODO (a fazer) e a FAQ para ver se o erro já não é conhecido. Se você não conseguir decodificar a informação da lista TODO, relate seu problema. O mínimo que podemos fazer é tornar a lista TODO mais clara.

O que relatar

O mais importante a ser lembrado quando relatamos erros é declarar todos os fatos, e somente os fatos. Não especule sobre o que você acha que está certo ou errado, o que "parece que deva ser feito", ou qual parte do programa está falhando. Se você não está familiarizado com a implementação você provavelmente vai supor errado, e não vai nos ajudar nem um pouco. E, mesmo que você esteja familiarizado, uma explanação educada é um grande suplemento, mas não substitui os fatos. Se nós formos corrigir o erro, nós temos que vê-lo acontecer primeiro. Informar fatos cruamente é relativamente direto (provavelmente pode ser copiado e colado a partir da tela), mas geralmente detalhes importantes são deixados de fora porque alguém pensou que não tinha importância, ou que o relatório seria entendido de qualquer forma.

Os seguintes itens devem estar contidos em todo relatório de erro:

A seqüência exata dos passos, desde o início do programa, necessários para reproduzir o problema. Isto deve estar autocontido; não é suficiente enviar apenas um comando SELECT sem enviar os comandos CREATE TABLE e INSERT que os precederam, se a saída depender dos dados contidos nas tabelas. Nós não temos tempo para realizar a engenharia reversa do esquema do seu banco de dados e, se nós tivermos que criar os nossos próprios dados, nós provavelmente não vamos conseguir reproduzir o problema. O melhor formato para realizar um caso de teste, para problemas associados à linguagem de consulta, é um arquivo que possa ser executado a partir da aplicação psql e que mostre o problema (certifique-se que não existe nada em seu arquivo de inicialização ~/.psqlrc). Um modo fácil de começar este arquivo é usar o pg_dump para gerar as declarações da tabela e dos dados necessários para montar o cenário, e depois incluir a consulta problemática. Encorajamos você a minimizar o tamanho do exemplo, mas isto não é absolutamente necessário. Se o erro for reproduzível, nós o encontraremos de qualquer forma.

Se o sua aplicação utiliza alguma outra interface no cliente tal como o PHP então, por favor, tente isolar a consulta problemática. Provavelmente nós não vamos configurar um servidor Web para reproduzir seu problema. De qualquer forma, lembre-se de fornecer os arquivos de entrada exatos, e não suponha que o problema aconteça com "arquivos grandes" ou "bancos de dados de tamanho médio", etc. porque estas informações são muito pouco precisas para serem utilizadas.

A saída produzida. Por favor, não diga que "não funcionou" ou que "deu pau". Havendo uma mensagem de erro mostre-a, mesmo que você não consiga entendê-la. Se o programa terminar com um erro do sistema operacional, diga qual. Se nada acontecer, informe. Mesmo que o resultado do seu caso de teste seja o término anormal do programa, ou seja óbvio de alguma outra forma, pode ser que isto não aconteça em nossa plataforma. O mais fácil a ser feito é copiar a saída do terminal, quando possível.

Nota: No caso de erros fatais a mensagem de erro informada pelo cliente pode não conter toda a informação disponível. Por favor, olhe também o log produzido no servidor de banco de dados. Se você não mantém a saída do log do servidor, esta é uma boa hora para começar a fazê-lo.

A saída esperada é uma informação importante a ser declarada. Se você escreve apenas "Este comando produz esta saída." ou "Isto não é o que eu esperava.", nós podemos executar, olhar a saída, e achar que está tudo correto e é exatamente o que nós esperávamos que fosse. Nós não temos que perder tempo para decodificar a semântica exata por trás de seus comandos. Abstenha-se especialmente de dizer meramente "Isto não é o que o SQL diz ou o que o Oracle faz". Pesquisar o comportamento correto do SQL não é uma tarefa divertida, nem nós sabemos como todos os outros bancos de dados relacionais existentes se comportam (se seu problema é o término anormal do seu programa, este item obviamente pode ser omitido).

Qualquer opção de linha de comando e outras opções de inicialização, incluindo as variáveis de ambiente relacionadas ou arquivos de configuração que foram alterados em relação ao padrão. Novamente, seja exato. Se estiver sendo utilizada uma distribuição pré-configurada que inicializa o servidor de banco de dados durante o boot, deve-se tentar descobrir como isto é feito.

Qualquer coisa feita diferente das instruções de instalação.

A versão do PostgreSQL. Pode ser executado o comando SELECT version(); para descobrir a versão do servidor ao qual se está conectado. A maioria dos programas executáveis suporta a opção --version; ao menos postmaster --version e psql --version devem funcionar. Se a função ou as opções não existirem, então a versão sendo usada é muito antiga e merece ser atualizada. Também pode ser visto o arquivo README no diretório do fonte, ou o arquivo com o nome da distribuição ou o nome do pacote. Se a versão for pré-configurada, como RPMs, informe, incluindo qualquer sub-versão que o pacote possa ter. Se estiver se referindo a uma versão do CVS isto deve ser mencionado, incluindo a data e a hora.

Se sua versão for anterior a 7.3.4 provavelmente nós lhe pediremos que atualize. Existem toneladas de correções de erro a cada nova liberação, sendo este o motivo das novas liberações.

Informações da plataforma. Isto inclui o nome do núcleo e a versão, a biblioteca C, o processador e a memória. Na maioria dos casos é suficiente informar o fornecedor e a versão, mas não se deve supor que todo mundo sabe exatamente o que o "Debian" contém, ou que todo mundo use Pentium.

Havendo problemas de instalação, então as informações sobre compilador, make, etc. também são necessárias.

Não tenha medo que seu relatório de erro se torne muito longo. Este é um fato da vida. É melhor relatar tudo da primeira vez do que nós termos que extrair os fatos de você. Por outro lado, se seus arquivos de entrada são enormes, é justo perguntar primeiro se alguém está interessado em vê-los.

Não perca seu tempo tentando descobrir que mudanças na entrada fazem o problema desaparecer. Isto provavelmente não vai ajudar a resolver o problema. Se for visto que o erro não pode ser corrigido imediatamente, você ainda vai ter tempo para compartilhar sua descoberta com os outros. Também, novamente, não perca seu tempo adivinhando porque o erro existe. Nós o encontraremos brevemente.

Ao escrever o relatório de erro, por favor escolha uma terminologia que não confunda. O pacote de software em seu todo é chamado de "PostgreSQL" e, algumas vezes, de "Postgres" para abreviar. Se estiver se referindo especificamente ao processo servidor mencione isto, não diga apenas que o "PostgreSQL caiu". A queda de um único processo servidor é bem diferente da queda do processo "postmaster" pai; por favor não diga "o postmaster caiu" quando um único processo servidor caiu, nem o contrário. Além disso os programas cliente, como o terminal interativo "psql", são completamente separados do servidor. Por favor, tente especificar se o problema está no lado do cliente ou no lado do servidor.

Onde relatar os erros

De modo geral, os relatórios de erro devem ser enviados para a lista de discussão de relatórios de erros em <pgsql-bugs@postgresql.org>. É necessária a utilização de um assunto descritivo para a mensagem de correio eletrônico, talvez uma parte da própria mensagem de erro.

Outro método é preencher o relatório de erro disponível no sítio do projeto na Web em http://www.postgresql.org/. O preenchimento desta forma faz com que seja enviado para a lista de discussão <pgsql-bugs@postgresql.org>.

Não envie o relatório de erro para nenhuma lista de discussão dos usuários, tal como <pgsql-sql@postgresql.org> ou <pgsql-general@postgresql.org>. Estas listas de discussão são para responder perguntas dos usuários, e seus subscritores normalmente não desejam receber relatórios de erro. Mais importante ainda, eles provavelmente não vão conseguir corrigir o erro.

Por favor, também não envie relatórios para a lista de discussão dos desenvolvedores em <pgsql-hackers@postgresql.org>. Esta lista é para discutir o desenvolvimento do PostgreSQL e nós gostamos de manter os relatórios de erro em separado. Nós podemos decidir discutir seu relatório de erro em pgsql-hackers, se o problema necessitar uma maior averiguação.

Se você tiver problema com a documentação, o melhor lugar para relatar é na lista de discussão da documentação em <pgsql-docs@postgresql.org>. [3] Por favor, seja específico sobre qual parte da documentação você está descontente.

Se seu erro for algum problema de portabilidade ou uma plataforma não suportada, envie uma

3 Informe os erros encontrados na tradução deste manual a <halleypo@yahoo.com.br>. (N.T.)

mensagem de correio eletrônico para <pgsql-ports@postgresql.org>, para que nós (e você) possamos trabalhar para portar o PostgreSQL para esta plataforma.

Nota: Devido à grande quantidade de spam na Internet, todos os endereços de correio eletrônico acima são de listas de discussão fechadas. Ou seja, você precisa subscrever a lista primeiro para depois poder enviar mensagens (entretanto, você não precisa subscrever para utilizar o formulário de relatório de erro da Web). Se você deseja enviar uma mensagem de correio eletrônico, mas não deseja receber o tráfego da lista, você pode subscrever e configurar sua opção de subscrição com nomail. Para maiores informações envie uma mensagem para <majordomo@postgresql.org> contendo apenas a palavra help no corpo da mensagem.

Requerimentos e Especificações

Para MS Windows NT e MS Windows 9x:

Sistema Operacional	**WinNT, XP**	**Win9x ou Linux**
Memória	64 Mb no mínimo	16 Mb no mínimo
Processador	Pentium II	i486 DX2 100Mhz no mínimo

CAPÍTULO 3. Instalação

Para a instalação proceda conforme descrito abaixo. Caso não tenha os pacotes, realize download diretamente do sitio oficial www.postgresql.org ou de sua distribuição específica

Em ambiente GNU\Linux

Fedora	:	#rpm -ivh postgresql<versao>
Debian, Kurumin	:	#apt-get install postgresql<versao>

- Os arquivos de configurações são instalados no diretório /etc/postgresql
- Os arquivos de dados são instalados por padrão em:
  /var/lib/postgres

- O arquivo de inicialização está programado em shell-script e está instalado em /etc/rc.d/init.d/postgresql.
  Para iniciar o serviço (modo root) #/etc/rc.d/init.d/postgresql start
  Para parar o serviço (modo root) #/etc/rc.d/init.d/postgresql stop

Em ambiente Windows

Utilize o pacote postgresql<versao>.msi
Execute o pacote e a instalação será iniciada de uma maneira bastante fácil.

Observações:

Ambiente GNU\Linux e xBSD.

1. Caso ocorra erro em sua instalação verifique se não falta bibliotecas dependentes. Cada distribuição tem sua maneira própria de fixá-las.
2. O serviço "pai" é chamado de "postmaster". Para verificar se o serviço está sendo executado utilize o comando de exibição de processos:
   #ps aux

3. Por padrão o PostgreSQL utiliza a porta 5432. Se o pacote nmap estiver instalado execute
   #nmap <ip>
   e você poderá verificar se a porta foi liberada.

CAPÍTULO 4. Introdução ao SQL

Este capítulo descreve a sintaxe da linguagem SQL [4], estabelecendo uma base para compreender os próximos capítulos que descrevem detalhadamente como os comandos SQL são utilizados para definir e modificar os dados. [5]

Aconselha-se aos usuários já familiarizados com a linguagem SQL a leitura cuidadosa deste capítulo, porque existem várias regras e conceitos implementados pelos bancos de dados SQL de forma inconsistente, ou que são específicos do PostgreSQL.

1. Estrutura léxica

Uma declaração SQL é formada por uma seqüência de comandos. Um comando é formado por uma seqüência de termos (tokens) terminada por um ponto-e-vírgula (";"). O fim da declaração também termina o comando. Quais termos são válidos depende da sintaxe de cada comando.

Um termo pode ser uma palavra chave, um identificador, um identificador entre aspas, um literal (ou constante), ou um caractere especial. Geralmente os termos são separados por espaço em branco (espaço, tabulação ou nova-linha), mas não precisam estar se não houver ambigüidade (normalmente isto só acontece quando um caractere especial está adjacente a um termo de outro tipo).

Além disso, comentários podem estar presentes na declaração SQL. Os comentários não são termos, na realidade são equivalentes a espaços em branco.

Abaixo está mostrada uma declaração SQL válida (sintaticamente):

```
SELECT * FROM MINHA_TABELA;
UPDATE MINHA_TABELA SET A = 5;
INSERT INTO MINHA_TABELA VALUES (3, 'oi cara');
```

Esta é uma seqüência de três comandos, um por linha (embora isto não seja requerido; pode haver mais de um comando na mesma linha, e um único comando pode ocupar várias linhas).

A sintaxe do SQL não é muito consistente em relação a quais termos identificam comandos e quais são operandos ou parâmetros. Geralmente os primeiros termos são o nome do comando. Portanto, no exemplo mostrado acima, pode-se dizer que estão presentes os comandos "SELECT", "UPDATE" e "INSERT". Entretanto, o comando UPDATE sempre requer que o termo SET apareça em uma determinada posição, e esta forma particular do INSERT também requer a presença do

4 SQL (Structured Query Language)

5 Sintaxe é parte da gramática que estuda as palavras enquanto elementos de uma frase, as suas relações de concordância, de subordinação e de ordem - Dicionário Eletrônico Houaiss. (N.T.)

termo VALUES para estar completa. As regras precisas da sintaxe de cada comando estão descritas no Manual de Referência do PostgreSQL.

Identificadores e palavras chave

Os termos SELECT, UPDATE e VALUES mostrados no exemplo acima são exemplos de palavras chave, ou seja, palavras que possuem um significado estabelecido na linguagem SQL. Os termos MINHA_TABELA e A são exemplos de identificadores, os quais identificam nomes de tabelas, colunas e outros objetos do banco de dados, dependendo do comando onde são utilizados. Portanto, algumas vezes são simplesmente chamados de "nomes". As palavras chave e os identificadores possuem a mesma estrutura léxica, significando que não é possível saber se o termo é um identificador ou uma palavra chave sem conhecer a linguagem.

Os identificadores e as palavras chave do SQL devem iniciar por uma letra (a-z e, também, letras com diacrítico - áéç... - e letras não latinas), ou o caractere sublinhado (_). Os demais caracteres de um identificador, ou da palavra chave, podem ser letras, dígitos (0-9) ou sublinhados, embora o padrão SQL não defina nenhuma palavra chave contendo dígitos, ou que comece ou termine por um caractere sublinhado.

O sistema utiliza não mais que NAMEDATALEN-1 caracteres de um identificador; nomes maiores podem ser escritos nos comandos, mas serão truncados. Por padrão NAMEDATALEN é 64 e, portanto, o comprimento máximo de um identificador é 63 (mas antes de gerar o PostgreSQL NAMEDATALEN pode ser mudado no arquivo src/include/postgres_ext.h).

Os identificadores e as palavras chave não fazem distinção entre letras maiúsculas e minúsculas. Portanto,

```
UPDATE MINHA_TABELA SET A = 5;
```

é equivalente a

```
uPDaTE MINHA_TABELA SeT a = 5;
```

Normalmente utiliza-se a convenção de escrever as palavras chave em letras maiúsculas e os nomes em letras minúsculas, como mostrado abaixo:

```
UPDATE minha_tabela SET a = 5;
```

Existe um segundo tipo de identificador: o identificador delimitado ou identificador entre aspas, formado pela colocação de uma seqüência arbitrária de caracteres entre aspas ("). Um identificador delimitado é sempre um identificador, e nunca uma palavra chave. Portanto, "select" pode ser usado para fazer referência a uma tabela ou coluna chamada "select", enquanto um select sem aspas sempre é uma palavra chave ocasionando, por isso, um erro do analisador quando usado onde um nome de tabela ou de coluna é esperado. O exemplo acima pode ser reescrito utilizando identificadores entre aspas:

```
UPDATE "minha_tabela" SET "a" = 5;
```

Um identificador entre aspas pode conter qualquer caractere que não seja a própria aspa. Para incluir uma aspa, duas aspas devem ser escritas. Esta funcionalidade permite a criação de nomes de tabelas e de colunas que não seriam possíveis de outra forma, como os contendo espaços ou e-comercial (&). O limite de comprimento ainda se aplica.

Colocar um identificador entre aspas torna diferente as letras maiúsculas das minúsculas, enquanto os nomes não envoltos por aspas são sempre convertidos para letras minúsculas. Por exemplo, os identificadores FOO, foo e "foo" são considerados o mesmo pelo PostgreSQL, mas "Foo" e "FOO" são diferentes dos três primeiros e entre si. [6]

Constantes

Existem três tipos de constantes com tipo implícito no PostgreSQL: cadeias de caracteres, cadeias de bits e numéricas. As constantes também podem ser especificadas com tipo explícito, o que permite uma representação mais precisa, e um tratamento mais eficiente por parte do sistema. As constantes implícitas estão descritas abaixo; as constantes explícitas são discutidas mais adiante.

Constantes do tipo cadeia de caracteres

Uma constante do tipo cadeia de caracteres no SQL é uma seqüência arbitrária de caracteres envolta por apóstrofos (') como, por exemplo, 'Esta é uma cadeia de caracteres'. O SQL permite apóstrofos dentro de uma cadeia de caracteres digitando-se dois apóstrofos adjacentes (por exemplo, 'Maria D''Almeida'). No PostgreSQL existe a alternativa de utilizar a contrabarra ("\") como caractere de escape para colocar apóstrofos dentro de cadeia de caracteres (por exemplo, 'Maria D\'Almeida').

As seqüências de escape presentes na linguagem C também são permitidas: \b para voltar apagando (backspace), \f para avanço de página, \n para nova-linha, \r para retorno do carro, \t para tabulação e \xxx, onde xxx é o número octal correspondente ao código ASCII do caractere. Qualquer outro caractere vindo após a contrabarra é interpretado literalmente. Portanto, para incluir uma contrabarra em uma constante do tipo cadeia de caracteres, devem ser escritas duas contrabarras adjacentes.

O caractere com o código zero não pode estar presente em uma constante do tipo cadeia de caracteres.

Duas constantes do tipo cadeia de caracteres separadas apenas por espaço em branco e pelo menos um caractere de nova-linha, são concatenadas e tratadas como se a cadeia de caracteres fosse uma única constante. Por exemplo:

```
SELECT 'foo'
'bar';
```

é o mesmo que

```
SELECT 'foobar';
```

6 A conversão em minúsculas das letras dos nomes que não estão entre aspas feita pelo PostgreSQL não é compatível com o padrão SQL, que estabelece a conversão em maiúsculas das letras dos nomes que não estão entre aspas. Portanto, foo deve ser equivalente a "FOO" e não a "foo" de acordo com o padrão. Se for desejado desenvolver aplicações portáveis, o nome deve ser colocado sempre entre aspas, ou nunca ser colocado entre aspas.

mas

SELECT 'foo' 'bar';

não possui uma sintaxe válida (este comportamento, um tanto ao quanto esquisito, é especificado no padrão SQL; o PostgreSQL apenas segue o padrão).

Constantes do tipo cadeia de bits

Uma constante do tipo cadeia de bits se parece com uma constante do tipo cadeia de caracteres contendo a letra B (maiúscula ou minúscula) logo antes do apóstrofo inicial (sem espaços separando) como, por exemplo, B'1001'. Os únicos caracteres permitidos dentro de uma constante do tipo cadeia de bits são o 0 e o 1.

Como forma alternativa, uma constante do tipo cadeia de bits pode ser especificada usando a notação hexadecimal, colocando a letra X (maiúscula ou minúscula) no início como, por exemplo, X'1FF'. Esta notação é equivalente a uma constante do tipo cadeia de bits contendo quatro dígitos binários para cada dígito hexadecimal.

As duas formas de constantes do tipo cadeia de bits podem ocupar mais de uma linha, da mesma forma que uma constante do tipo cadeia de caracteres.

Constantes numéricas

As constantes numéricas são admitidas nas seguintes formas gerais:

dígitos
dígitos.[dígitos][e[+-]dígitos]
[dígitos].dígitos[e[+-]dígitos]
dígitose[+-]dígitos

onde dígitos é um ou mais dígitos decimais (0 a 9). Pelo menos um dígito deve existir antes ou depois do ponto decimal, se este for usado. Pelo menos um dígito deve existir após a marca de expoente (e), caso esteja presente. Não podem existir espaços ou outros caracteres embutidos em uma constante numérica. Observe que os sinais menos e mais que antecedem a constante não são na verdade considerados parte da constante, e sim um operador a ser aplicado à constante.

Abaixo estão mostrados alguns exemplo de constantes numéricas válidas:

42
3.5
4.
.001
5e2
1.925e-3

Uma constante numérica não contendo o ponto decimal nem o expoente é presumida, inicialmente,

como sendo do tipo inteiro se seu valor for apropriado para o tipo integer (32 bits); senão é presumida como sendo do tipo inteiro longo, se seu valor for apropriado para o tipo bigint (64 bits); caso contrário, é assumida como sendo do tipo numeric. As constantes que possuem pontos decimais e/ou expoentes sempre são inicialmente presumidas como sendo do tipo numeric.

O tipo de dado atribuído inicialmente para a constante numérica é apenas o ponto de partida para o algoritmo de resolução de tipo. Na maioria dos casos a constante é automaticamente tornada do tipo mais apropriado dependendo do contexto. Quando for necessário, pode ser feito o valor numérico ser interpretado como sendo de um tipo de dado específico, definindo a transformação a ser feita. Por exemplo, pode ser feito o valor numérico ser tratado como sendo do tipo real (float4) escrevendo

```
REAL '1.23'  -- estilo cadeia de caracteres
1.23::REAL   -- estilo PostgreSQL (histórico)
```

Constantes de outros tipos

Uma constante de um tipo arbitrário pode ser declarada utilizando uma das seguintes notações:

```
tipo 'cadeia de caracteres'
'cadeia de caracteres'::tipo
CAST ( 'cadeia de caracteres' AS tipo )
```

O texto da cadeia de caracteres é passado para a rotina de conversão de entrada para o tipo chamado tipo. O resultado é uma constante do tipo indicado. A conversão explícita de tipo pode ser omitida, caso não haja ambigüidade com relação ao tipo que a constante deva ter (por exemplo, quando passada como argumento para uma função não-sobrecarregada), caso onde é automaticamente transformada.

Também é possível especificar a transformação de tipo utilizando a sintaxe semelhante à chamada de função:

```
nome_do_tipo ( 'cadeia de caracteres' )
```

mas nem todos os nomes de tipo podem ser usados desta forma.

As sintaxes ::, CAST(), e chamada de função também podem ser utilizadas para especificar conversão de tipo em tempo de execução para expressões arbitrárias. Porém, a forma tipo 'cadeia de caracteres' somente pode ser utilizada para especificar tipo em constante literal. Outra restrição com relação a sintaxe tipo 'cadeia de caracteres' é que não pode ser usada em matrizes (arrays); deve ser usado :: ou CAST() para especificar tipo em matriz.

Constantes em forma de matriz

O formato geral de uma constante em forma de matriz é o seguinte:

```
'{ val1 delim val2 delim ... }'
```

onde delim é o caractere delimitador para o tipo, conforme registrado em sua entrada na tabela pg_type (para todos os tipos nativos é utilizado o caractere vírgula ","). Cada val pode ser tanto uma constante do tipo do elemento da matriz quanto uma submatriz. Um exemplo de uma constante em forma de matriz é

'{{1,2,3},{4,5,6},{7,8,9}}'

Esta constante é uma matriz bidimensional de 3x3, formada por 3 submatrizes de números inteiros.

Os elementos individuais da matriz podem ser colocados entre aspas ("), para evitar problemas de ambigüidade com respeito a espaços em branco. Sem aspas, o analisador de valores da matriz salta os espaços em branco precedentes.
As constantes em forma de matriz são na verdade apenas um caso especial do tipo genérico. A constante é inicialmente tratada como cadeia de caracteres e passada para a rotina de conversão de declaração de matriz. Uma especificação explícita do tipo pode ser necessária.

Operadores

Um operador é uma seqüência com até NAMEDATALEN-1 (por padrão 63) caracteres da seguinte lista:

+ - * / < > = ~ ! @ # % ^ & | ` ? $

Entretanto existem algumas poucas restrições relativas aos nomes dos operadores:

O $ (cifrão) não pode ser um operador de um único caractere, embora possa fazer parte do nome de um operador com vários caracteres.

As seqüências -- e /* não podem aparecer em nenhuma posição no nome do operador, porque são consideradas como início de comentário.

Um nome de operador com vários caracteres não pode terminar em + ou -, a não ser que o nome contenha também pelo menos um dos seguintes caracteres:

~ ! @ # % ^ & | ` ? $

Por exemplo, @- é um nome de operador permitido, mas *- não é. Esta restrição permite ao PostgreSQL analisar consultas que estão de acordo com o padrão SQL sem haver necessidade de espaço entre os termos.

Ao trabalhar com nomes de operadores fora do padrão SQL, normalmente é necessário separar operadores adjacentes com espaço para evitar ambigüidade. Por exemplo, se for definido um operador unário esquerdo chamado @, não poderá ser escrito X*@Y; deverá ser escrito X* @Y para garantir que o PostgreSQL leia dois nomes de operadores e não apenas um.

Caracteres especiais

Alguns caracteres não alfanuméricos possuem um significado especial diferente de ser um operador.

Os detalhes da utilização podem ser encontrados nos locais onde a sintaxe do respectivo elemento é descrita. Esta seção se destina apenas a informar a existência e fazer um resumo das finalidades destes caracteres.

O caractere cifrão ($) seguido por dígitos é utilizado para representar os parâmetros posicionais no corpo da definição de uma função. Em outros contextos o caractere cifrão pode fazer parte do nome de um operador.

Os parênteses (()) possuem seu significado usual de agrupar expressões e impor a precedência. Em alguns casos os parênteses são necessários como parte fixa da sintaxe de um determinado comando SQL.

Os colchetes ([]) são utilizados para selecionar elementos da matriz.

As vírgulas (,) são utilizadas em algumas construções sintáticas para separar os elementos da lista.

O ponto-e-vírgula (;) termina um comando SQL, não podendo aparecer em nenhum lugar dentro do comando, exceto dentro de constante do tipo cadeia de caracteres ou identificadores entre aspas.

Os dois-pontos (:) são utilizados para selecionar "faixas" em matrizes. Em certos dialetos do SQL (como o SQL embutido), os dois-pontos são utilizados como prefixo dos nomes das variáveis.

O asterisco (*) possui um significado especial quando utilizado no comando SELECT ou na função de agregação COUNT.

O ponto (.) é utilizado nas constantes de ponto flutuante, e para separar os nomes de esquemas, tabelas e colunas.

Comentários

Um comentário é uma seqüência arbitrária de caracteres começando por dois hífens e prosseguindo até o fim da linha como, por exemplo:

```
-- Este é um comentário no padrão SQL92
```

Como alternativa, podem ser utilizados blocos de comentários no estilo C:

```
/* comentário de várias linhas
 * com: /* bloco de comentário aninhado */
 */
```

onde o comentário começa por /* se estendendo até encontrar a ocorrência de */. Estes blocos de comentários podem estar aninhados, conforme especificado no SQL99 (mas diferentemente da linguagem C), permitindo transformar em comentário grandes blocos de código que possuam blocos de comentários existentes.

Os comentários são removidos da declaração antes de prosseguir com a análise sintática sendo substituídos por espaço em branco.

Precedência léxica

A maioria dos operadores possuem a mesma precedência e possuem associatividade esquerda. A precedência e a associatividade dos operadores está codificada no analisador, podendo ocasionar um comportamento contra-intuitivo; por exemplo, os operadores booleanos < e > possuem uma precedência diferente dos operadores booleanos <= e >=. Também, em alguns casos é necessário adicionar parênteses quando é utilizada a combinação de operadores unários e binários. Por exemplo,

SELECT 5 ! - 6; será analisado como SELECT 5 ! (- 6);

porque o analisador não possui a menor idéia -- até ser tarde demais -- que o ! é definido como operador unário direito (postfix), e não um operador binário colocado entre os operandos (infix). Neste caso, para obter o comportamento desejado deve ser escrito

SELECT (5 !) - 6;

Este é o preço a ser pago pela extensibilidade.

Tabela - Precedência dos operadores (decrescente)

Operador/Elemento	Associatividade	Descrição
.	esquerda	separador de nome de tabela/coluna
::	esquerda	transformação 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	todos os operadores nativos e definidos pelo usuário
IN		membro de um conjunto
BETWEEN		intervalo fechado
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

Observe que as regras de precedência dos operadores também se aplicam aos operadores definidos pelos usuários que possuem os mesmos nomes dos operadores nativos mencionados acima. Por exemplo, se for definido um operador "+" para algum tipo de dados personalizado, este terá a mesma precedência do operador "+" nativo, não importando o que faça.

Quando é utilizado um nome de operador qualificado pelo esquema na sintaxe do OPERATOR como, por exemplo,

SELECT 3 OPERATOR(pg_catalog.+) 4;

a construção OPERATOR é assumida como tendo a precedência padrão para "qualquer outro" operador. Isto é sempre verdade não importando qual é o nome específico do operador dentro de OPERATOR().

2. Expressões de valor

As expressões de valor são utilizadas em diversos contextos, como na lista de seleção do comando SELECT, nos valores das colunas nos comandos INSERT e UPDATE, e na condição de pesquisa em vários comandos. O resultado de uma expressão de valor é algumas vezes chamado de escalar, para distinguir do resultado de uma expressão de tabela (que é uma tabela). As expressões de valor são, portanto, chamadas também de expressões escalares (ou pura e simplesmente de expressões). A sintaxe da expressão permite o cálculo de valores a partir de partes primitivas utilizando operações aritméticas, lógicas, de conjunto e outras.

A expressão de valor é uma das seguintes:

Um valor constante ou literal;
Uma referência a uma coluna.
Uma referência a um parâmetro posicional, na declaração do corpo da função.
Uma chamada de operador.
Uma chamada de função.
Uma expressão de agregação.
Uma transformação de tipo.
Uma subconsulta escalar.

Outra expressão de valor entre parênteses, útil para agrupar subexpressões e mudar precedências.

Devem ser acrescentadas a esta lista diversas construções que podem ser classificadas como expressão, mas que não seguem as regras gerais de sintaxe. Possuem, normalmente, a semântica de uma função ou de um operador. Um exemplo é a cláusula IS NULL.

Referências a colunas

Uma coluna pode ser referenciada usando a forma

correlação.nome_da_coluna

ou a forma

correlação.nome_da_coluna[índice]

(Neste caso, os colchetes [] devem ser escritos literalmente)

onde correlação é o nome (possivelmente qualificado) de uma tabela, ou um aliás para a tabela definido na cláusula FROM, ou ainda as palavras chave NEW ou OLD (NEW e OLD somente podem aparecer nas regras de reescrita, enquanto os outros nomes de correlação podem ser usados em qualquer declaração SQL). O nome de correlação e o ponto separador podem ser omitidos se o nome da coluna for único entre todas as tabelas utilizadas pela consulta corrente.

Se a coluna for do tipo matriz (array), então o índice opcional seleciona um elemento específico ou elementos da matriz. Se nenhum índice for fornecido, então toda a matriz é selecionada.

Parâmetros posicionais

Uma referência a um parâmetro posicional é utilizada para indicar o parâmetro fornecido externamente a uma declaração SQL. Os parâmetros são utilizados nas definições das funções SQL e nas consultas preparadas. A forma de fazer referência a um parâmetro é:

$número

Por exemplo, considere a definição da função dept como sendo

```
CREATE FUNCTION dept(text) RETURNS dept
   AS 'SELECT * FROM dept WHERE name = $1'
   LANGUAGE SQL;
```

Neste caso $1 será substituído pelo primeiro argumento da função quando a função for chamada.

Chamadas de operador

Existem três sintaxes possíveis para a chamada de operador:

expressão operador expressão (operador binário intermediário)
operador expressão (operador unário esquerdo)
expressão operador (operador unário direito)

onde o termo operador segue as regras de sintaxe já descritas ou é uma das palavras chave AND, OR ou NOT, ou é um nome de operador qualificado
OPERATOR(esquema.nome_do_operador)

Quais são os operadores existentes, e se são unários ou binários, depende de quais operadores foram definidos pelo sistema e pelo usuário.

Chamadas de funções

A sintaxe de uma chamada de função é o nome da função (possivelmente qualificado pelo nome do esquema), seguida por sua lista de argumentos entre parênteses:

função ([expressão [, expressão ...]])

Por exemplo, a função abaixo calcula a raiz quadrada de 2:
sqrt(2)

Outras funções podem ser adicionadas pelo usuário.

Funções de agregação

Uma expressão de agregação representa a aplicação de uma função de agregação nas linhas selecionadas por uma consulta. Uma função de agregação transforma vários valores de entrada em um único valor de saída, tal como a soma ou a média. A sintaxe da expressão de agregação é uma das seguintes:

nome_da_agregação (expressão)
nome_da_agregação (ALL expressão)
nome_da_agregação (DISTINCT expressão)
nome_da_agregação (*)

onde o nome_da_agregação é uma agregação definida previamente (possivelmente um nome qualificado), e expressão é qualquer expressão de valor que não contenha uma expressão de agregação.
A primeira forma de expressão de agregação chama a função de agregação para todas as linhas de entrada onde a expressão fornecida produz um valor não nulo (na verdade, é decisão da função de agregação ignorar ou não os valores nulos --- porém todas as funções padrão o fazem). A segunda forma é idêntica à primeira, porque ALL é o padrão. A terceira forma chama a função de agregação para todos os valores distintos e não nulos da expressão encontrados nas linhas de entrada. A última forma chama a função de agregação uma vez para cada linha de entrada independentemente do valor ser nulo ou não; uma vez que nenhum valor específico de entrada é especificado, geralmente é útil apenas para a função de agregação count().

Por exemplo, count(*) retorna o número total de linhas de entrada; count(campo) retorna o número de linhas de entrada onde campo não é nulo; count(distinct campo) retorna o número de valores distintos e não nulos de campo.

Outras funções de agregação podem ser adicionadas pelo usuário.

Transformação de tipo

Uma transformação de tipo (type cast) especifica a conversão de um tipo de dado em outro. O PostgreSQL aceita duas sintaxes equivalentes para transformação de tipo:

CAST (expressão AS tipo)
expressão::tipo

A sintaxe CAST está em conformidade com o padrão SQL; a sintaxe com :: é uma utilização histórica do PostgreSQL.
Quando a transformação é aplicada a uma expressão de valor de um tipo conhecido, esta representa uma conversão em tempo de execução. A transformação será bem sucedida apenas se uma função de conversão adequada estiver disponível. Observe que isto é sutilmente diferente da utilização de transformação com constante. Uma transformação aplicada a uma cadeia de caracteres sem adornos representa a atribuição inicial do tipo para o valor constante literal e, portanto, será bem sucedida para qualquer tipo (se o conteúdo da cadeia de caracteres possuir uma sintaxe válida para servir de entrada para o tipo de dado).
Geralmente uma transformação explícita de tipo pode ser omitida quando não há ambigüidade em relação ao tipo que a expressão de valor deve produzir (por exemplo, quando é atribuído para coluna de tabela); o sistema aplica automaticamente a transformação de tipo nestes casos. Entretanto, a transformação automática de tipo é feita apenas para as transformações marcadas como "OK para aplicar implicitamente" nos catálogos do sistema. As outras transformações devem ser chamadas por meio da sintaxe de transformação explícita. Esta restrição tem por finalidade prevenir que aconteçam conversões surpreendentes aplicadas silenciosamente.
Também é possível especificar uma transformação utilizando a sintaxe na forma de função:

nome_do_tipo (expressão)

Entretanto, somente funciona para os tipos cujos nomes também são válidos como nome de função. Por exemplo, double precision não pode ser utilizado desta maneira, mas a forma equivalente float8 pode. Também, os nomes interval, time e timestamp somente podem ser utilizados desta maneira se estiverem entre aspas, devido a conflitos sintáticos. Portanto, o uso da sintaxe na forma de função para transformações pode ocasionar inconsistências devendo ser evitada em novas aplicações. (A sintaxe na forma chamada de função é na verdade apenas a chamada de uma função. Quando uma das duas sintaxes padrão de conversão é utilizada para fazer uma conversão em tempo de execução, internamente é chamada a função registrada para realizar esta conversão. Por convenção, estas funções de conversão possuem o mesmo nome do tipo de dado da saída, mas isto não é algo em que uma aplicação portável deva confiar).

Subconsultas escalares

Uma subconsulta escalar é um comando SELECT comum, entre parênteses, que retorna exatamente uma linha com uma coluna. A consulta SELECT é executada e o único valor retornado é utilizado na expressão. É errado utilizar uma consulta que retorne mais de uma linha ou mais de uma coluna como subconsulta escalar (porém, se durante uma determinada execução a subconsulta não retornar nenhuma linha, não acontece nenhum erro: o resultado escalar é assumido como nulo). A subconsulta pode fazer referência a variáveis da consulta principal, as quais atuam como sendo constantes durante qualquer avaliação da subconsulta.
Por exemplo, a consulta abaixo retorna a cidade com a maior população de cada estado:

```
SELECT nome, (SELECT max(populacao) FROM cidades WHERE cidades.estado = estado.nome)
   FROM estados;
```

Avaliação de expressão

A ordem de avaliação das subexpressões não é definida. Especificamente, as entradas de um operador ou função não são necessariamente avaliadas da esquerda para a direita ou em qualquer outra ordem determinada.
Além disso, se o resultado da expressão puder ser determinado avaliando somente algumas partes da expressão, então as outras subexpressões podem não ser avaliadas. Por exemplo, se for escrito

SELECT true OR alguma_funcao();

então alguma_funcao() não será chamada (provavelmente). O mesmo caso pode acontecer se for escrito

SELECT alguma_funcao() OR true;

Observe que isto não é o mesmo que os "curtos circuitos" esquerda para direita dos operadores booleanos encontrados em algumas linguagens de programação.
Como conseqüência, não é bom utilizar funções com efeitos colaterais como parte de expressões complexas. É particularmente perigoso confiar em efeitos colaterais ou na ordem de avaliação nas cláusulas WHERE e HAVING, porque estas cláusulas são extensamente reprocessadas como parte do desenvolvimento do plano de execução. As expressões booleanas (combinações de AND/OR/NOT) nestas cláusulas podem ser reorganizadas em qualquer forma permitida pelas leis da álgebra booleana.
Quando for essencial estabelecer a ordem de avaliação, uma construção CASE pode ser utilizada. Por exemplo, esta é uma forma não confiável para tentar evitar uma divisão por zero na cláusula WHERE:

SELECT ... WHERE x <> 0 AND y/x > 1.5;

Mas esta forma é segura:

SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;

A construção CASE utilizada desta forma impede as tentativas de otimização devendo, portanto, ser utilizada apenas quando necessário.

CAPÍTULO 5. Definição de Dados

Este capítulo mostra a criação das estruturas no banco de dados que armazenam os dados. Em bancos de dados relacionais os dados são armazenados em tabelas, portanto a maior parte deste capítulo é dedicada a explicar como as tabelas são criadas e modificadas, e as funcionalidades disponíveis para controlar os dados que podem ser armazenados nas tabelas. Em seguida é discutido como as tabelas podem ser organizadas em esquemas, e quais privilégios podem ser atribuídos às tabelas. Ao final são vistas superficialmente outras funcionalidades que afetam o armazenamento dos dados, como visões, funções e gatilhos. Informações detalhadas relativas a estes tópicos podem ser encontradas no Guia do Programador do PostgreSQL.

1. Fundamentos de tabela

Uma tabela em um banco de dados relacional é muito semelhante a uma tabela no papel: é composta por linhas e colunas. O número e a ordem das colunas são fixos, e cada coluna possui um nome. O número de linhas é variável, refletindo a quantidade de dados armazenados em um determinado instante. O SQL não dá nenhuma garantia relativa à ordem das linhas na tabela. Quando uma tabela é lida, as linhas aparecem em uma ordem aleatória, a não ser que uma ordenação seja explicitamente requisitada. Além disso, o SQL não atribui identificadores únicos para as linhas e, portanto, é possível existirem várias linhas totalmente idênticas na tabela. Isto é uma conseqüência do modelo matemático no qual a linguagem SQL está baseada, mas geralmente não é desejável. Mais adiante neste capítulo será mostrado como lidar com esta questão.

Cada coluna possui um tipo de dado. O tipo de dado restringe o conjunto de valores que podem ser atribuídos à coluna. Isto confere uma semântica [7] aos dados armazenados na coluna, permitindo que estes dados possam ser operados por computador. Por exemplo, uma coluna declarada como sendo de um tipo numérico não pode armazenar cadeias de caracteres com texto arbitrário, e os dados armazenados nesta coluna podem ser utilizados para efetuar cálculos matemáticos. Ao contrário, uma coluna declarada como sendo do tipo cadeia de caracteres aceita praticamente qualquer dado, mas não pode ser usada para efetuar cálculos matemáticos, embora possam ser efetuadas outras operações, como a concatenação de cadeias de caracteres.

O PostgreSQL possui um extenso conjunto de tipos de dado nativos, adequados para muitas aplicações. Os usuários também podem definir seus próprios tipos de dado. A maioria dos tipos de dado nativos possui um nome e uma semântica óbvia. Alguns dos tipos de dado freqüentemente utilizados são o integer para números inteiros, numeric para números possivelmente fracionários, text para cadeias de caracteres, date para datas, time para valores da hora do dia, e timestamp para valores contendo tanto a data quanto a hora.

Para criar uma tabela é utilizado o comando CREATE TABLE, próprio para esta tarefa. Neste comando são especificados ao menos o nome da nova tabela, os nomes das colunas, e os tipos de

7 Semântica: o significado das palavras, por oposição à sua forma - Dicionário Eletrônico Houaiss. (N.T.)

dado de cada coluna. Por exemplo:

```
CREATE TABLE minha_primeira_tabela (
    primeira_coluna text,
    segunda_coluna integer
);
```

Este comando cria a tabela chamada minha_primeira_tabela contendo duas colunas. A primeira coluna chama-se primeira_coluna, e possui o tipo de dado text; a segunda coluna chama-se segunda_coluna, e possui o tipo de dado integer. Normalmente os nomes dos tipos também são identificadores, mas existem algumas exceções. Observe que a lista de colunas é separada por vírgula, e que esta lista está entre parênteses.

Obviamente, o exemplo anterior é muito artificial. Normalmente os nomes das tabelas e das colunas estão associados às informações armazenadas. Sendo assim, vejamos um exemplo mais realista:

```
CREATE TABLE produtos (
    id_produto integer,
    nome       text,
    preco      numeric
);
```

(O tipo numeric pode armazenar a parte fracionária, usual em valores monetários)

Dica: Quando são criadas tabelas inter-relacionadas, aconselha-se escolher uma forma consistente para atribuir nomes às tabelas e colunas. Para exemplificar, existe a possibilidade de utilizar nomes de tabelas no singular ou no plural, e cada uma destas formas é defendida por uma teoria ou por outra.

Existe um limite de quantas colunas uma tabela pode conter. Dependendo dos tipos das colunas, pode ser entre 250 e 1600. Entretanto, definir uma tabela com esta quantidade de colunas é muito raro e, geralmente, torna o projeto questionável.

Se uma tabela não é mais necessária, pode ser removida pelo comando DROP TABLE. Por exemplo:

```
DROP TABLE minha_primeira_tabela;
DROP TABLE produtos;
```

Tentar excluir uma tabela não existente gera erro. Entretanto, é comum os arquivos de comandos SQL tentarem excluir incondicionalmente a tabela antes de criá-la, ignorando a mensagem de erro.

Utilizando as ferramentas discutidas até este ponto é possível criar tabelas totalmente funcionais. O restante deste capítulo está relacionado com a adição de funcionalidades na definição da tabela para garantir a integridade dos dados, a segurança, ou a conveniência.

2. Colunas do sistema

Toda tabela possui diversas colunas do sistema, as quais são implicitamente definidas pelo sistema. Portanto, estes nomes não podem ser utilizados como nomes de colunas definidas pelo usuário (observe que estas restrições são distintas do nome ser uma palavra chave ou não; colocar o nome entre aspas não faz esta restrição deixar de ser aplicada). Os usuários não precisam se preocupar com estas colunas, basta apenas saber que existem.

oid

O identificador de objeto (object ID) de uma linha. É um número serial automaticamente adicionado pelo PostgreSQL a todas as linhas da tabela (a não ser que a tabela seja criada com WITHOUT OIDS; neste caso esta coluna não estará presente). O tipo desta coluna é oid (o mesmo nome da coluna).

tableoid

O OID da tabela que contém esta linha. Este atributo é particularmente útil nas consultas fazendo seleção em hierarquias de herança, porque sem este atributo é difícil saber de qual tabela cada linha se origina. Pode ser feita uma junção entre tableoid e a coluna oid de pg_class para obter o nome da tabela.

xmin

O identificador da transação de inserção (transaction ID) desta tupla (Nota: neste contexto, a tupla é um estado individual de uma linha; cada atualização da linha cria uma nova tupla para a mesma linha lógica).

cmin

O identificador do comando, começando por zero, dentro da transação de inserção.

xmax

O identificador da transação de exclusão (transaction ID), ou zero para uma tupla não excluída. É possível que este campo seja diferente de zero para uma tupla visível: normalmente isto indica que a transação fazendo a exclusão ainda não foi efetivada (commit), ou que uma tentativa de exclusão foi desfeita (rollback).

cmax

O identificador do comando dentro da transação de exclusão, ou zero.

ctid

A localização física da tupla dentro da tabela. Observe que, embora seja possível usar o ctid para

localizar uma tupla muito rapidamente, o ctid da linha muda cada vez que a linha é atualizada ou movida pelo VACUUM FULL. Portanto, o ctid não serve como identificador de linha duradouro. O OID, ou melhor ainda, um número serial definido pelo usuário, deve ser utilizado para identificar logicamente uma linha.

3. Valor padrão

Uma coluna pode possuir um valor padrão. Quando uma nova linha é criada, e nenhum valor é especificado para algumas colunas, o valor padrão de cada uma destas colunas é atribuído à mesma. Também, um comando de manipulação de dados pode requerer explicitamente que seja atribuído o valor padrão a uma coluna, sem saber qual é este valor.

Se nenhum valor padrão for declarado explicitamente, o valor nulo se torna o valor padrão. Isto geralmente faz sentido, porque o valor nulo pode ser considerado como representando um dado desconhecido.

Na definição da tabela, o valor padrão é posicionado após o tipo de dado da coluna. Por exemplo:

```
CREATE TABLE produtos (
   id_produto integer PRIMARY KEY,
   nome       text,
   preco      numeric DEFAULT 9.99
);
```

O valor padrão pode ser uma expressão escalar, avaliada sempre que o valor padrão é atribuído (e não quando a tabela é criada).

4. Restrições

O tipo de dado é uma forma de limitar os dados que podem ser armazenados na tabela. Entretanto, para muitas aplicações esta restrição é abrangente demais. Por exemplo, uma coluna contendo preços de produtos normalmente só pode aceitar valores positivos, mas não existe nenhum tipo de dado que aceita apenas números positivos. 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 relativas aos produtos deve haver apenas uma linha contendo um determinado código de produto.

Para esta finalidade, a linguagem SQL permite definir restrições em colunas e tabelas. As restrições permitem o nível de controle que for necessário sobre os dados de uma tabela. Se o usuário tentar armazenar dados em uma coluna da tabela violando a restrição, ocasiona erro. Isto se aplica até quando o erro é originado pela definição do valor padrão.

Restrição de verificação

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 arbitrária. Por exemplo, pode ser utilizado para permitir apenas valores positivos para os preços:

```
CREATE TABLE produtos (
   id_produto integer,
   nome       text,
   preco      numeric CHECK (preco > 0)
);
```

Como pode ser observado, a definição da restrição está posicionada após o tipo de dado, do mesmo modo que a definição de valor padrão. O valor padrão e as restrições podem estar em qualquer ordem. Uma restrição de verificação é composto pela palavra chave CHECK seguida por uma expressão entre parênteses. A expressão de 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 for desejado alterá-la. A sintaxe é:

```
CREATE TABLE produtos (
   id_produto integer,
   nome       text,
   preco      numeric CONSTRAINT preco_positivo CHECK (preco > 0)
);
```

Portanto, para especificar uma restrição com nome deve ser utilizada a palavra chave CONSTRAINT, seguida por um identificador, seguido por sua vez pela definição da restrição.

Uma restrição de verificação também pode referenciar várias colunas. Supondo que seja desejado armazenar o preço normal e o preço com desconto, e que seja necessário garantir que o preço com desconto seja menor que o preço normal:

```
CREATE TABLE produtos (
   id_produto        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, que não está presa a nenhuma determinada coluna. Em vez disso, aparece como item à parte na lista de colunas separadas por vírgula. As definições das colunas e as definições destas restrições podem aparecer 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. O exemplo acima também pode ser escrito do seguinte modo

```
CREATE TABLE produtos (
   id_produto        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 (
   id_produto        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.

Deve ser observado que a expressão de verificação está satisfeita se o resultado desta expressão for verdade ou se for nulo. Uma vez que quase todas as expressões retornam um resultado nulo quando um dos operandos é nulo, estas expressões não impedem a presença de valores nulos nas colunas com restrição. Para garantir que uma coluna não aceita o valor nulo, deve ser utilizada a restrição de não nulo descrita a seguir.

1. Restrição de não-nulo

Uma restrição de não-nulo simplesmente especifica que uma coluna não pode conter o valor nulo. Um exemplo da sintaxe:

```
CREATE TABLE produtos (
   id_produto integer NOT NULL,
   nome       text    NOT NULL,
   preco      numeric
);
```

Uma restrição de não-nulo é sempre escrita como restrição de coluna. Uma restrição de não-nulo é funcionalmente equivalente a 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 não poder ser dado um nome explícito para uma restrição criada deste modo.

Uma coluna pode possuir mais de uma restrição, bastando simplesmente escrever uma restrição após a outra:

```
CREATE TABLE produtos (
   id_produto integer NOT NULL,
   nome       text    NOT NULL,
   preco      numeric NOT NULL CHECK (preco > 0)
);
```

A ordem das restrições não importa, porque não afeta, necessariamente, a ordem pela qual as restrições são verificadas.

A restrição NOT NULL possui uma inversa: a restrição NULL. Isto não significa que a coluna deve ser nula, 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, e não deve ser utilizada em aplicações 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 arquivo de comandos. Por exemplo, é possível começar com

```
CREATE TABLE produtos (
   id_produto integer NULL,
   nome       text    NULL,
   preco      numeric NULL
);
```

e depois colocar a palavra chave NOT onde for desejado.

Dica: Na maioria dos projetos de banco de dados, a maioria das colunas deve ser especificada como não-nula.

2. Restrição 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 (
   id_produto integer UNIQUE,
   nome       text,
   preco      numeric
);
```

quando escrita como restrição de coluna, e

```
CREATE TABLE produtos (
   id_produto integer,
   nome       text,
   preco      numeric,
   UNIQUE (id_produto)
```

```
);
```

quando escrita como restrição de tabela.

Se uma restrição de unicidade faz referência a um grupo de colunas, as colunas da lista são separadas por vírgula:

```
CREATE TABLE exemplo (
   a integer,
   b integer,
   c integer,
   UNIQUE (a, c)
);
```

Também é possível atribuir nomes às restrições de unicidade:

```
CREATE TABLE produtos (
   id_produto integer CONSTRAINT deve_ser_diferente UNIQUE,
   nome       text,
   preco      numeric
);
```

De um modo geral, uma restrição de unicidade é violada quando existem (pelo menos) duas linhas na tabela onde os valores de cada uma das colunas correspondentes, que fazem parte da restrição, são iguais. Entretanto, os valores nulos não são considerados iguais nesta situação. Isto significa que, na presença de uma restrição de unicidade multicolunas, é possível armazenar-se um número ilimitado de linhas, se estas linhas contiverem um valor nulo em pelo menos uma das colunas da restrição. Este comando 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 aplicações onde se pretenda haver portabilidade.

3. Chave primária

Tecnicamente uma 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 (
   id_produto integer UNIQUE NOT NULL,
   nome       text,
   preco      numeric
);

CREATE TABLE produtos (
   id_produto integer PRIMARY KEY,
   nome       text,
   preco      numeric
);
```

As chaves primárias também podem abranger mais de uma coluna; a sintaxe é similar a das restrições 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 para as linhas da tabela (Uma conseqüência direta da definição da chave primária. Observe que a restrição de unicidade na verdade não fornece um identificador único, porque não exclui os valores nulos). Isto é útil tanto para fins de documentação quanto para aplicações cliente. Por exemplo, uma interface gráfica que permite modificar os valores das linhas provavelmente necessita conhecer a chave primária da tabela para poder identificar as linhas.

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 determina que toda tabela deve ter uma chave primária. Esta regra não é obrigatória no PostgreSQL, mas normalmente é bom segui-la.

4. Integridade Referencial - Chave estrangeira

A restrição de chave estrangeira especifica que o valor da coluna (ou grupo de colunas) deve corresponder a algum valor que existe em uma linha de outra tabela. Diz-se que este comportamento mantém a integridade referencial entre duas tabelas relacionadas.

Supondo que já temos a tabela de produtos utilizada diversas vezes anteriormente:

```
CREATE TABLE produtos (
  id_produto integer PRIMARY KEY,        nome      text,        preco     numeric );
```

Agora vamos supor, também, que existe uma tabela armazenando os pedidos destes produtos, e desejamos garantir que a tabela de pedidos somente contenha pedidos de produtos que realmente existem. Para isso é definida uma restrição de chave estrangeira na tabela pedidos, fazendo referência à tabela produtos:

```
CREATE TABLE pedidos (
   id_pedido  integer PRIMARY KEY,
   id_produto integer REFERENCES produtos (id_produto),
   quantidade integer
);
```

Isto torna impossível criar pedidos com ocorrências de id_produto que não existam na tabela produtos.

Nesta situação é dito que a tabela pedidos é a tabela que faz referência, e a tabela produtos é a tabela referenciada. Da mesma forma existem colunas fazendo referência e sendo referenciadas.

O comando acima pode ser abreviado escrevendo-se

```
CREATE TABLE pedidos (
    id_pedido  integer PRIMARY KEY,
    id_produto integer REFERENCES produtos,
    quantidade integer
);
```

porque na ausência da lista de colunas, a chave primária da tabela referenciada é assumida como sendo a coluna referenciada.

A chave estrangeira também pode conter e referenciar um grupo de colunas. Como usual, é necessário escrever 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 o tipo das colunas na restrição precisam corresponder ao número e tipo das colunas referenciadas.

Uma tabela pode conter mais de uma restrição de chave estrangeira, utilizado para implementar relacionamentos muitos-para-muitos entre tabelas. Supondo que existam as tabelas produto e pedidos, e desejamos permitir que um pedido contenha vários produtos (o que não é possível na estrutura acima) podemos, então, utilizar a seguinte estrutura de tabela:

```
CREATE TABLE produtos (
    id_produto integer PRIMARY KEY,
    nome       text,
    preco      numeric
);

CREATE TABLE pedidos (
    id_pedido integer PRIMARY KEY,
    endereco_entrega text,
    ...
);

CREATE TABLE itens_pedidos (
    id_produto integer REFERENCES produtos,
    id_pedido  integer REFERENCES pedidos,
```

```
    quantidade integer,
    PRIMARY KEY (id_produto, id_pedido)
);
```

Observe, também, que a chave primária se sobrepõe às chaves estrangeiras na última tabela.

Sabemos que a chave estrangeira não permite a criação de pedidos sem relacionamento com um 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 a especificação desta 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 muitos-para-muitos acima: Quando alguém desejar excluir um produto referenciado por um pedido (através de itens_pedidos), não será permitido. Se alguém excluir um pedido, os itens do pedido também serão removidos.

```
CREATE TABLE produtos (
    id_produto integer PRIMARY KEY,
    nome       text,
    preco      numeric
);

CREATE TABLE pedidos (
    id_pedido       integer PRIMARY KEY,
    endereco_entrega text,
    ...
);

CREATE TABLE pedido_itens (
    id_produto integer REFERENCES produtos ON DELETE RESTRICT,
    id_pedido  integer REFERENCES pedidos  ON DELETE CASCADE,
    quantidade integer,
    PRIMARY KEY (id_produto, id_pedido)
);
```

Restringir ou excluir em cascata são as duas opções mais comuns. RESTRICT também pode ser escrito na forma NO ACTION, e também é o padrão se nada for especificado. Existem duas outras opções relativas ao que deve acontecer com as colunas da chave estrangeira quando a chave primária é excluída: SET NULL e SET DEFAULT. Observe que isto não livra da obediência às restrições. Por exemplo, se uma ação especificar SET DEFAULT, mas o valor padrão não satisfizer a chave estrangeira, a exclusão da chave primária vai falhar.

Semelhante a ON DELETE existe também ON UPDATE, chamada quando uma chave primária é alterada. As ações possíveis são as mesmas.

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 com relação a como os valores nulos são tratados. Esta parte está explicada na entrada CREATE TABLE no Manual de Referência do PostgreSQL.

5. Herança

Vamos criar duas tabelas. A tabela capitais contém as capitais dos estados, que também são cidades. Por conseguinte, 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, uma linha da tabela capitais herda todos os atributos (nome, população e altitude) de sua tabela ancestral cidades. O tipo do atributo nome é text, um tipo nativo do PostgreSQL para cadeias de caracteres ASCII de tamanho variável. O tipo do atributo populacao é float, um tipo nativo do PostgreSQL para números de ponto flutuante de precisão dupla. As capitais dos estados possuem um atributo extra chamado estado, contendo a sigla do estado. No PostgreSQL uma tabela pode herdar de nenhuma, de uma, ou de várias tabelas, e uma consulta pode acessar todas as linhas de uma tabela, ou todas as linhas de uma tabela mais as linhas de suas tabelas descendentes.

Nota: A hierarquia de herança é na verdade um grafo acíclico dirigido. [8]

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

8 Grafo: uma coleção de vértices e arestas; Grafo dirigido: um grafo com arestas unidirecionais; Grafo acíclico dirigido: um grafo dirigido que não contém ciclos - FOLDOC - Free On-Line Dictionary of Computing (N.T.)

Por outro lado, a consulta abaixo retorna todas as cidades que não são capitais de estados situadas a uma altitude superior a 500 pés:

```
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 discutidos até agora -- SELECT, UPDATE e DELETE -- permitem esta notação incluindo ONLY.

Em alguns casos pode ser desejado saber de qual tabela uma determinada tupla 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;
```

que retorna:

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 ver 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;
```

que retorna:

relname	nome	altitude
cities	Las Vegas	2174
cities	Mariposa	1953
capitals	Madison	845

Obsoleto: Em versões anteriores do PostgreSQL o padrão era não acessar as tabelas descendentes. Isto ocasionava muitos erros, e também era uma violação do padrão SQL. Utilizando a sintaxe antiga, para acessar as tabelas descendentes devia ser adicionado um * no final do nome da tabela.

Por exemplo:

```
SELECT * from cidades*;
```

Ainda é possível especificar explicitamente a varredura das tabelas descendentes escrevendo o *, assim como especificar explicitamente a não varredura das tabelas descendentes escrevendo "ONLY" mas, a partir da versão 7.1, o comportamento padrão para um nome de tabela sem adornos passou a ser varrer as tabelas descendentes também, enquanto nas versões anteriores o padrão era não efetuar esta varredura. Para ativar o comportamento antigo deve ser definida a opção de configuração SQL_Inheritance como off. Por exemplo usar

SET SQL_Inheritance TO OFF;

ou adicionar uma linha ao arquivo postgresql.conf.

Uma limitação da funcionalidade da herança é que os índices (incluindo as restrições de unicidade) e chaves estrangeiras somente se aplicam a própria tabela, e não às suas descendentes. Portanto, no exemplo acima, especificando-se que uma coluna de outra tabela REFERENCES cidades(nome) permite à outra tabela conter nomes de cidades, mas não nomes das capitais. Esta deficiência deverá, provavelmente, ser corrigida em alguma versão futura.

5. Modificação de tabelas

Quando percebemos, após a tabela ter sido criada, que foi cometido um erro ou que as necessidades da aplicação mudaram, é possível remover a tabela e criá-la novamente. Porém, este procedimento não é conveniente quando existem dados na tabela, ou se a tabela é referenciada por outros objetos do banco de dados (por exemplo, uma restrição de chave estrangeira). Para esta finalidade o PostgreSQL disponibiliza um conjunto de comandos que realizam modificações em tabelas existentes.

Pode ser feito:

- Incluir coluna;
- Excluir coluna;
- Incluir restrição;
- Excluir restrição;
- Mudar valor padrão;
- Mudar nome de coluna;
- Mudar nome de tabela.

Todas estas atividades são realizadas utilizando o comando ALTER TABLE.

Incluir coluna

Para incluir uma coluna deve ser utilizado o comando:

ALTER TABLE produtos ADD COLUMN descricao text;

Inicialmente a nova coluna conterá valores nulos nas linhas existentes na tabela.

Também pode ser definida uma restrição para a coluna quando esta é incluída utilizando a sintaxe usual:

ALTER TABLE produtos ADD COLUMN descricao text CHECK (descricao <> '');

A nova coluna não pode possuir a restrição de não-nulo, porque a coluna inicialmente deve conter valores nulos. Porém, a restrição de não-nulo pode ser adicionada posteriormente. Também não pode ser definido um valor padrão para a nova coluna. De acordo com o padrão SQL esta definição deve fazer, nas novas colunas, as linhas existentes serem preenchidas com o valor padrão, mas ainda não está implementado. Porém, o valor padrão para a coluna pode ser especificado posteriormente.

Excluir coluna

Para excluir uma coluna deve ser utilizado o comando:

ALTER TABLE produtos DROP COLUMN descricao;

Incluir restrição

É utilizada a sintaxe de restrição de tabela para incluir uma nova restrição. Por exemplo:

ALTER TABLE produtos ADD CHECK (nome <> '');
ALTER TABLE produtos ADD CONSTRAINT algum_nome UNIQUE (id_produto);
ALTER TABLE produtos ADD FOREIGN KEY (id_grupo_produto) REFERENCES grupo_produtos;

Para adicionar uma restrição de não nulo, que não pode ser escrita na forma de restrição de tabela, deve ser utilizada a sintaxe:

ALTER TABLE produtos ALTER COLUMN id_produto SET NOT NULL;

A restrição será verificada imediatamente, portanto os dados da tabela devem satisfazer a restrição antes desta ser criada.

Excluir restrição

Para excluir uma restrição é necessário conhecer seu nome. Quando o usuário atribuiu um nome à restrição é fácil, caso contrário o sistema atribui para a restrição um nome gerado que precisa ser descoberto. O comando \d nome_da_tabela do psql pode ser útil nesta situação; outras interfaces também podem oferecer um modo de inspecionar os detalhes das tabelas. O comando a ser utilizado para excluir a restrição é:

ALTER TABLE produtos DROP CONSTRAINT nome_restrição;

Esta sintaxe serve para todos os tipos de restrição exceto não-nulo. Para excluir uma restrição de

não-nulo deve ser utilizado o comando:

ALTER TABLE produtos ALTER COLUMN id_produto DROP NOT NULL;

(Lembre-se que as restrições de não-nulo não possuem nome)

Mudar valor padrão

Para definir um novo valor padrão para a coluna deve ser utilizado o comando:

ALTER TABLE produtos ALTER COLUMN preco SET DEFAULT 7.77;

Para remover o valor padrão deve ser utilizado o comando:

ALTER TABLE produtos ALTER COLUMN preco DROP DEFAULT;

Este último comando equivale definir o valor nulo como sendo o valor padrão, ao menos no PostgreSQL. Como conseqüência, não ocasiona erro remover um valor padrão que não tenha sido definido, porque implicitamente o valor nulo é o valor padrão.

Mudar nome de coluna

Para mudar o nome de uma coluna deve ser utilizado o comando:

ALTER TABLE produtos RENAME COLUMN id_produto TO cod_produto;

Mudar nome de tabela

Para mudar o nome de uma tabela deve ser utilizado o comando:

ALTER TABLE produtos RENAME TO equipamentos;

6. Privilégios

Quem cria o objeto no banco de dados se torna seu dono. Por padrão, apenas o dono do objeto pode realizar alguma operação com o objeto. Para permitir outros usuários utilizarem o objeto, devem ser concedidos privilégios (existem usuários que possuem o privilégio de superusuário, os quais sempre podem acessar qualquer objeto).

Nota: Para mudar o dono de uma tabela, índice, seqüência ou visão deve ser utilizado o comando ALTER TABLE.

Existem muitos privilégios diferentes: SELECT, INSERT, UPDATE, DELETE, RULE, REFERENCES, TRIGGER, CREATE, TEMPORARY, EXECUTE, USAGE e ALL PRIVILEGES. Para obter informações completas relativas aos diferentes tipos de privilégio suportados pelo PostgreSQL deve ser consultada a página de referência do comando GRANT.

O direito de modificar e destruir objetos são exclusivos do seu criador.

Para conceder privilégios é utilizado o comando GRANT. Portanto, se joel é um usuário existente, e contas é uma tabela existente, o privilégio de poder atualizar esta tabela pode ser concedido por meio do comando:

GRANT UPDATE ON contas TO joel;

Para executar este comando o usuário deve ser o dono da tabela. Para conceder privilégios para um grupo deve ser utilizado o comando:

GRANT SELECT ON contas TO GROUP faturamento;

O nome especial de "usuário" chamado PUBLIC pode ser utilizado para conceder privilégios para todos os usuários do sistema. Escrevendo-se ALL no lugar do nome específico do privilégio causa a concessão de todos os privilégios.

Para revogar um privilégio deve ser utilizado o comando REVOKE:

REVOKE ALL ON contas FROM PUBLIC;

Os privilégios especiais do dono da tabela (ou seja, os direitos de DROP, GRANT, REVOKE, etc.) são inerentes à situação de ser o dono, não podendo ser concedidos ou revogados. Porém, o dono da tabela pode decidir revogar seus próprios privilégios como, por exemplo, permitir apenas leitura na tabela para o próprio e para os outros.

7. Esquemas

Um agrupamento de banco de dados do PostgreSQL (instalação) contém um ou mais bancos de dados nomeados. Os usuários e os grupos de usuários são compartilhados por todo o agrupamento, mas nenhum outro dado é compartilhado por todos os bancos de dados. Todas as conexões dos clientes com o servidor podem acessar somente os dados de um único banco de dados, aquele que foi especificado ao estabelecer a 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 mais de um usuário com o mesmo nome, digamos, joel em dois bancos de dados do mesmo agrupamento; mas o sistema pode ser configurado para permitir que o usuário joel acesse somente um dos bancos de dados.

Um banco de dados contém um ou mais esquemas nomeados, os quais por sua vez contém tabelas. Os esquemas também contêm outros tipos de objetos nomeados, incluindo tipos de dado, funções e operadores. O mesmo nome de objeto pode ser utilizado em esquemas diferentes sem conflitos; por exemplo, tanto o esquema_1 quanto o meu_esquema podem conter uma tabela chamada minha_tabela sem que haja conflito. Ao contrário dos bancos de dados, os esquemas não são separados rigidamente: um usuário pode acessar objetos de vários esquemas no banco de dados ao

qual está conectado, caso possua os privilégios necessários para fazê-lo.
Existem diversos motivos pelos quais a utilização de esquemas pode ser desejada:
Para permitir vários usuários utilizarem o mesmo banco de dados sem que um interfira com o outro.
Para organizar objetos em grupos lógicos tornando-os mais gerenciáveis.
Aplicações desenvolvidas por terceiros podem ser colocadas 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.

6. Criação de esquema

Para criar um novo esquema deve ser utilizado o comando CREATE SCHEMA. O nome do esquema é escolhido livremente pelo usuário. Por exemplo:

```
CREATE SCHEMA meu_esquema;
```

Para criar ou acessar objetos em um esquema deve ser escrito um nome qualificado, composto pelo nome do esquema e da tabela separados por um ponto:

```
esquema.tabela
```

Na verdade, a sintaxe mais geral

banco_de_dados.esquema.tabela

também pode ser utilizada, mas atualmente é apenas uma conformidade pró-forma com o padrão SQL; se for escrito o nome do banco de dados, este deverá ser o mesmo nome do banco de dados ao qual se está conectado.

Portanto, para criar uma tabela no novo esquema deve ser utilizado o comando:

```
CREATE TABLE meu_esquema.minha_tabela (
 ...
);
```

Esta forma funciona em qualquer lugar onde um nome de tabela é esperado, inclusive nos comandos de modificação de tabela e nos demais comandos de acesso a dados discutidos nos próximos capítulos.

Para excluir um esquema vazio (todos seus objetos já foram excluídos), deve ser utilizado o comando:

```
DROP SCHEMA meu_esquema;
```

Para excluir um esquema juntamente com todos os objetos que este contém, deve ser utilizado o comando:

DROP SCHEMA meu_esquema CASCADE;

Muitas vezes deseja-se criar um esquema cujo dono é outro usuário (porque este é um dos modos utilizados para restringir as atividades dos usuários a espaços de nomes bem definidos). A sintaxe para esta operação é:

CREATE SCHEMA nome_do_esquema AUTHORIZATION nome_do_usuário;

Pode inclusive ser omitido o nome do esquema, e neste caso o nome do esquema será o mesmo nome do usuário.

Os nomes de esquemas começando por pg_ são reservados para uso pelo sistema, não devendo ser utilizados pelos usuários.

O esquema público

Nas seções anteriores foram criadas tabelas sem que fosse especificado nenhum nome de esquema. Por padrão, estas tabelas (e outros objetos) são automaticamente colocadas no esquema chamado public. Todo banco de dados novo possui este esquema. Portanto, as duas formas abaixo são equivalentes:

CREATE TABLE produtos (...);
e
CREATE TABLE public.produtos (...);

O caminho de procura do esquema

Os nomes qualificados são desagradáveis de escrever sendo melhor, geralmente, não ficar preso a um determinado esquema em uma aplicação. Portanto, as tabelas são geralmente referenciadas por meio de nomes não qualificados, composto apenas pelo nome da tabela. O sistema determina qual tabela está sendo referenciada seguindo o caminho de procura, o qual é uma lista de esquemas a ser pesquisados. A primeira tabela correspondente no caminho de procura é assumida como sendo a desejada. Não havendo nenhuma correspondência no caminho de procura ocasiona erro, mesmo que a tabela correspondente exista em outro esquema deste banco de dados.

O primeiro esquema nomeado do caminho de procura é chamado de esquema corrente. Além de ser o primeiro esquema a ser pesquisado, também é o esquema no qual as novas tabelas serão criadas se o comando CREATE TABLE não especificar o nome do esquema.

Para ver o caminho de procura corrente deve ser utilizado o comando:

SHOW search_path;

Na configuração padrão este comando retorna:

```
 search_path
--------------
 $user,public
```

O primeiro elemento especifica que o esquema cujo nome é o mesmo nome do usuário corrente deve ser pesquisado. Uma vez que este esquema ainda não foi criado, esta entrada é ignorada. O segundo elemento se refere ao esquema público visto anteriormente.

O primeiro esquema existente no caminho de procura é o local padrão para a criação dos novos objetos. Esta é a razão pela qual os objetos padrão são criados no esquema público. Quando os objetos são referenciados em qualquer outro contexto sem que haja qualificação pelo esquema (comandos de modificação de tabelas, modificação de dados ou consultas) o caminho de procura é percorrido até que o objeto correspondente seja encontrado. Portanto, na configuração padrão, qualquer acesso não qualificado somente pode fazer referência ao esquema público.

Para incluir o novo esquema no caminho deve ser utilizado o comando:

SET search_path TO meu_esquema,public;

O esquema $user foi omitido porque não há necessidade imediata dele. Fazendo isto, as tabelas do esquema meu_esquema podem ser acessadas sem serem qualificadas pelo esquema:

DROP TABLE minha_tabela;

Novamente, uma vez que meu_esquema é o primeiro elemento do caminho, os novos objetos serão criados neste esquema por padrão.

Também é possível escrever

SET search_path TO meu_esquema;

para retirar o acesso ao esquema público, a não ser que seja feita uma qualificação explícita deste esquema. Não existe nada em especial com relação ao esquema público, a não ser que existe por padrão. Também pode ser excluído.

O caminho de procura funciona para nomes de tipos de dado, nomes de funções e nomes de operadores do mesmo modo que funciona para nomes de tabelas. Os tipos de dado e os nomes de funções podem ser qualificados do mesmo modo que os nomes das tabelas. Se for necessário escrever um nome qualificado de operador na expressão existe um modo especial para fazê-lo: deve ser escrito

OPERATOR(esquema.operador)

Isto é necessário para evitar uma sintaxe ambígua. Um exemplo pode ser

SELECT 3 OPERATOR(pg_catalog.+) 4;

Na prática confia-se no caminho de procura para os operadores, não havendo necessidade de escrever nada tão feio assim.

Esquemas e privilégios

Por padrão, os usuários não conseguem enxergar os objetos dos esquemas que não possuem. Para poderem enxergar, o dono do esquema deve conceder o privilégio USAGE para o esquema. Para permitir os usuários utilizarem os objetos do esquema é necessário conceder privilégios adicionais, apropriados para cada objeto.

Também pode ser permitido que um usuário crie objetos no esquema de outro usuário. Para permitir que isto seja feito deve ser concedido o privilégio CREATE para o esquema. Observe que por padrão todos possuem o privilégio CREATE para o esquema public. Isto permite a todos os usuários que podem se conectar ao banco de dados criar objetos neste banco de dados. Se isto não for desejado, este privilégio pode ser revogado:

REVOKE CREATE ON public FROM PUBLIC;

O primeiro "public" acima é o nome do esquema, enquanto o segundo "PUBLIC" significa "todos os usuários". Na primeira ocorrência é um identificador, enquanto na segunda ocorrência é uma palavra reservada. Por isso, na primeira vez está escrito em minúsculas enquanto na segunda vez está em maiúscula.

O esquema do catálogo do sistema

Além do esquema public e dos esquemas criados pelos usuários, cada banco de dados possui um esquema chamado pg_catalog, contendo as tabelas do sistema e todos os tipos de dado, funções e operadores nativos. O pg_catalog é sempre uma parte efetiva do caminho de procura. Se não for colocado explicitamente no caminho de procura, então é implicitamente procurado antes dos esquemas do caminho de procura. Isto garante que os nomes nativos sempre poderão ser encontrados. Entretanto, é possível colocar explicitamente o pg_catalog no final do caminho de procura, se for desejado que os nomes definidos pelo usuário prevaleçam sobre os nomes nativos.

Nas versões do PostgreSQL anteriores a 7.3, os nomes das tabelas começando por pg_ eram reservados. Isto não é mais verdade: estas tabelas podem ser criadas em qualquer esquema que não seja o do sistema. Entretanto, é melhor continuar evitando estes nomes, para garantir que não haverá conflito caso alguma versão futura defina um catálogo do sistema com o mesmo nome da tabela criada (com o caminho de procura padrão, uma referência não qualificada à tabela criada será resolvida por um catálogo do sistema). Os catálogos do sistema vão continuar utilizando a convenção de possuir nomes começando por pg_, não havendo conflito com os nomes não qualificados das tabelas dos usuários, desde que os usuários evitem utilizar o prefixo pg_.

Formas de utilização

Os esquemas podem ser utilizados para organizar os dados de várias maneiras. Existem algumas formas de utilização recomendadas e que são facilmente suportadas pela configuração padrão:

Se não for criado nenhum esquema, então todos os usuários acessam o esquema público implicitamente, simulando a situação onde os esquemas não estão disponíveis. Esta configuração é recomendada, principalmente, quando existe no banco de dados apenas um usuário ou alguns

poucos usuários colaborativos. Esta configuração também permite uma transição suave de uma situação sem esquemas.

Pode ser criado um esquema para cada usuário com o mesmo nome do usuário. Lembre-se que o caminho de procura padrão começa por $user, que representa o nome do usuário. Portanto, se cada usuário possuir um esquema próprio, vão acessar seus próprios esquemas por padrão.

Se esta configuração for utilizada, também pode ser revogado o acesso ao esquema público (ou mesmo removê-lo), ficando os usuários totalmente restritos aos seus próprios esquemas.

A instalação de aplicações compartilhadas (tabelas utilizadas por todos, funções adicionais fornecidas por terceiros, etc.), deve ser feita em esquemas separados. Devem ser concedidos, também, os privilégios necessários para permitir acesso aos outros usuários. Os outro usuários poderão, então, referenciar estes objetos qualificando seus nomes com o nome do esquema, ou poderão adicionar esquemas ao caminho de procura, conforme julgarem melhor.

Portabilidade

No padrão SQL, a noção de objetos no mesmo esquema possuídos por usuários diferentes não existe. Além disso, algumas implementações não permitem a criação de esquemas com nome diferente do nome do seu dono. Na verdade, os conceitos de esquema e de usuário são praticamente equivalentes em sistemas de banco de dados que implementam apenas o suporte ao esquema básico especificado no padrão. Portanto, muitos usuários consideram os nomes qualificados na verdade compostos por nome_do_usuário.nome_da_tabela. Esta é a forma como o PostgreSQL se comportará efetivamente se for criado um esquema por usuário para todos os usuários.

Além disso, não existe o conceito do esquema public no padrão SQL. Para uma conformidade total com o padrão não deve ser utilizado (talvez deva até ser removido) o esquema public.

Obviamente, alguns sistemas de banco de dados SQL podem não implementar esquemas de nenhuma maneira, ou oferecer suporte a espaços de nomes permitindo apenas um acesso (possivelmente limitado) entre bancos de dados. Se for necessário trabalhar com estes sistemas, o máximo de portabilidade é obtido não utilizando nada relacionado aos esquemas.

8. Outros objetos de banco de dados

As tabelas são os objetos centrais da estrutura de um banco de dados relacional, porque armazenam os dados. Porém, as tabelas não são os únicos objetos que existem no banco de dados. Diversos objetos de outros tipos podem ser criados, para tornar o uso e o gerenciamento dos dados mais eficiente, ou mais conveniente. Estes outros objetos não são discutidos neste capítulo, mas são listados para ficarem conhecidos.

Visões (Views)
Funções, operadores, tipos de dado, domínios
Gatilhos e regras de reescrita

Seguindo as dependências

Ao criar uma estrutura de banco de dados complexa, envolvendo muitas tabelas com restrições de chave estrangeira, visões, gatilhos, funções, etc. cria-se, implicitamente, uma rede de dependências entre os objetos. Por exemplo, uma tabela com uma restrição de chave estrangeira depende da tabela referenciada.

Para garantir a integridade de toda a estrutura do banco de dados, o PostgreSQL não permite a exclusão de um objeto havendo outros objetos dependentes dele. Por exemplo, tentar remover a tabela produtos conforme declarada anteriormente, onde a tabela pedidos depende dela, produz uma mensagem de erro como mostrada abaixo:

```
DROP TABLE produtos;
NOTICE:  constraint $1 on table pedidos depends on table produtos
ERROR:  Cannot drop table produtos because other objects depend on it
        Use DROP ... CASCADE to drop the dependent objects too
```

A mensagem de erro mostra uma dica útil: Se não tem importância a exclusão de todos os objetos dependentes, então pode ser executado

```
DROP TABLE produtos CASCADE;
```

e todos os objetos dependentes serão removidos. Neste caso, não será removida a tabela pedidos, será removida apenas a restrição de chave estrangeira (caso se deseje saber o que DROP ... CASCADE fará, deve ser executado o comando DROP sem o CASCADE, e lida a mensagem NOTICE).

Todos os comandos de exclusão do PostgreSQL suportam a especificação de CASCADE. Obviamente, a natureza das dependências possíveis varia conforme o tipo do objeto. Pode ser escrito RESTRICT em vez de CASCADE para obter o comportamento padrão de impedir a exclusão do objeto quando existem objetos dependentes.

Nota: No padrão SQL a especificação de RESTRICT ou CASCADE é obrigatória. Nenhum banco de dados implementa deste modo, mas fazer RESTRICT ou CASCADE o comportamento padrão varia entre os sistemas.

Nota: As dependências de chave estrangeira e as dependências de coluna serial das versões do PostgreSQL anteriores a 7.3, não são mantidas ou criadas durante o processo de atualização de versão. Todos os outros tipos de dependência são criados de forma apropriada durante a atualização de versão.

CAPÍTULO 6. Manipulação de dados

No capítulo anterior foi visto como criar tabelas e outras estruturas de armazenamento de dados. Agora chegou a hora de colocar dados nas tabelas. Este capítulo abrange a inclusão, atualização e exclusão de dados em tabelas. Também são introduzidas maneiras automáticas de realizar mudanças quando certos eventos acontecem: gatilhos (triggers) e regras de reescrita (rewrite rules). Para completar, o próximo capítulo explica como fazer consultas para extrair os dados do banco de dados.

Inclusão de dados

Ao ser criada a tabela não contém nenhum dado. A primeira coisa a ser feita para o banco de dados ser útil é colocar dados. Conceitualmente, os dados são inseridos uma linha por vez. É claro que pode ser inserida mais de uma linha, mas não existe modo de inserir menos de uma linha de cada vez. Mesmo conhecendo apenas o valor de algumas colunas, uma linha inteira deve ser criada.

Para criar uma nova linha deve ser utilizado o comando INSERT. Este comando requer o nome da tabela, e um valor para cada coluna da tabela. Por exemplo, considere a tabela produtos anteriormente descrita:

```
CREATE TABLE produtos (
   id_produto integer,
   nome       text,
   preco      numeric
);
```

Um comando mostrando a inclusão de uma linha pode ser:

```
INSERT INTO produtos VALUES (1, 'Queijo', 9.99);
```

Os valores dos dados são colocados na mesma ordem que as colunas aparecem na tabela, separados por vírgula. Geralmente os valores dos dados são literais (constantes), mas expressões escalares também são permitidas.

A sintaxe mostrada acima tem como desvantagem necessitar o conhecimento da ordem das colunas da tabela. Para evitar isto, as colunas podem ser declaradas explicitamente. Por exemplo, os dois comandos mostrados abaixo possuem o mesmo efeito do comando mostrado acima:

```
INSERT INTO produtos (id_produto, nome, preco) VALUES (1, 'Queijo', 9.99);
INSERT INTO produtos (nome, preco, id_produto) VALUES ('Queijo', 9.99, 1);
```

Muitos usuários consideram boa prática escrever sempre os nomes das colunas.
Se não forem conhecidos os valores de todas as colunas, as colunas com valor desconhecido podem ser omitidas. Neste caso, o valor padrão de cada coluna será atribuído à coluna. Por exemplo:

```
INSERT INTO produtos (id_produto, nome) VALUES (1, 'Queijo');
INSERT INTO produtos VALUES (1, 'Queijo');
```

A segunda forma é uma extensão do PostgreSQL, que preenche as colunas a partir da esquerda com quantos valores forem fornecidos, e as demais com o valor padrão.

Para ficar mais claro, o valor padrão pode ser requisitado explicitamente para cada coluna, ou para toda a linha:

```
INSERT INTO produtos (id_produto, nome, preco) VALUES (1, 'Queijo', DEFAULT);
INSERT INTO produtos DEFAULT VALUES;
```

Dica: Para realizar "cargas volumosas", ou seja, a inclusão de muitos dados, veja o comando COPY (consulte o Manual de Referência do PostgreSQL). O comando COPY não é tão flexível quanto o comando INSERT, mas é mais eficiente.

Atualização de dados

A modificação dos dados armazenados no banco de dados é referida como atualização. Pode ser atualizada uma linha da tabela, todas as linhas da tabela, ou um subconjunto das linhas. Cada coluna pode ser atualizada individualmente; as outras colunas não são afetadas.

Para efetuar uma atualização são necessárias três informações:

- O nome da tabela e da coluna;
- O novo valor para a coluna;
- Quais linhas serão atualizadas.

Lembre-se que o SQL, por si só, não fornece um identificador único para as linhas. Portanto, não é sempre possível poder especificar diretamente a linha que será atualizada. Em vez disso, devem ser especificadas as condições que a linha deve atender para ser atualizada. Somente havendo uma chave primária na tabela (não importando se foi declarada ou não) é possível endereçar uma linha específica, escolhendo uma condição corresponte à chave primária. Ferramentas gráficas de acesso a banco de dados dependem disto para poder atualizar as linhas individualmente.

Por exemplo, o comando mostrado abaixo atualiza todos os produtos com preço igual a 5, mudando estes preços para 10:

```
UPDATE produtos SET preco = 10 WHERE preco = 5;
```

Este comando faz nenhuma, uma, ou muitas linhas serem atualizadas. Não é errado tentar fazer uma atualização sem nenhuma linha correspondente.

Vejamos este comando em detalhe: Primeiro aparece a palavra chave UPDATE seguida pelo nome da tabela. Como usual, o nome da tabela pode ser qualificado pelo esquema, senão é procurado no caminho. Depois aparece a palavra chave SET, seguida pelo nome da coluna, por um sinal de igual, e do novo valor da coluna. O novo valor da coluna pode ser qualquer expressão escalar, e não

apenas uma constante. Por exemplo, sendo desejado aumentar o preço de todos os produtos em 10% pode ser usado:

UPDATE produtos SET preco = preco * 1.10;

Como pode ser visto, a expressão para obter o novo valor também pode referenciar o valor antigo. Também foi deixada de fora a cláusula WHERE. Quando a cláusula WHERE é omitida significa que todas as linhas da tabela serão atualizadas. Quando está presente, apenas as linhas atendendo a condição escrita após o WHERE serão atualizadas. Observe que o sinal de igual na cláusula SET é uma atribuição, enquanto o sinal de igual na cláusula WHERE é uma comparação, mas isto não causa ambigüidade. Obviamente, esta condição não precisa ser um teste de igualdade. Vários outros operadores estão disponíveis (veja o Capítulo 6), mas a expressão precisa produzir um resultado booleano.

Também pode ser atualizada mais de uma coluna pelo comando UPDATE, colocando mais de uma atribuição na cláusula SET. Por exemplo:

UPDATE minha_tabela SET a = 5, b = 3, c = 1 WHERE a > 0;

Exclusão de dados

Até aqui mostrou-se como adicionar dados em tabelas, e como modificar estes dados. Está faltando mostrar como remover os dados que não são mais necessários. Assim como só é possível adicionar dados para toda uma linha, também uma linha só pode ser removida por completo de uma tabela. Na seção anterior foi visto que o SQL não oferece funcionalidade para endereçar diretamente linhas específicas. Portanto, a remoção de linhas só pode ser feita por meio da especificação das condições que as linhas a ser removidas devem atender. Havendo uma chave primária na tabela, então é possível especificar exatamente a linha. Mas também pode ser removido um grupo de linhas atendendo a uma determinada condição, ou podem ser removidas todas as linhas da tabela de uma só vez.

É utilizado o comando DELETE para excluir linhas; a sintaxe deste comando é muito semelhante a do comando UPDATE. Por exemplo, para excluir todas as linhas da tabela produtos possuindo preço igual a 10, usa-se:

DELETE FROM produtos WHERE preco = 10;

Se for escrito simplesmente

DELETE FROM produtos;

então todas as linhas da tabela serão excluídas! Dica de programador.

CAPÍTULO 7. Consultas

Os capítulos anteriores explicaram como criar tabelas, como preenchê-las com dados, e como manipular estes dados. Neste capítulo será discutido como trazer estes dados para fora do banco de dados.

Visão geral

O processo de trazer de volta, ou o comando para trazer os dados armazenados no banco de dados, é chamado de consulta. No SQL, o comando SELECT é utilizado para especificar as consultas. A sintaxe geral do comando SELECT é

SELECT lista_seleção FROM expressão_tabela [especificação_ordenação]

As próximas seções descrevem os detalhes da lista de seleção, da expressão de tabela, e da especificação da ordenação.

O tipo mais simples de consulta possui a forma:

SELECT * FROM tabela1;

Supondo que exista a tabela chamada tabela1, este comando traz todas as linhas e todas as colunas da tabela1. A forma de trazer depende da aplicação cliente. Por exemplo, o programa psql exibe uma tabela ASCII na tela, enquanto as bibliotecas cliente possuem funções para obter linhas e colunas individualmente. O * especificado na lista de seleção significa todas as colunas que a expressão de tabela tem para oferecer. A lista de seleção também pode especificar um subconjunto das colunas disponíveis, ou efetuar cálculos utilizando as colunas. Por exemplo, se tabela1 possui colunas chamadas a, b e c (e talvez outras), pode ser feita a seguinte consulta

SELECT a, b + c FROM tabela1;

supondo que b e c possuem um tipo de dado numérico.

FROM tabela1 é um tipo particularmente simples de expressão de tabela: lê apenas uma tabela. De modo geral, as expressões de tabela podem ser construções complexas de tabelas base, junções e subconsultas. Mas a expressão de tabela pode ser totalmente omitida quando se deseja utilizar o comando SELECT como uma calculadora:

SELECT 3 * 4;

Isto é mais útil quando a expressão da lista de seleção retorna resultados variáveis. Por exemplo, uma função pode ser chamada deste modo:

SELECT random();

Expressão de tabela

Uma expressão de tabela produz uma tabela. A expressão de tabela contém uma cláusula FROM seguida, opcionalmente, pelas cláusulas WHERE, GROUP BY e HAVING. As expressões de tabela triviais simplesmente fazem referência às tabelas em disco, chamadas de tabelas base, mas expressões mais complexas podem ser utilizadas para modificar ou combinar tabelas base de várias maneiras.

As cláusulas opcionais WHERE, GROUP BY e HAVING da expressão de tabela especificam um processo de transformações sucessivas realizadas na tabela produzida pela cláusula FROM. Estas transformações produzem uma tabela virtual que fornece as linhas passadas para a lista de seleção, para então serem produzidas as linhas de saída da consulta.

A cláusula FROM

A cláusula FROM produz uma tabela a partir de uma ou mais tabelas especificadas na lista, separada por vírgulas, de referências de tabela.

FROM referência_tabela [, referência_tabela [, ...]]

Uma referência de tabela pode ser um nome de tabela (possivelmente qualificado pelo esquema), ou uma tabela derivada como uma subconsulta, uma junção de tabelas ou, ainda, uma combinação complexa destas. Se mais de uma referência de tabela estiver presente na cláusula FROM é feita uma junção cruzada (cross-join) (veja abaixo) para produzir uma tabela virtual intermediária que, então, estará sujeita às transformações especificadas nas cláusulas WHERE, GROUP BY e HAVING, gerando o resultado final de toda a expressão de tabela.

Quando uma referência de tabela especifica uma tabela ancestral em uma hierarquia de herança de tabelas, a tabela referenciada não produz linhas de apenas uma tabela, mas inclui as linhas de todas as tabelas descendentes, a não ser que a palavra chave ONLY preceda o nome da tabela. Entretanto, esta referência produz apenas as colunas que aparecem na tabela especificada --- todas as colunas adicionadas às tabelas descendentes são ignoradas.

Junção de tabelas

Uma tabela juntada é uma tabela derivada de outras duas tabelas (reais ou derivadas), de acordo com as regras do tipo de junção. Estão disponíveis as junções internas, externas e cruzadas.

Tipos de junção

Junção cruzada
T1 CROSS JOIN T2

Para cada combinação de linha de T1 com T2, a tabela derivada contém uma linha formada por todas as colunas de T1 seguidas por todas as colunas de T2. Se as tabelas possuirem N e M linhas, respectivamente, a tabela juntada terá N * M linhas. Uma junção cruzada é equivalente a INNER JOIN ON TRUE.

Dica: FROM T1 CROSS JOIN T2 é equivalente a FROM T1, T2.

Junção qualificada
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 ON expressão_booleana
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 USING (lista de colunas de junção)
T1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2

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

A cláusula ON é o tipo mais geral de condição de junção: recebe o valor de uma expressão booleana do mesmo tipo utilizado na cláusula WHERE. Um par de linhas de T1 e T2 são correspondentes, se a expressão da cláusula ON produz um resultado verdade para este par de linhas.

USING é uma notação abreviada: recebe uma lista separada por vírgulas contendo nomes de colunas que as tabelas juntadas possuem em comum, formando a condição de junção especificando a igualdade de cada par destas colunas. Além disso, a saída de JOIN USING possui apenas uma coluna para cada par da igualdade de colunas de entrada, seguidas por todas as outras colunas de cada tabela. Portanto, USING (a, b, c) é equivalente a ON (t1.a = t2.a AND t1.b = t2.b AND t1.c = t2.c), mas quando ON for utilizado vão existir duas colunas a, b e c no resultado, enquanto com USING existirá apenas uma de cada.

Finalizando, NATURAL é uma forma abreviada de USING: gera uma lista para o USING formada pelas colunas cujos nomes existem nas duas tabelas de entrada. Assim como no USING, estas colunas aparecem somente uma vez na tabela de saída.

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

RIGHT OUTER JOIN

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

FULL OUTER JOIN

Primeiro, uma junção interna é realizada. Depois, para cada linha de T1 que não satisfaz a condição de junção com nenhuma linha de T2, uma linha juntada é adicionada com valores nulos nas colunas de T2. Também, para cada linha de T2 que não satisfaz a condição de junção com nenhuma linha de T1, uma linha juntada com valores nulos nas colunas de T1 é adicionada.
As junções de todos os tipos podem ser encadeadas ou aninhadas: tanto T1 como T2, ou ambas, podem ser tabelas juntadas. Parênteses podem colocados em torno das cláusulas JOIN para controlar a ordem de junção. Na ausência de parênteses, as cláusulas JOIN são aninhadas da esquerda para a direita.

Para reunir tudo isto, vamos supor que temos as tabelas t1

num	nome
1	a
2	b
3	c

e t2

num	nome
1	xxx
3	yyy
5	zzz

e mostrar os resultados para vários tipos de junção:

```
=> SELECT * FROM t1 CROSS JOIN t2;
 num | nome | num | valor
-----+------+-----+-------
   1 | a    |   1 | xxx
   1 | a    |   3 | yyy
   1 | a    |   5 | zzz
   2 | b    |   1 | xxx
   2 | b    |   3 | yyy
   2 | b    |   5 | zzz
   3 | c    |   1 | xxx
   3 | c    |   3 | yyy
   3 | c    |   5 | zzz
(9 rows)
```

```
=> SELECT * FROM t1 INNER JOIN t2 ON t1.num = t2.num;
 num | nome | num | valor
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
(2 rows)

=> SELECT * FROM t1 INNER JOIN t2 USING (num);
 num | nome | valor
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 NATURAL INNER JOIN t2;
 num | nome | valor
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num;
 num | nome | num | valor
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |   3 | yyy
(3 rows)

=> SELECT * FROM t1 LEFT JOIN t2 USING (num);
 num | nome | valor
-----+------+-------
   1 | a    | xxx
   2 | b    |
   3 | c    | yyy
(3 rows)

=> SELECT * FROM t1 RIGHT JOIN t2 ON t1.num = t2.num;
 num | nome | num | valor
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
     |      |   5 | zzz
(3 rows)

=> SELECT * FROM t1 FULL JOIN t2 ON t1.num = t2.num;
 num | nome | num | valor
```

```
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |   3 | yyy
     |      |   5 | zzz
(4 rows)
```

A condição de junção especificada em ON também pode conter condições não relacionadas diretamente com a junção. Pode ser útil em algumas consultas, mas deve ser usado com cautela. Por exemplo:

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num AND t2.valor = 'xxx';
 num | nome | num | valor
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |     |
(3 rows)
```

Aliás para tabela e coluna

Nomes temporários podem ser atribuídos a tabelas, e a referências de tabela complexas, para ser utilizado como referência à tabela derivada em processamentos posteriores. Isto é chamado de aliás de tabela.

Para criar um aliás de tabela deve ser escrito

FROM referência_tabela AS aliás

ou

FROM referência_tabela aliás

A palavra chave AS é opcional. O aliás pode ser qualquer identificador.

Uma aplicação típica de aliases de tabelas é atribuir identificadores curtos para nomes longos de tabelas, para manter a cláusula de junção legível. Por exemplo:

SELECT * FROM um_nome_muito_comprido u JOIN outro_nome_muito_comprido o ON u.id = o.num;

O aliás se torna o novo nome de referência da tabela para a consulta corrente -- não é mais possível fazer referência à tabela pelo seu nome original. Portanto,

SELECT * FROM minha_tabela AS m WHERE minha_tabela.a > 5;

não é uma sintaxe SQL válida. O que acontece de verdade (isto é uma extensão do PostgreSQL ao padrão) é que uma referência implícita à tabela é adicionada à cláusula FROM. Portanto, a consulta

é processada como se estivesse escrita assim

SELECT * FROM minha_tabela AS m, minha_tabela AS minha_tabela WHERE minha_tabela.a > 5;

resultando em uma junção cruzada, que provavelmente não é o desejado.

Os aliases de tabela servem principalmente como uma notação conveniente, mas sua utilização é necessária para fazer a junção de uma tabela consigo mesma. Por exemplo:

SELECT * FROM minha_tabela AS a CROSS JOIN minha_tabela AS b ...

Além disso, um aliás é requerido se a referência de tabela é uma subconsulta.

Os parênteses são utilizados para resolver ambigüidades. A declaração abaixo atribui o aliás b ao resultado da junção, de forma diferente do exemplo anterior:

SELECT * FROM (minha_tabela AS a CROSS JOIN minha_tabela) AS b ...

Uma outra forma de aliás para tabela também especifica nomes temporários para as colunas da tabela:

FROM referência_tabela [AS] aliás (coluna1 [, coluna2 [, ...]])

Se for especificado um número menor de aliases de coluna que o número de colunas da tabela, as demais colunas não serão renomeadas. Esta sintaxe é especialmente útil em auto-junções e subconsultas.

Quando um aliás é aplicado a saída de uma cláusula JOIN, utilizando qualquer uma destas formas, o aliás esconde o nome original dentro do JOIN. Por exemplo:

SELECT a.* FROM minha_tabela AS a JOIN sua_tabela AS b ON ...

é um comando SQL válido, mas

SELECT a.* FROM (minha_tabela AS a JOIN sua_tabela AS b ON ...) AS c

não é válido: o aliás de tabela a não é visível fora do aliás c.

Subconsultas

Subconsultas especificando uma tabela derivada devem estar entre parênteses, e precisam possuir um nome de aliás de tabela. Por exemplo:

FROM (SELECT * FROM tabela1) AS nome_aliás

Este exemplo é equivalente a FROM tabela1 AS nome_aliás. Casos mais interessantes, que não podem ser reduzidos a junções simples, ocorrem quando a subconsulta envolve agrupamento ou

agregação.

A cláusula WHERE

A sintaxe da cláusula WHERE é

WHERE condição_pesquisa

onde a condição_pesquisa é qualquer expressão de valor, que retorna um valor do tipo booleano.

Após o processamento da cláusula FROM ter sido realizado, cada linha da tabela virtual derivada é verificada com relação à condição de pesquisa. Se o resultado da condição for verdade, a linha é mantida na tabela de saída, senão (ou seja, se o resultado for falso ou nulo) a linha é rejeitada. A condição de pesquisa tipicamente faz referência a pelo menos uma coluna da tabela gerada pela cláusula FROM; isto não é necessário, mas se não for assim a cláusula WHERE não terá utilidade.

Nota: Antes da implementação da sintaxe do JOIN era necessário colocar a condição de junção, de uma junção interna, na cláusula WHERE. Por exemplo, as duas expressões de tabela abaixo são equivalentes:

FROM a, b WHERE a.id = b.id AND b.val > 5

e

FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5

ou talvez até

FROM a NATURAL JOIN b WHERE b.val > 5

Qual destas formas deve ser utilizada é principalmente uma questão de estilo. A sintaxe do JOIN na cláusula FROM é provavelmente a mais portável para outros produtos de banco de dados SQL. Para as junções externas não existe escolha em nenhum caso: devem ser feitas na cláusula FROM. Uma cláusula ON/USING de uma junção externa não é equivalente a uma condição WHERE, porque determina a adição de linhas (para as linhas de entrada sem correspondência) assim como a remoção de linhas do resultado final.

Abaixo estão mostrados alguns exemplos da cláusula WHERE:

SELECT ... FROM fdt WHERE c1 > 5

SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)

SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)

SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)

SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)

AND 100

SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)

sendo que fdt é a tabela derivada da cláusula FROM. As linhas que não correspondem à condição de pesquisa da cláusula WHERE são eliminadas de fdt. Observe a utilização de subconsultas escalares como expressões de valor. Assim como qualquer outra consulta, as subconsultas podem utilizar expressões de tabela complexas. Observe como fdt é referida nas subconsultas. A qualificação de c1 como fdt.c1 somente é necessária se c1 também for o nome de uma coluna na tabela de entrada derivada da subconsulta. A qualificação do nome da coluna torna mais clara a consulta, mesmo quando não é necessária. Isto mostra como o escopo dos nomes das colunas de uma consulta externa se estende nas suas consultas internas.

As cláusulas GROUP BY e HAVING

Após passar pelo filtro WHERE, a tabela de entrada derivada pode estar sujeita a agrupamento, utilizando a cláusula GROUP BY, e a eliminação de grupos de linhas, utilizando a cláusula HAVING.

```
SELECT lista_seleção
   FROM ...
   [WHERE ...]
   GROUP BY referência_coluna_agrupamento [, referência_coluna_agrupamento]...
```

A cláusula GROUP BY é utilizada para agrupar linhas de uma tabela que compartilham os mesmos valores em todas as colunas listadas. Em que ordem as colunas são listadas não faz diferença. A finalidade é reduzir cada grupo de linhas compartilhando valores comuns a uma única linha agrupada representando todas as linhas do grupo. Isto é feito para eliminar redundância na saída, e/ou para calcular agregações aplicáveis a estes grupos. Por exemplo:

```
=> SELECT * FROM teste1;
 x | y
---+---
 a | 3
 c | 2
 b | 5
 a | 1
(4 rows)

=> SELECT x FROM teste1 GROUP BY x;
 x
---
 a
 b
 c
(3 rows)
```

Na segunda consulta não poderia ser escrito SELECT * FROM teste1 GROUP BY x, porque não

existe um único valor da coluna y que poderia ser associado com cada grupo. As colunas agrupadas podem ser referenciadas na lista de seleção, porque possuem um valor constante conhecido para cada grupo.

De modo geral, se uma tabela é agrupada as colunas que não são usadas nos agrupamentos não podem ser referenciadas, exceto nas expressões de agregação. Um exemplo de expressão de agregação é:

```
=> SELECT x, sum(y) FROM teste1 GROUP BY x;
 x | sum
---+-----
 a |   4
 b |   5
 c |   2
(3 rows)
```

Aqui sum() é a função de agregação que calcula um único valor para o grupo todo.

Dica: Um agrupamento sem expressão de agregação na verdade computa o conjunto de linhas distintas de uma coluna. Também poderia ser obtido por meio da cláusula DISTINCT .

Abaixo está mostrado um outro exemplo: sum(vendas) em uma tabela agrupada pelo código do produto fornece o valor total das vendas de cada produto, e não o total das vendas de todos os produtos.

```
SELECT id_produto, p.nome, (sum(s.unidades) * p.preco) AS vendas
   FROM produtos p LEFT JOIN vendas s USING (id_produto)
   GROUP BY id_produto, p.nome, p.preco;
```

Neste exemplo, as colunas id_produto, p.nome e p.preco devem estar na cláusula GROUP BY, porque são referenciadas na lista de seleção da consulta (dependendo da forma exata como a tabela produtos for definida, as colunas nome e preço podem ser totalmente dependentes da coluna id_produto, tornando os agrupamentos adicionais teoricamente desnecessários, mas isto ainda não está implementado). A coluna s.unidades não precisa estar na lista do GROUP BY, porque é usada apenas na expressão de agregação (sum()), que representa o grupo de vendas do produto. Para cada produto, uma linha é retornada contendo o total de vendas do produto.

No SQL estrito, a cláusula GROUP BY somente pode agrupar pelas colunas da tabela de origem, mas o PostgreSQL estende esta funcionalidade permitindo o GROUP BY agrupar pelas colunas da lista de seleção. O agrupamento por expressões de valor, em vez de nomes simples de colunas, também é permitido.

Se uma tabela for agrupada utilizando a cláusula GROUP BY, mas há interesse em alguns grupos apenas, a cláusula HAVING pode ser utilizada, da mesma forma que a cláusula WHERE, para remover grupos da tabela agrupada. A sintaxe é:

```
SELECT lista_seleção FROM ... [WHERE ...] GROUP BY ... HAVING expressão_booleana
```

As expressões na cláusula HAVING podem fazer referência tanto a expressões agrupadas quanto a expressões não agrupadas (as quais necessariamente envolvem uma função de agregação).

Exemplo:

```
=> SELECT x, sum(y) FROM teste1 GROUP BY x HAVING sum(y) > 3;
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)

=> SELECT x, sum(y) FROM teste1 GROUP BY x HAVING x < 'c';
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)
```

Agora vamos fazer um exemplo mais próximo da realidade:

```
SELECT id_produto, p.nome, (sum(s.unidades) * (p.preco - p.custo)) AS lucro
   FROM produtos p LEFT JOIN vendas s USING (id_produto)
   WHERE s.data > CURRENT_DATE - INTERVAL '4 weeks'
   GROUP BY id_produto, p.nome, p.preco, p.custo
   HAVING sum(p.preco * s.unidades) > 5000;
```

No exemplo acima, a cláusula WHERE está selecionando linhas por uma coluna que não é agrupada, enquanto a cláusula HAVING restringe a saída para os grupos com um total bruto de vendas acima de 5000. Observe que as expressões de agregação não precisam ser necessariamente as mesmas em todos os lugares.

Listas de seleção

Conforme foi mostrado na seção anterior, a expressão de tabela do comando SELECT constrói uma tabela virtual intermediária, possivelmente por meio da combinação de tabelas, visões, eliminação de linhas, agrupamento, etc. Esta tabela é finalmente passada adiante para ser processada pela lista de seleção. A lista de seleção determina quais colunas da tabela intermediária fazem realmente parte da saída.

Itens da lista de seleção

O tipo mais simples de lista de seleção é o *, o qual emite todas as colunas produzidas pela expressão de tabela. Senão, a lista de seleção é uma lista separada por vírgulas de expressões de valor. Por exemplo, esta pode ser uma lista de nomes de colunas:

```
SELECT a, b, c FROM ...
```

Os nomes das colunas a, b e c podem ser os nomes de verdade das colunas das tabelas referenciadas na cláusula FROM, ou aliases destas colunas. O espaço de nome disponível na lista de seleção é o mesmo da cláusula WHERE, a não ser que o agrupamento seja utilizado, caso em que passa a ser o mesmo da cláusula HAVING.

Se mais de uma tabela possuir uma coluna com o mesmo nome, o nome da tabela também deve ser fornecido, como em:

SELECT tbl1.a, tbl2.b, tbl1.c FROM ...

Se uma expressão de valor arbitrária for utilizada na lista de seleção, conceitualmente uma nova coluna virtual é adicionada à tabela retornada. A expressão de valor é avaliada uma vez para cada linha retornada, com os valores da linha substituídos nas referências a colunas. Porém, a expressão da lista de seleção não precisa referenciar nenhuma coluna da expressão de tabela da cláusula FROM; podem ser, inclusive, expressões aritméticas constantes, por exemplo.

Rótulos de coluna

Podem ser atribuídos nomes para as entradas da lista de seleção para processamento posterior. Neste caso "processamento posterior" é uma especificação opcional de ordenação e a aplicação cliente (por exemplo, os títulos das colunas para exibição). Por exemplo:

SELECT a AS valor, b + c AS soma FROM ...

Se nenhum nome de coluna de saída for especificado utilizando AS, o sistema atribui um nome padrão. Para referências simples a colunas é o nome da coluna referenciada. Para chamadas de função é o mesmo nome da função. Para expressões complexas o sistema gera um nome genérico.

Nota: A nomeação das colunas de saída neste caso é diferente daquela feita na cláusula FROM. Este duto na verdade permite renomear a mesma coluna duas vezes, mas o nome escolhido na lista de seleção é que será passado adiante.

DISTINCT

Após a lista de seleção ser processada, a tabela resultante pode opcionalmente estar sujeita a remoção das linhas duplicadas. A palavra chave DISTINCT deve ser escrita logo após o SELECT para ativar esta funcionalidade:

SELECT DISTINCT lista_seleção ...

(Em vez de DISTINCT a palavra ALL pode ser utilizada para selecionar o comportamento padrão de manter todas as linhas)

Como é óbvio, duas linhas são consideradas distintas quando tiverem pelo menos uma coluna diferente. Os valores nulos são considerados iguais nesta comparação.

Como alternativa, uma expressão arbitrária pode ser utilizada para determinar quais linhas devem ser consideradas distintas:

SELECT DISTINCT ON (expressão [, expressão ...]) lista_seleção ...

Neste caso expressão é uma expressão de valor arbitrária avaliada para todas as linhas. Um conjunto de linhas para as quais todas as expressões são iguais são consideradas duplicadas, e somente a primeira linha do conjunto é mantida na saída. Observe que a "primeira linha" de um conjunto não pode ser predita, a não ser que a consulta seja ordenada por um número suficiente de colunas para garantir a ordenação única das linhas que chegam no filtro DISTINCT (o processamento de DISTINCT ON ocorre após a ordenação do ORDER BY).

A cláusula DISTINCT ON não é parte do padrão SQL, sendo algumas vezes considerada um estilo ruim devido à natureza potencialmente indeterminada de seus resultados. Utilizando-se adequadamente GROUP BY e subconsultas no FROM esta construção pode ser evitada, mas geralmente é a alternativa mais conveniente.

Combinação de consultas

Os resultados de duas consultas podem ser combinados utilizando as operações com conjuntos [9] união (union), interseção (intersection) e diferença (difference). 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 discutidas anteriormente. As operações com conjuntos também podem ser aninhadas ou encadeadas. Por exemplo:

consulta1 UNION consulta2 UNION consulta3

significa, na verdade,

(consulta1 UNION consulta2) UNION consulta3

UNION anexa o resultado da consulta2 ao resultado da consulta1 (embora não seja garantido que as linhas retornem nesta ordem). Além disso, são eliminadas todas as linhas duplicadas, do mesmo modo que no DISTINCT, a não ser que UNION ALL seja utilizado.

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 INTERSECT ALL seja utilizado.

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 os dois resultados). Novamente, as linhas duplicadas são eliminadas a não ser que EXCEPT ALL seja utilizado.

9 Dados dois conjuntos A e B: chama-se diferença ente A e B o conjunto formado pelos elementos de A que não pertencem a B; chama-se interseção de A com B o conjunto formado pelos elementos comuns ao conjunto A e ao conjunto B; chama-se união de A com B o conjunto formado pelos elementos que pertencem a A ou B. Edwaldo Bianchini e Herval Paccola - Matemática - Operações com conjuntos. (N.T.)

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

Ordenação de linhas

Após a consulta ter produzido uma tabela de saída (após a lista de seleção ter sido processada) esta tabela pode, opcionalmente, ser ordenada. Se nenhuma ordenação for especificada, as linhas retornam em uma ordem aleatória. Na verdade, neste caso a ordem depende dos tipos de plano de varredura e de junção e da ordem no disco, mas não se deve confiar nisto. Uma determinada ordem de saída somente pode ser garantida se a etapa de ordenação for explicitamente especificada.

A cláusula ORDER BY especifica a ordenação:

```
SELECT lista_seleção
   FROM expressão_tabela
   ORDER BY coluna1 [ASC | DESC] [, coluna2 [ASC | DESC] ...]
```

onde coluna1, etc. fazem referência às colunas da lista de seleção. Pode ser tanto o nome de saída de uma coluna quanto o número da coluna. Alguns exemplos:

```
SELECT a, b FROM tabela1 ORDER BY a;
SELECT a + b AS soma, c FROM tabela1 ORDER BY soma;
SELECT a, sum(b) FROM tabela1 GROUP BY a ORDER BY 1;
```

Como uma extensão do padrão SQL, o PostgreSQL também permite a ordenação por expressões arbitrárias:

```
SELECT a, b FROM tabela1 ORDER BY a + b;
```

Também é permitido fazer referência a nomes de colunas da cláusula FROM que foram renomeados na lista de seleção:

```
SELECT a AS b FROM tabela1 ORDER BY a;
```

Mas estas extensões não funcionam nas consultas que envolvem UNION, INTERSECT ou EXCEPT, e não são portáveis para outros bancos de dados SQL.

Cada coluna especificada pode ser seguida pela palavra opcional ASC ou DESC, para determinar a forma de ordenação como ascendente ou descendente. A forma ASC é o padrão. A ordenação ascendente coloca os valores menores na frente, sendo que "menor" é definido nos termos do operador <. De forma semelhante, a ordenação descendente é determinada pelo operador >.

Se mais de uma coluna de ordenação for especificada, as últimas colunas são utilizadas para ordenar as linhas iguais na ordem imposta pelas primeiras colunas ordenadas.

LIMIT e OFFSET

LIMIT e OFFSET permitem trazer apenas uma parte das linhas geradas pela consulta:

```
SELECT lista_seleção
   FROM expressão_tabela
   [LIMIT { número | ALL }] [OFFSET número]
```

Se o limite for fornecido, não mais que esta quantidade de linhas será retornada (mas possivelmente menos, se a consulta produzir menos linhas). LIMIT ALL é o mesmo que omitir a cláusula LIMIT.

OFFSET informa para saltar esta quantidade de linhas antes de começar a retornar as linhas para o cliente. OFFSET 0 é o mesmo que omitir a cláusula OFFSET. Se tanto OFFSET quando LIMIT forem especificados, então são saltadas OFFSET linhas antes de começar a contar as LIMIT linhas que serão retornadas.

Quando se utiliza LIMIT é uma boa idéia utilizar também a cláusula ORDER BY, estabelecendo uma ordem única para as linhas do resultado. Senão, será retornado um subconjunto imprevisível de linhas da consulta --- pode ser desejado obter da décima a vigésima linha, mas da décima a vigésima de qual ordem? A ordenação é desconhecida a não ser que ORDER BY seja especificado.

O otimizador de consultas leva em consideração o LIMIT para gerar o plano da consulta, portanto é bastante provável obter planos diferentes (resultando em uma ordem diferente das linhas) dependendo do que for especificado para LIMIT e OFFSET. Sendo assim, utilizar valores diferentes de LIMIT/OFFSET para selecionar subconjuntos diferentes do resultado de uma consulta produz resultados inconsistentes, a não ser que seja estabelecida uma ordem determinada por meio da cláusula ORDER BY. Isto não está errado; isto é uma conseqüência direta do fato do SQL não prometer retornar os resultados de uma consulta em nenhuma ordem específica, a não ser que ORDER BY seja utilizado para especificar esta ordem.

CAPÍTULO 8. Tipos de dado

O PostgreSQL disponibiliza para os usuários um amplo conjunto de tipos de dado nativos. Os usuários podem adicionar novos tipos ao PostgreSQL utilizando o comando CREATE TYPE.
A tabela abaixo mostra todos os tipos de dado de propósito geral incluídos na distribuição padrão. A maioria dos nomes alternativos listados na coluna "Aliases" é o nome utilizado internamente pelo PostgreSQL por motivos históricos. Além desses, existem alguns tipos usados internamente ou obsoletos não mostrados aqui.

Tabela - Tipos de dado

Nome do tipo	Aliases	Descrição
bigint	int8	inteiro de oito bytes com sinal
bigserial	serial8	inteiro de oito bytes autoincremental
bit		cadeia de bits de comprimento fixo
bit varying(n)	varbit(n)	cadeia de bits de comprimento variável
boolean	bool	booleano lógico (verdade/falso)
box		caixa retangular em plano 2D
bytea		dados binários
character varying(n)	varchar(n)	cadeia de caracteres de comprimento variável
character(n)	char(n)	cadeia de caracteres de comprimento fixo
cidr		endereço de rede IP
circle		círculo em plano 2D
date		data de calendário (ano, mês,dia)
double precision	float8	número de ponto flutuante de precisão dupla
inet		endereço de hospedeiro IP
integer	int, int4	inteiro de quatro bytes com sinal
interval(p)		intervalo de tempo de uso geral
line		linha infinita em plano 2D (não implementado)
lseg		segmento de linha em plano 2D
macaddr		endereço MAC
money		valor monetário
numeric [(p, s)]	decimal [(p, s)]	numérico exato com precisão selecionável
path		caminho geométrico aberto e fechado em plano 2D
point		ponto geométrico em plano 2D
polygon		caminho geométrico fechado em plano 2D
real	float4	número de ponto flutuante de precisão simples

smallint	int2	inteiro de dois bytes com sinal
serial	serial4	inteiro de quatro bytes autoincremental
text		cadeia de caracteres de comprimento variável
time [(p)] [without time zone]	time	hora do dia
time [(p)] with time zone	timetz	hora do dia, incluindo a zona horária
timestamp [(p)] without time zone	timestamp	data e hora
timestamp [(p)] [with time zone]	timestamptz	data e hora, incluindo a zona horária

Compatibilidade: Os seguintes tipos são especificados pelo SQL: bit, bit varying, boolean, char, character varying, character, varchar, date, double precision, integer, interval, numeric, decimal, real, smallint, time, timestamp (ambos com ou sem zona horária).

Cada tipo de dado possui uma representação externa determinada pelas suas funções de entrada e de saída. Muitos tipos nativos possuem um formato externo óbvio. Entretanto, muitos tipos são exclusivos do PostgreSQL, como caminhos abertos e fechados, ou possuem formatos com várias possibilidades, como os tipos de data e hora. A maioria das funções de entrada e de saída correspondentes aos tipos básicos (por exemplo, números inteiros e de ponto flutuante) executa alguma verificação de erro. Algumas das funções de entrada e saída não são inversíveis, ou seja, o resultado da função de saída pode perder precisão quando comparado com a entrada original.

Para melhorar o tempo de execução, alguns operadores e funções (por exemplo, adição e multiplicação) não realizam verificação de erro em tempo de execução. Por isso, em alguns sistemas os operadores numéricos para alguns tipos de dado podem estourar a capacidade sem avisar.

Tipos numéricos

Os tipos numéricos consistem de inteiros de dois, quatro e oito bytes, números de ponto flutuante de quatro e oito bytes, e decimais de precisão fixa.

Tabela - Tipos numéricos

Nome do tipo	Tamanho	Descrição	Faixa de valores
smallint	2 bytes	precisão fixa com faixa pequena	-32768 a +32767
integer	4 bytes	escolha usual para precisão fixa	-2147483648 a +2147483647
bigint	8 bytes	precisão fixa com faixa grande	-9223372036854775808 a 9223372036854775807
decimal	variável	precisão especificada pelo usuário, exato	sem limite
numeric	variável	precisão especificada pelo usuário, exato	sem limite
real	4 bytes	precisão variável, inexato	precisão de 6 dígitos decimais
double precision	8 bytes	precisão variável, inexato	precisão de 15 dígitos decimais
serial	4 bytes	inteiro autoincremental	1 a 2147483647
bigserial	8 bytes	inteiro autoincremental grande	1 a 9223372036854775807

Os tipos numéricos possuem um conjunto completo de operadores aritméticos e funções correspondentes. Consulte o Capítulo 6 para obter mais informação. As próximas seções descrevem os tipos em detalhe.

Os tipos inteiros

Os tipos smallint, integer e bigint armazenam números inteiros, ou seja, números sem a parte fracionária, com diferentes faixas de valor. A tentativa de armazenar um valor fora da faixa permitida ocasiona erro.

O tipo integer é a escolha usual, porque oferece o melhor equilíbrio entre faixa de valores, tamanho de armazenamento e desempenho. Geralmente o tipo smallint só é utilizado quando o espaço em disco está muito escasso. O tipo bigint somente deve ser usado quando a faixa de valores de integer não for suficiente, porque este último é bem mais rápido.

O tipo bigint pode não funcionar de modo correto em todas as plataformas, porque depende de suporte no compilador para inteiros de oito bytes. Nas máquinas sem este suporte, o bigint age do mesmo modo que o integer (mas ainda demanda oito bytes para seu armazenamento). Entretanto, não é de nosso conhecimento nenhuma plataforma razoável onde este caso ainda se aplique.

O padrão SQL somente especifica os tipos inteiros integer (ou int) e smallint. O tipo bigint, e os nomes de tipo int2, int4 e int8 são extensões, também compartilhadas por vários outros sistemas de banco de dados SQL.

Nota: Havendo uma coluna do tipo smallint ou bigint com um índice, podem ocorrer problemas ao tentar fazer o sistema utilizar este índice. Por exemplo, uma cláusula do tipo

... WHERE coluna_smallint = 42

não usará o índice, porque o sistema atribui o tipo integer à constante 42, e o PostgreSQL atualmente não pode utilizar um índice quando dois tipos de dado diferentes estão envolvidos. Um modo de contornar este problema é colocar a constante entre apóstrofos. Portanto,

... WHERE coluna_smallint = '42'

atrasa a resolução do tipo pelo sistema, fazendo com que seja atribuído o tipo correto para a constante.

Números de precisão arbitrária

O tipo numeric pode armazenar números com até 1.000 dígitos de precisão realizando cálculos exatos, sendo recomendado, especialmente, para armazenar quantias monetárias e outras quantidades onde a exatidão é requerida. Entretanto, o tipo numeric é muito lento quando comparado com os tipos de ponto flutuante descritos na próxima seção.

Os seguintes termos são utilizados : A escala do tipo numeric é o número de dígitos decimais da parte fracionária, à direita do ponto decimal. A precisão do tipo numeric é o número total de dígitos

significativos em todo o número, ou seja, o número de dígitos nos dois lados do ponto decimal. Portanto, o número 23.5141 [1] possui precisão igual a 6 e escala igual a 4. Os inteiros podem ser considerados como tendo escala igual a zero.

Tanto a precisão quanto a escala do tipo numérico podem ser configuradas. Para declarar a coluna como sendo do tipo numeric deve ser utilizada a sintaxe:

NUMERIC(precisão, escala)

A precisão deve ser um número positivo, enquanto a escala pode ser zero ou positiva. Como forma alternativa,

NUMERIC(precisão)

especifica escala igual a 0. Especificando-se

NUMERIC

sem qualquer precisão ou escala é criada uma coluna onde os valores numéricos com qualquer precisão ou escala podem ser armazenados, até a precisão limite da implementação. Uma coluna deste tipo não transforma os valores de entrada para nenhuma determinada escala, enquanto as colunas do tipo numeric com escala declarada transformam os valores da entrada para esta escala (O padrão SQL requer a escala padrão sendo igual a 0, ou seja, uma transformação para a precisão inteira. Consideramos isto sem utilidade. Havendo preocupação com a portabilidade, sempre deve ser especificada explicitamente a precisão e a escala).

Se a precisão ou a escala do valor for maior que a precisão ou a escala declarada para a coluna, o sistema tenta arredondar o valor. Se o valor não puder ser arredondado para satisfazer os limites declarados, ocasiona erro.

Os tipos decimal e numeric são equivalentes. Os dois tipos fazem parte do padrão SQL.

Tipos de ponto flutuante

Os tipos de dado real e double precision são tipos numéricos de precisão variável não exatos. Na prática, estes tipos são geralmente implementações do padrão IEEE 754 para aritmética binária de ponto flutuante de precisão simples e dupla, respectivamente, conforme suportado pelo processador, sistema operacional e compilador utilizados.

Não exato significa que alguns valores não podem ser convertidos exatamente para o formato interno, sendo armazenados como aproximações. Portanto, ao armazenar e imprimir um valor podem ocorrer pequenas discrepâncias. A gerência destes erros, e como se propagam através dos cálculos, é assunto de um ramo da ciência da computação e da matemática que não será discutido aqui, com exceção dos seguintes pontos:

Se o armazenamento e cálculos exatos (tal como em valores monetários) forem necessários, deve ser utilizado o tipo numeric em vez destes.

Se cálculos complicados forem efetuados com estes tipos para algo importante, especialmente se depender de certos comportamentos em situações limites (infinito, underflow), a implementação deve ser avaliada cuidadosamente.

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

Normalmente o tipo real possui uma faixa de pelo menos -1E+37 a +1E+37, com uma precisão de pelo menos 6 dígitos decimais. O tipo double precision normalmente possui uma faixa em torno de -1E+308 a +1E+308 com uma precisão de pelo menos 15 dígitos. Os valores muito pequenos ou muito grandes causam erro. O arredondamento pode ser efetuado se a precisão do número entrado for muito grande. Os números muito próximos de zero, que não podem ser representados de forma diferente de zero, causam erro de underflow.

Os tipos seriais

O tipo de dado serial não é na verdade um tipo, mas meramente uma notação conveniente para especificar colunas identificadoras (semelhante à propriedade AUTO_INCREMENT existente em alguns outros bancos de dados). Na implementação corrente especificar

```
CREATE TABLE nome_tabela (
    nome_coluna SERIAL
);
```

é o mesmo que especificar:

```
CREATE SEQUENCE nome_tabela_nome_coluna_seq;
CREATE TABLE nome_tabela (
    nome_coluna integer DEFAULT nextval('nome_tabela_nome_coluna_seq') NOT NULL
);
```

Conforme visto, foi criada uma coluna do tipo inteiro e feito o valor padrão ser atribuído a partir de um gerador de seqüência. A restrição NOT NULL foi aplicada para garantir que o valor nulo não pode ser inserido explicitamente. Na maioria das vezes, deve ser colocada uma restrição UNIQUE ou PRIMARY KEY para não permitir a inserção de valores duplicados por acidente, mas isto não é automático.

A utilização da coluna serial para inserir o próximo valor da seqüência na tabela é feita atribuindo o valor padrão da coluna serial. Pode ser feito omitindo a coluna da lista de colunas do comando INSERT, ou por meio da utilização da palavra chave DEFAULT.

Os nomes de tipo serial e serial4 são equivalentes: os dois criam uma coluna do tipo integer. Os nomes de tipo bigserial e serial8 funcionam do mesmo modo, exceto por criarem uma coluna do tipo bigint. O tipo bigserial deve ser utilizado se forem esperados mais que 231 identificadores durante a existência da tabela.

A seqüência criada pelo tipo serial é automaticamente excluída quando a coluna onde foi definida é excluída, e não pode ser excluída de outra forma (Isto não era verdade no PostgreSQL antes da

versão 7.3. Observe que esta ligação de exclusão automática não ocorre para uma seqüência criada pela restauração da cópia de segurança de um banco de dados pré-7.3; a cópia de segurança não contém as informações necessárias para estabelecer a ligação de dependência). Além disso, a dependência entre a seqüência e a coluna é feita apenas para a própria coluna serial; se qualquer outra coluna fizer referência à seqüência (talvez chamando manualmente a função nextval()), pode haver quebra se a seqüência for removida. A utilização de colunas serial como costume é considerado um estilo ruim.

Nota: Antes do PostgreSQL 7.3 serial implicava UNIQUE. Isto não é mais automático. Se for desejado que uma coluna serial também seja UNIQUE ou PRIMARY KEY, deve ser especificado da mesma forma como é feito para qualquer outro tipo.

Notas

[1] Padrão americano, ponto e não vírgula separando a parte fracionária. (N.T.)

Tipo monetário

Nota: O tipo money está obsoleto (deprecated). Deve ser utilizado o tipo numeric ou decimal em seu lugar, em combinação com a função to_char. O tipo monetário poderá vir a ser uma camada de regionalização sobre um tipo numeric em alguma versão futura.

O tipo money armazena a quantidade de dinheiro com uma representação de ponto decimal fixa; veja a Tabela 5-3. O formato de saída é específico para a região.

Várias formas de entrada são aceitas, incluindo literais inteiros e de ponto flutuante, assim como o formato monetário "típico" '$1,000.00'. A saída é nesta última forma.

Tabela - Tipos monetários

Nome do tipo	Armazenamento	Descrição	Faixa de valores
money	4 bytes	quantidade de dinheiro	-21474836.48 a +21474836.47

Tipos para caracteres

Tabela - Tipos para caracteres

Nome do tipo	Descrição
character varying(n), varchar(n)	comprimento variável com limite
character(n), char(n)	comprimento fixo, completado com brancos
text	comprimento variável não limitado

O SQL define dois tipos básicos para caracteres: character varying(n) e character(n), onde n é um número inteiro positivo. Estes dois tipos podem armazenar cadeias de caracteres com até n caracteres de comprimento. A tentativa de armazenar uma cadeia de caracteres mais longa em uma coluna de um destes tipos resulta em erro, a não ser que os caracteres excedentes sejam todos espaços. Neste caso a cadeia de caracteres será truncada em seu comprimento máximo (Esta

exceção um tanto bizarra é requerida pelo padrão SQL). Se a cadeia de caracteres a ser armazenada for mais curta que o comprimento declarado, os valores do tipo character serão completados com espaço; os valores do tipo character varying simplesmente vão armazenar uma cadeia de caracteres mais curta.

Nota: Se um valor for explicitamente transformado (cast) para character varying(n) ou para character(n), então um valor com comprimento excedente será truncado para n caracteres sem ocorrência de erro (isto também é requerido pelo padrão SQL).

Nota: Antes do PostgreSQL 7.2 as cadeias de caracteres muito longas eram sempre truncadas sem a ocorrência de erro, tanto no contexto de transformação explícita quanto no de implícita.

As notações varchar(n) e char(n) são aliases para character varying(n) e character(n), respectivamente. O character sem especificação de comprimento é equivalente a character(1); se character varying for utilizado sem especificação de comprimento, este tipo aceita cadeias de caracteres de qualquer tamanho. Este último é uma extensão do PostgreSQL.

Além desses, o PostgreSQL suporta o tipo mais geral text, que armazena cadeias de caracteres de qualquer comprimento. Diferentemente de character varying, text não requer um limite superior explicitamente declarado de seu tamanho. Embora o tipo text não esteja no padrão SQL, muitos outros RDBMS também o incluem.

São requeridos para armazenar os dados destes tipos 4 bytes mais a própria cadeia de caracteres e, no caso do character, mais os espaços para completar o tamanho. As cadeias de caracteres longas são comprimidas pelo sistema automaticamente, portanto o espaço físico requerido em disco pode ser menor. Os valores longos também são armazenados em tabelas de segundo plano não interferindo, portanto, com o acesso rápido aos valores mais curtos da coluna. De qualquer forma, o tamanho possível da maior cadeia de caracteres armazenada é em torno de 1 GB (O valor máximo permitido para n na declaração do tipo de dado é menor que isto. Não seria muito útil mudar, porque nas codificações de caractere multibyte o número de caracteres e de bytes é bem diferente de qualquer modo. Se for desejado armazenar cadeias de caracteres longas, sem um limite superior especificado, deve ser utilizado text ou character varying sem a especificação de comprimento, em vez de especificar um limite de comprimento arbitrário).

Dica: Não existe diferença de desempenho entre estes três tipos, a não ser pelo aumento do tamanho do armazenamento quando é utilizado o tipo completado com brancos.

Exemplo: Utilização dos tipos caractere

```
CREATE TABLE teste1 (a character(4));
INSERT INTO teste1 VALUES ('ok');
SELECT a, char_length(a) FROM teste1; -- (1)
  a   | char_length
------+-------------
 ok   |           4

CREATE TABLE teste2 (b varchar(5));
INSERT INTO teste2 VALUES ('ok');
```

```
INSERT INTO teste2 VALUES ('bom       ');
INSERT INTO teste2 VALUES ('muito longo');
ERROR:  value too long for type character varying(5)
INSERT INTO teste2 VALUES ('muito longo'::varchar(5)); -- truncamento explícito
SELECT b, char_length(b) FROM teste2;
   b   | char_length
-------+-------------
 ok    |           2
 bom   |           5
 muito |           5
(1)
```

Existem dois outros tipos para caracteres de comprimento fixo no PostgreSQL, mostrados na Tabela 5-5. O tipo name existe apenas para armazenamento dos nomes do catálogo interno, não havendo pretensão para ser de uso geral. Seu comprimento é atualmente definido como 64 bytes (63 caracteres utilizáveis mais o terminador) mas deve ser referenciado utilizando a constante NAMEDATALEN. O comprimento é definido em tempo de compilação (sendo, portanto, ajustável para usos especiais); o padrão para comprimento máximo poderá mudar em uma versão futura. O tipo "char" (observe as aspas) é diferente de char(1), porque utiliza apenas um byte para armazenamento. É utilizado internamente nos catálogos do sistema como o tipo de enumeração do homem pobre (poor-man's enumeration type).

Tabela - Tipos especiais para caracteres

Nome do tipo	Armazenamento	Descrição
"char"	1 byte	tipo interno de um único caractere
name	64 bytes	tipo interno de sessenta e três caracteres

Tipo para cadeias binárias

O tipo de dado bytea permite o armazenamento de cadeias binárias; veja a Tabela 5-6.

Tabela - Tipo para cadeias binárias

Nome do tipo	Armazenamento	Descrição
bytea	4 bytes mais a cadeia binária	Cadeia binária de comprimento variável (sem limite específico)

A cadeia binária é uma seqüência de octetos (ou bytes). As cadeias binárias se diferenciam das cadeias de caracteres por duas características: Em primeiro lugar as cadeias binárias permitem o armazenamento de octetos com o valor zero, além de outros octetos "não imprimíveis". Em segundo lugar, as operações nas cadeias binárias processam os bytes armazenados, enquanto o processo de codificação das cadeias de caracteres depende das configurações regionais.

Ao entrar com valores para bytea, octetos com certos valores devem possuir uma seqüência de escape (porém, todos os valores de octeto podem possuir uma seqüência de escape) quando utilizado como parte da cadeia literal na declaração SQL. Em geral, para construir a seqüência de escape de um octeto, este é convertido em um número octal de três dígitos equivalente ao valor

decimal do octeto, e precedido por duas contrabarras. Alguns valores de octeto possuem seqüências de escape alternativas.

Tabela - bytea Octetos literais com seqüência de escape

Valor decimal do octeto	Descrição	Entrada com seqüência de escape	Exemplo	Resultado impresso
0	octeto zero	'\\000'	SELECT '\\000'::bytea;	\000
39	apóstrofo	'\'' ou '\\047'	SELECT '\''::bytea;	'
92	contrabarra	'\\\\' ou '\\134'	SELECT '\\\\'::bytea;	\\

Observe que o resultado de cada um dos exemplos da Tabela 5-7 tem um octeto de comprimento, embora a representação de contrabarra do octeto zero possua mais de um caractere. A saída dos octetos de bytea também possui seqüência de escape. De modo geral, cada octeto com valor decimal "não imprimível" é convertido em um número octal equivalente de três dígitos, e precedido por uma contrabarra. A maioria dos octetos "imprimíveis" é representada por sua representação padrão no conjunto de caracteres do cliente. O octeto com valor decimal 92 (contrabarra) possui uma representação alternativa especial.

Tabela - bytea Saída dos octetos com escape

Valor decimal do octeto	Descrição	Representação com seqüência de escape da saída	Exemplo	Resultado impresso
92	contrabarra	\\	SELECT '\\134'::bytea;	\\
0 a 31 e 127 a 255	octetos "não imprimíveis"	\### (valor octal)	SELECT '\\001'::bytea;	\001
32 a 126	octetos "imprimíveis"	representação ASCII	SELECT '\\176'::bytea;	~

Para utilizar a notação de octeto com seqüência de escape em bytea, os literais (cadeias de caracteres de entrada) devem conter duas contrabarras, porque passam por dois analisadores no servidor PostgreSQL. A primeira contrabarra é interpretada como caractere de escape pelo analisador de cadeia de caracteres, sendo por este consumida, deixando os caracteres seguintes. A contrabarra restante é reconhecida pela função de entrada de bytea como sendo o prefixo de um valor octal de três dígitos. Por exemplo, a cadeia de caracteres passada para o servidor como '\\001' se torna '\001' após passar pelo analisador de cadeias de caracteres. O '\001' é então enviado para a função de entrada de bytea, onde é convertido em um único octeto com valor decimal igual a 1.

Pela mesma razão, uma contrabarra deve ser entrada como '\\\\' (ou '\\134'). A primeira e a terceira contrabarras são interpretadas como caractere de escape pelo analisado de cadeia de caracteres e, portanto, são consumidas, deixando duas contrabarras na cadeia de caracteres passada para a função de entrada de bytea, que as interpreta como sendo uma única contrabarra. Por exemplo, a cadeia de caracteres passada para o servidor como '\\\\' se torna '\\' após passar pelo analisador de cadeias de caracteres. A '\\' é então enviada para a função de entrada de bytea, onde é convertida em um único octeto com valor decimal igual a 92.

O apóstrofo é um pouco diferente porque precisa ser entrado como '\'' (ou '\\047'), e não como '\\'.

Isto acontece porque enquanto o analisador de literais interpreta o apóstrofo como caractere especial, removendo a única contrabarra, a função de entrada de bytea não reconhece o apóstrofo como sendo um octeto especial. Portanto, uma cadeia de caracteres passada para o servidor como '\'' se torna ''' após passar pelo analisador de cadeias de caracteres. O ''' é enviado em seguida para a função de entrada de bytea onde, então, é retido o valor decimal igual a 39 do seu único octeto.

Dependendo do programa cliente do PostgreSQL utilizado, pode haver trabalho adicional a ser realizado em relação aos escapes das cadeias de caracteres bytea. Por exemplo, pode ser necessário colocar escapes para os caracteres de nova-linha e retorno-de-carro se a interface realizar a tradução automática destes caracteres. Do mesmo modo pode ser necessário duplicar as contrabarras se o analisador utilizado também trata a contrabarra como caractere de escape.

O padrão SQL define um tipo diferente de cadeia de caracteres binária, chamada BLOB ou BINARY LARGE OBJECT (objeto binário grande). O formato de entrada é diferente se comparado com bytea, mas as funções e operadores fornecidos são praticamente os mesmos.

Tipos para data e hora

O PostgreSQL suporta o conjunto completo de tipos de data e hora do SQL, mostrados na abaixo.

Tabela - Tipos para data e hora

Tipo	Descrição	Armazenamento	Mais cedo	Mais tarde	Resolução
timestamp [(p)] [without time zone]	tanto data quanto hora	8 bytes	4713 AC	1465001 DC	1 microssegundo / 14 dígitos
timestamp [(p)] with time zone	tanto data quanto hora	8 bytes	4713 BC	AD 1465001	1 microssegundo / 14 dígitos
interval [(p)]	intervalos de tempo	12 bytes	-178000000 anos	178000000 anos	1 microssegundo
date	somente datas	4 bytes	4713 AC	32767 DC	1 dia
time [(p)] [without time zone]	somente a hora do dia	8 bytes	00:00:00.00	23:59:59.99	1 microssegundo
time [(p)] with time zone	somente a hora do dia	12 bytes	00:00:00.00+12	23:59:59.99-12	1 microssegundo

time, timestamp, e interval aceitam um valor opcional de precisão p, que especifica o número de dígitos fracionários presentes no campo de segundos. Por padrão não existe limite explícito para a precisão. O intervalo permitido para p é de 0 a 6 para os tipos timestamp e interval.

Nota: Quando os valores de timestamp são armazenados como números de ponto flutuante de precisão dupla (atualmente o padrão), o limite efetivo da precisão pode ser inferior a 6, porque os valores de timestamp são armazenados como segundos decorridos desde 2000-01-01. A precisão de microssegundos é obtida para datas próximas a 2000-01-01 (alguns anos), mas a precisão degrada para datas mais afastadas. Quando os timestamps são armazenadas como inteiros de oito bytes (uma opção em tempo de compilação), a precisão de microssegundo está disponível para todo o intervalo de valores.

Para os tipos time, o intervalo permitido para p é de 0 a 6 quando armazenados em inteiros de oito bytes, ou de 0 a 10 quando armazenados em ponto flutuante.

Zonas horárias e convenções de zonas horárias são influenciadas por decisões políticas, e não apenas pela geometria da terra. As zonas horárias em torno do mundo ficaram bastante padronizadas durante o século XX, mas continuam suscetíveis a mudanças arbitrárias. O PostgreSQL utiliza as funcionalidades presentes no sistema operacional para fornecer apoio à saída de zona horária, sendo que os sistemas geralmente contêm informações apenas para o intervalo de tempo entre 1902 e 2038 (correspondendo ao período completo de um sistema Unix convencional). timestamp with time zone e time with time zone usam a informação de zona horária somente dentro deste intervalo de tempo, pressupondo que as horas fora deste intervalo estão em UTC.

O tipo time with time zone é definido pelo padrão SQL, mas a definição apresenta propriedades que levam a utilizações duvidosas. Na maioria dos casos, a combinação de date, time, timestamp without time zone e timestamp with time zone deve fornecer uma faixa completa das funcionalidades de data e hora necessária para qualquer aplicação.

Os tipos abstime e reltime são tipos de menor precisão usados internamente apenas. É desencorajada a utilização destes tipos em novas aplicações, além de ser encorajada a migração das aplicações antigas quando for conveniente. Qualquer um, ou mesmo todos estes tipos internos, podem desaparecer nas próximas versões.

Entrada de data e hora

A entrada da data e da hora é aceita em praticamente todos os formatos razoáveis, incluindo o ISO 8601, o SQL-compatível, o PostgreSQL tradicional, além de outros. Para alguns formatos, a ordem do dia e do mês da entrada da data pode ser ambíguo e, por isso, existe apoio para especificar a ordem esperada destes campos. O comando SET DateStyle TO 'US' ou SET DateStyle TO 'NonEuropean' especifica a variante "mês antes do dia", o comando SET DateStyle TO 'European' especifica a variante "dia antes do mês".

O PostgreSQL é mais flexível no tratamento de data e hora do que o requerido pelo padrão SQL. Consulte o Apêndice A para conhecer as regras exatas de análise da entrada de data e hora e para os campos texto reconhecidos, incluindo meses, dias da semana e zonas horárias.

Lembre-se que qualquer entrada de data ou de hora literal necessita estar entre apóstrofos, como os textos das cadeias de caracteres.

type [(p)] 'valor'

onde p na especificação opcional da precisão é um número inteiro correspondente ao número de dígitos fracionários do campo de segundos. A precisão pode ser especificada para os tipos para time, timestamp e interval.

Datas

Tabela - Entrada de data

Exemplo	Descrição
January 8, 1999	não-ambíguo
1999-01-08	formato ISO-8601, o preferido
01/08/1999	Americano; lê primeiro de agosto no modo EuropeuEuropeu; lê primeiro de agosto no modo Americano
08/01/1999	
1/18/1999	Americano; lê 18 de janeiro em qualquer modo
19990108	ISO-8601 ano, mês e dia
990108	ISO-8601 ano, mês e dia
'1999.008	ano e dia do ano
99008	ano e dia do ano
J2451187	data juliana
January 8, 99 BC	ano de 99 Antes de Cristo

Horas

O tipo time pode ser especificado como time ou como time without time zone. A precisão opcional p deve estar entre 0 e 6, tomando como padrão a precisão do literal de entrada da hora.

Tabela - Entrada de hora

Exemplo	Descrição
04:05:06.789	ISO 8601
04:05:06	ISO 8601
04:05:00	ISO 8601
40506	ISO 8601
04:05:00	o mesmo que 04:05; AM não afeta o valor
16:05:00	o mesmo que 16:05; a hora entrada deve ser <= 12
allballs	o mesmo que 00:00:00

O tipo time with time zone aceita todas as entradas incluindo a legal para o tipo time, acrescida da zona horária legal.

Tabela - Entrada de hora com zona horária

Exemplo	Descrição
04:05:06.789-8	ISO 8601
04:05:06-08:00	ISO 8601

Exemplo	Descrição
04:05-08:00	ISO 8601
040506-08	ISO 8601

Registro de data

Os tipos registro de data são timestamp [(p)] without time zone e timestamp [(p)] with time zone. Escrever apenas timestamp é equivalente a escrever timestamp without time zone.

Nota: Antes do PostgreSQL 7.3 escrever apenas timestamp era equivalente a escrever timestamp with time zone. Foi mudado para ficar conforme o padrão SQL.

As entradas válidas para registro de data consistem na concatenação da data com a hora, seguida opcionalmente por AD ou BC, seguida por uma zona horária opcional. Portanto

1999-01-08 04:05:06

e

1999-01-08 04:05:06 -8:00

são valores válidos, que seguem o padrão ISO 8601. Além desses, o formato muito utilizado

January 8 04:05:06 1999 PST

é suportado.

A precisão opcional p deve estar entre 0 e 6, tomando como padrão a precisão do literal de entrada do tipo timestamp.

Para timestamp without time zone, uma zona horária explícita especificada na entrada é ignorada silenciosamente, ou seja, o valor resultante do valor da data/hora é derivada dos campos de data/hora explicitados no valor da entrada, sem ser ajustado para a zona horária.

Para timestamp with time zone, o valor armazenado internamente está sempre em UTC (GMT). O valor de entrada possuindo zona horária especificada explicitamente é convertido em UTC, utilizando o deslocamento adequado para a zona horária. Se nenhuma zona horária for especificada na cadeia de caracteres da entrada, pressupõe-se que está na mesma zona horária indicada pelo parâmetro TimeZone do sistema, sendo convertida em UTC utilizando o deslocamento da zona TimeZone.

Quando um valor de timestamp with time zone é enviado para a saída é sempre convertido de UTC para o a zona corrente de TimeZone, e exibido como hora local desta zona. Para ver a hora em outra zona horária, ou se muda a TimeZone ou se usa a construção AT TIME ZONE.

As conversões entre timestamp without time zone e timestamp with time zone normalmente pressupõem que o valor de timestamp without time zone devem ser tomados ou fornecidos como

hora local da TimeZone. A referência para uma zona horária diferente pode ser especificada para a conversão utilizando AT TIME ZONE.

Tabela - Entrada de zona horária

Zona horária	Descrição
PST	Pacific Standard Time
-08:00:00	deslocamento ISO-8601 para PST
-800	deslocamento ISO-8601 para PST
-8	deslocamento ISO-8601 para PST

Intervalos

Os valores para o tipo interval podem ser escritos utilizando uma das seguintes sintaxes:

Quantidade Unidade [Quantidade Unidade...] [Direção]
@ Quantidade Unidade [Quantidade Unidade...] [Direção]

onde: Quantidade é o número (possivelmente com sinal), Unidade é second, minute, hour, day, week, month, year, decade, century, millennium, ou abreviaturas ou plurais destas unidades; A Direção pode ser ago (atrás) ou vazio. O sinal de arroba (@) é uma notação opcional. As quantidades com unidades diferentes são implicitamente adicionadas na conta com o sinal adequado.

As quantidades de dias, horas, minutos e segundos podem ser especificadas sem se informar explicitamente as unidades. Por exemplo, '1 12:59:10' é lido do mesmo modo que '1 day 12 hours 59 min 10 sec'.

A precisão opcional p deve estar entre 0 e 6, tomando como padrão a precisão do literal da entrada.

Valores especiais

As seguintes funções, compatíveis com o padrão SQL podem ser utilizadas como valores de data e hora para o tipo de dado correspondente: CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP. A última aceita uma especificação opcional de precisão.

O PostgreSQL também suporta diversos valores especiais para entrada de data e hora por conveniência. Os valores infinity e -infinity são representados de forma especial dentro do sistema, sendo mostrados da mesma forma; porém, os demais são apenas notações abreviadas convertidas para valores comuns de data e hora quando lidos.

Tabela - Entradas especiais de data e hora

Cadeia de caracteres entrada	Descrição
epoch	1970-01-01 00:00:00+00 (zero horas do sistema Unix)
infinity	depois de todos os outros registros de data e hora (não disponível para o tipo date)

Cadeia de caracteres entrada	Descrição
-infinity	antes de todos os outros registros de data e hora (não disponível para o tipo date)
now	hora da transação corrente
today	meia-noite de hoje
tomorrow	meia-noite de amanhã
yesterday	meia-noite de ontem
zulu, allballs, z	00:00:00.00 GMT

Saídas de data e hora

O formato da saída pode ser definido como um dos quatro estilos entre ISO 8601, SQL (Ingres), PostgreSQL tradicional e German, utilizando SET DateStyle. O padrão é o formato ISO (o padrão SQL requer a utilização do formato ISO 8601; o nome do formato de saída "SQL" é um acidente histórico). A tabela abaixo mostra um exemplo de cada um dos estilos de saída. A saída dos tipos date e time obviamente utilizam apenas a parte da data ou da hora em conformidade com os exemplos fornecidos.

Tabela - Estilos de data e hora

Especificação de estilo	Descrição	Exemplo
ISO	ISO 8601/padrão SQL	1997-12-17 07:37:16-08
SQL	estilo tradicional	12/17/1997 07:37:16.00 PST
PostgreSQL	estilo original	Wed Dec 17 07:37:16 1997 PST
German	estilo regional	17.12.1997 07:37:16.00 PST

O estilo SQL possui as variantes Européia e não Européia (U.S.), as quais determinam se o mês vem antes ou depois do dia para ver como esta definição também afeta a interpretação dos valores entrados).

Tabela - Convenções de ordem na data

Especificação de estilo	Descrição	Exemplo
European	dia/mês/ano	17/12/1997 15:37:16.00 MET
US	mês/dia/ano	12/17/1997 07:37:16.00 PST

A saída do tipo interval se parece com o formato da entrada, exceto que as unidades como week ou century são convertidas em dias. No modo ISO a saída se parece com

[Quantidade Unidade [...]] [Dias] Horas:Minutos [ago]

Os estilos de data e hora podem ser selecionados pelo usuário utilizando o comando SET DATESTYLE, o parâmetro datestyle no arquivo de configuração postgresql.conf, e a variável do ambiente PGDATESTYLE no servidor ou no cliente. A função de formatação to_char também pode ser utilizada como forma mais flexível de formatar a saída de data e hora.

Zonas horárias

O PostgreSQL se esforça em ser compatível com as definições do padrão SQL para a utilização típica. Entretanto, o padrão SQL possui uma mistura de funcionalidades peculiar para tipos de data e hora. Dois problemas são:

Embora o tipo date não possua zona horária associada, o tipo time pode possuir. As zonas horárias do mundo real não possuem nenhum significado, a não ser quando associadas a uma data e hora, porque o deslocamento pode variar durante o ano devido ao horário de verão.

A zona horária padrão é especificada por uma constante inteira contendo o deslocamento em relação à GMT/UTC. Não é possível fazer ajuste devido ao horário de verão (DST) ao se realizar aritmética de data e hora entre fronteiras do horário de verão (DST). [1]

Para superar estas dificuldades, recomenda-se utilizar tipos de data e hora contendo tanto a data quanto a hora quando utilizar zonas horárias. Recomenda-se a não utilização do tipo time with time zone (embora seja suportado pelo PostgreSQL para aplicações legadas e para compatibilidade com outras implementações do SQL). O PostgreSQL pressupõe a zona horária local para qualquer tipo contendo apenas a data ou a hora. Além disso, o suporte da zona horária é derivado da capacidade do sistema operacional subjacente e, portanto, pode tratar horário de verão (DST) e outros comportamentos esperados.

O PostgreSQL obtém o suporte de zona horária do sistema operacional subjacente para datas entre 1902 e 2038 (próximo dos limites típicos de data dos sistemas da família Unix). Fora deste intervalo, todas as datas são pressupostas como sendo especificadas e utilizadas no Universal Coordinated Time (UTC/Tempo Coordenado Universal).

Todas as datas e horas são armazenadas internamente em UTC, tradicionalmente conhecido como Greenwich Mean Time (GMT/Hora do meridiano de Greenwich). As horas são convertidas para a hora local no servidor de banco de dados antes do envio para a aplicação cliente estando, portanto, na zona horária do servidor por padrão.

Existem vários modos de selecionar a zona horária utilizada pelo servidor:

A variável de ambiente TZ no hospedeiro do servidor é utilizada pelo servidor como a zona horária padrão, se nenhuma outra for especificada.

O parâmetro de configuração timezone pode ser definido no arquivo postgresql.conf.

A variável de ambiente PGTZ, quando definida no cliente, é utilizada pelas aplicações que utilizam a libpq para enviar o comando SET TIME ZONE para o servidor durante a conexão.

O comando SQL SET TIME ZONE define a zona horária para a sessão.

Nota: Se uma zona horária inválida for especificada, então a zona horária passa a ser a UTC (pelo menos na maioria dos sistemas).

Consulte o Apêndice A para obter a lista das zonas horárias disponíveis.

Internals

O PostgreSQL utiliza datas Julianas para todos os cálculos de data e hora, porque possuem a boa propriedade de predizer/calcular corretamente qualquer data mais recente que 4713 AC até bem distante no futuro, partindo da premissa que o ano possui 365,2425 dias.

As convenções de data anteriores ao século 19 são uma leitura interessante, mas não são suficientemente consistentes para permitir a codificação em rotinas tratadoras de data e hora.

Notas

[1] DST significa "Daylight Saving Time", também conhecido como Horário de Verão. O DST é utilizado em muitos países como ajuste dos relógios locais, para obter mais vantagens com a luz natural existente durante os meses de verão. (N.T.)

Tipo booleano

O PostgreSQL disponibiliza o tipo boolean padrão do SQL. O tipo boolean pode possuir apenas dois estados: "verdade" ou "falso". O terceiro estado, "desconhecido", é representado pelo valor nulo do SQL.

Os valores literais válidos para o estado "verdade" são:

TRUE 't' 'true' 'y' 'yes' '1'

Para o estado "falso" os seguintes valores podem ser utilizados:

FALSE 'f' 'false' 'n' 'no' '0'

A utilização das palavras chave TRUE e FALSE é preferida (e em conformidade com o padrão SQL).

Exemplo - Utilização do tipo boolean

```
CREATE TABLE teste1 (a boolean, b text);
INSERT INTO teste1 VALUES (TRUE, 'sic est');
INSERT INTO teste1 VALUES (FALSE, 'non est');
SELECT * FROM teste1;
 a |    b
---+---------
 t | sic est
 f | non est

SELECT * FROM teste1 WHERE a;
 a |    b
---+---------
```

t | sic est

Dica: Os valores do tipo boolean não podem ser transformados diretamente em outros tipos (por exemplo, CAST (valor_booleano AS integer) não funciona). Isto pode ser obtido utilizando a expressão CASE: CASE WHEN valor_booleano THEN 'valor se for verdade' ELSE 'valor se for falso' END.

O tipo boolean utiliza 1 byte para seu armazenamento.

Tipos geométricos

Os tipos de dado geométricos representam objetos no espaço bidimensional. A Tabela 5-17 mostra os tipos geométricos disponíveis no PostgreSQL. O tipo mais fundamental, o ponto, forma a base de todos os outros tipos.

Tabela - Tipos geométricos

Tipos geométricos	Armazenamento	Representação	Descrição
point	16 bytes	(x,y)	Ponto no espaço
line	32 bytes	((x1,y1),(x2,y2))	Linha infinita (não implementado completamente)
lseg	32 bytes	((x1,y1),(x2,y2))	Segmento finito de linha
box	32 bytes	((x1,y1),(x2,y2))	caixa retangular
path	16+16n bytes	((x1,y1),...)	Caminho fechado (similar ao polígono)
path	16+16n bytes	[(x1,y1),...]	Caminho aberto
polygon	40+16n bytes	((x1,y1),...)	Polígono (similar ao caminho fechado)
circle	24 bytes	<(x,y),r>	Círculo (centro e raio)

Um amplo conjunto de funções e operadores está disponível para realizar várias operações geométricas, como escala, translação, rotação e a determinação de interseções.

Ponto

Pontos são os blocos fundamentais para a construção dos tipos geométricos bidimensionais. O tipo point é especificado utilizando a seguinte sintaxe:

(x , y)
  x , y

onde os argumentos são

x
a coordenada no eixo x como número de ponto flutuante
y
 a coordenada no eixo y como número de ponto flutuante

Segmento de linha

Segmentos de linha (lseg) são representados por pares de pontos. O tipo lseg é especificado utilizando a seguinte sintaxe:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

onde os argumentos são

(x1,y1)
(x2,y2)

os pontos das extremidades do segmento de linha

Caixa

As caixas são representadas por pares de pontos de vértices opostos da caixa. O tipo box é especificado utilizando a seguinte sintaxe:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

onde os argumentos são

(x1,y1)
(x2,y2)

vértices opostos da caixa

A saída das caixas utiliza a primeira sintaxe. Os vértices são reordenados na entrada para armazenar o vértice direito superior, e depois o vértice esquerdo inferior. Os outros vértices da caixa podem ser fornecidos, mas o esquerdo inferior e o direito superior são determinados a partir da entrada e armazenados.

Caminho

Os caminhos são representados por conjuntos de pontos conectados. Os caminhos podem ser abertos, onde o primeiro e o último ponto do conjunto não estão conectados, e fechados, onde o primeiro e o último ponto estão conectados. As funções popen(p) e pclose(p) são fornecidas para forçar o caminho ser aberto ou fechado, e as funções isopen(p) e isclosed(p) são fornecidas para testar estes tipos na consulta.

O tipo path é especificado utilizando uma das seguintes sintaxes:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
```

[(x1 , y1) , ... , (xn , yn)]
(x1 , y1) , ... , (xn , yn)
(x1 , y1 , ... , xn , yn)
x1 , y1 , ... , xn , yn

onde os argumentos são

(x,y)

Pontos das extremidades dos segmentos de linha que formam o caminho. Um colchete no início ([) indica um caminho aberto, enquanto um parêntese no início (() indica um caminho fechado.

Os caminhos são exibidos utilizando a primeira sintaxe.

Polígono

Os polígonos são representados por um conjunto de pontos. Os polígonos deveriam provavelmente ser considerados equivalentes aos caminhos fechados, mas são armazenados de forma diferente e possuem um conjunto próprio de rotinas de apoio.

O tipo polygon é especificado utilizando uma das seguintes sintaxes:

((x1 , y1) , ... , (xn , yn))
(x1 , y1) , ... , (xn , yn)
(x1 , y1 , ... , xn , yn)
x1 , y1 , ... , xn , yn

onde os argumentos são

(x,y)

Os pontos das extremidades dos segmentos de linha formam a fronteira do polígono

Os polígonos são exibidos utilizando a primeira sintaxe.

Círculo

Os círculos são representados por um ponto central e um raio. O tipo circle é especificado utilizando uma das seguintes sintaxes:

< (x , y) , r >
((x , y) , r)
(x , y) , r
x , y , r

onde os argumentos são

(x,y)

centro do círculo
r
raio do círculo

Os círculos são mostrados utilizando a primeira sintaxe

Tipos de dado para endereço de rede

O PostgreSQL disponibiliza tipos de dado para armazenar endereços de IP e de MAC. É preferível utilizar estes tipos em vez dos tipos de texto puro, porque estes tipos possuem verificação de erro na entrada, além de vários operadores e funções especializadas.

Tabela - Tipos de dado para endereço de rede

Nome	Armazenamento	Descrição	Intervalo
cidr	12 bytes	redes IP	redes válidas IPv4
inet	12 bytes	hospedeiros e redes IP	hospedeiros e redes válidas IPv4
macaddr	6 bytes	endereço MAC	formatos personalizados

O IPv6 ainda não é suportado.

inet

O tipo de dado inet armazena um endereço de hospedeiro IP e, opcionalmente, a identificação da sub-rede onde se encontra, tudo em um único campo. A identificação da sub-rede é representada pelo número de bits na parte de rede do endereço (a "máscara de rede"). Se a máscara de rede for 32, então o valor não indica uma sub-rede, somente um único hospedeiro. Quando for desejado aceitar apenas redes, deve ser utilizado o tipo cidr em vez do tipo inet.

O formato de entrada para este tipo é x.x.x.x/y onde x.x.x.x é o endereço de IP e y é o número de bits da máscara de rede. Se a parte /y for deixada de fora, então a máscara de rede é 32, e o valor representa somente um único hospedeiro. Ao mostrar, o /y é deixado de fora se a máscara de rede for 32.

cidr

O tipo cidr armazena uma especificação de rede IP. Os formatos de entrada e de saída seguem as convenções do Classless Internet Domain Routing [10]. O formato para especificar redes sem classe é x.x.x.x/y onde x.x.x.x é a rede e y é o número de bits na máscara de rede. Se y for omitido será calculado pressupondo o sistema antigo de numeração com classes, exceto que será pelo menos suficientemente grande para incluir todos os octetos escritos na entrada.

Tabela - cidr Exemplos de entrada para este tipo

10 O CIDR, definido pela RFC1519, elimina o sistema de classes que originalmente determinou a parte de rede de um endereço IP. Como a sub-rede, da qual é uma extensão direta, ele conta com uma máscara de rede explícita para definir o limite entre as partes de rede e de host de um endereço. Manual de Administração do Sistema Unix - Evi Nemeth e outros - Bookman. (N.T.)

CIDR Entrada	CIDR Mostrado	abbrev(CIDR)
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24
128.1	128.1.0.0/16	128.1/16
128	128.0.0.0/16	128.0/16
128.1.2	128.1.2.0/24	128.1.2/24
10.1.2	10.1.2.0/24	10.1.2/24
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8

inet vs cidr

A diferença essencial entre os tipos de dado inet e cidr é que inet aceita valores com bits diferentes de zero à direita da máscara de rede, enquanto cidr não.

Dica: Caso não goste do formato de saída para os valores de inet ou cidr, tente utilizar as funções host(), text() e abbrev().

macaddr

O tipo macaddr armazena endereços de MAC, ou seja, endereços de hardware da placa Ethernet (embora os endereços MAC sejam utilizados para outras finalidades também). A entrada é aceita em vários formatos personalizados incluindo

'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'08-00-2b-01-02-03'
'08:00:2b:01:02:03'

sendo que todos especificam o mesmo endereço. Letras maiúsculas e minúsculas são aceitas para os dígitos de a a f. A saída é sempre na última forma mostrada.

O diretório contrib/mac na distribuição do fonte do PostgreSQL contém ferramentas que podem ser utilizadas para mapear endereços MAC em nomes de fabricantes de hardware.

Tipos para cadeias de bits

Cadeias de bits são cadeias contendo apenas dígitos 1 e 0. Podem ser armazenadas ou visualizar máscaras de bits. Existem dois tipos de dado SQL para bits: BIT(n) e BIT VARYING(n), onde n é um número inteiro positivo.

O tipo de dado BIT deve corresponder ao tamanho n exatamente; ocasiona erro tentar armazenar uma cadeia de bits mais curta ou mais longa. O tipo de dado BIT VARYING possui um

comprimento variável até o máximo de n; cadeias mais longas são rejeitadas. Escrever BIT sem o comprimento é equivalente a escrever BIT(1), enquanto BIT VARYING sem a especificação do comprimento significa comprimento ilimitado.

Nota: Se for feita uma transformação explícita do valor de uma cadeia de bits para BIT(n), os bits serão truncados ou preenchidos à direita com zeros para ficar exatamente com n bits, sem ocasionar erro. De forma semelhante, se for feita uma transformação explícita do valor de uma cadeia de bits para BIT VARYING(n), os bits serão truncados à direita caso existam mais de n bits.

Nota: Antes do PostgreSQL 7.2, os dados do tipo BIT eram sempre truncados ou completados à direita com zeros, com ou sem uma transformação explícita. Este comportamento foi modificado para ficar de acordo com o padrão SQL.

Exemplo. Utilização dos tipos cadeia de bits

```
CREATE TABLE teste (a BIT(3), b BIT VARYING(5));
INSERT INTO teste VALUES (B'101', B'00');
INSERT INTO teste VALUES (B'10', B'101');
ERROR:  Bit string length 2 does not match type BIT(3)
INSERT INTO teste VALUES (B'10'::bit(3), B'101');
SELECT * FROM teste;
  a  |  b
-----+-----
 101 | 00
 100 | 101
```

Tipos identificadores de objetos

Os identificadores de objeto (OIDs) são utilizados internamente pelo PostgreSQL como chaves primárias para várias tabelas do sistema. Além disso, uma coluna do sistema OID é adicionada às tabelas criadas pelo usuário (a não ser que WITHOUT OIDS seja especificado na criação da tabela). O tipo oid representa um identificador de objeto. Também existem diversos aliases para oid: regproc, regprocedure, regoper, regoperator, regclass, e regtype.

O tipo oid é implementado atualmente como inteiro de quatro bytes sem sinal. Portanto, não é grande o suficiente para fornecer unicidade para todo o banco de dados em bancos de dados grandes, ou mesmo em tabelas individuais grandes. Por isso, a utilização de uma coluna OID criada pelo usuário como chave primária é desencorajada. Os OIDs são melhor usados somente para referências às tabelas do sistema.

O tipo oid possui poucas operações próprias além da comparação (a qual é implementada utilizando a comparação sem sinal). Pode, entretanto, ser transformado em inteiro e, então, manipulado utilizando os operadores padrão para inteiros (Tome cuidado com as possíveis confusões entre inteiros com sinal e sem sinal se isto for feito).

Os tipos aliases de oid não possuem operações próprias com exceção de rotinas de entrada e saída especializadas. Estas rotinas são capazes de aceitar e mostrar nomes simbólicos para objetos do sistema, em vez do valor numérico puro e simples que o tipo oid usaria. Os tipos aliases permitem

uma visão simplificada dos valores de OID para os objetos: por exemplo, pode ser escrito 'minha_tabela'::regclass para obter o OID da tabela minha_tabela, em vez de SELECT oid FROM pg_class WHERE relname = 'minha_tabela' (Na verdade, um comando SELECT muito mais complicado é necessário para obter o OID correto quando existem diversas tabelas com o nome minha_tabela em esquemas diferentes).

Tabela - Tipos identificadores de objetos

Nome do tipo	Referencia	Descrição	Exemplo de valor
oid	qualquer um	identificador numérico de objeto	564182
regproc	pg_proc	nome de função	sum
regprocedure	pg_proc	função com tipos dos argumentos	sum(int4)
regoper	pg_operator	nome de operador	+
regoperator	pg_operator	operador com tipos dos argumentos	*(integer,integer) ou -(NONE,integer)
regclass	pg_class	nome da relação	pg_type
regtype	pg_type	nome de tipo	integer

Todos os tipos aliases de OID aceitam nomes qualificados pelo esquema, e mostram nomes qualificados pelo esquema na saída se o objeto não for encontrado no caminho de procura corrente sem que esteja qualificado. Os tipos aliases regproc e regoper somente aceitam a entrada de nomes únicos (não sobrecarregados), sendo portanto de uso limitado; para a maioria dos casos regprocedure e regoperator são mais apropriados. Em regoperator, os operadores unários são identificados escrevendo NONE no lugar do operando não utilizado.

Os OIDs são quantidades de 32 bits atribuídas a partir de um único contador para todo o agrupamento de bancos de dados. Em um banco de dados grande, ou de longa duração, é possível o contador reiniciar. Portanto, não é boa prática supor que os OID são únicos, a não ser que sejam tomadas medidas para garantir que sejam realmente únicos. Uma prática recomendada na utilização de OIDs como identificador de linhas é a criação de uma restrição de unicidade na coluna OID de cada tabela onde OID for utilizado. Nunca deve ser suposto que os OID são únicos entre tabelas; utilize a combinação de tableoid com OID da linha se for necessário criar um identificador único para todo o banco de dados (Nas versões futuras do PostgreSQL é possível que venha a ser utilizado um contador OID separado para cada tabela, fazendo com que tableoid deva ser incluído para obter um identificador único global).

Um outro tipo identificador utilizado pelo sistema é xid, ou identificador de transação (abreviada como xact). Este é o tipo de dado das colunas do sistema xmin e xmax. Os identificadores de transação são quantidades de 32 bits. Em um banco de dados de longa duração é possível que os identificadores de transação reiniciem. Isto não é um problema fatal devido aos procedimentos apropriados de manutenção; consulte o Guia do Administrador do PostgreSQL para ver os detalhes. Entretanto, não é sensato depender da unicidade do identificador de transação por muito tempo (mais de um bilhão de transações).

Um terceiro tipo de identificador utilizado pelo sistema é o cid, ou identificador de comando. Este é o tipo de dado das colunas do sistema cmin e cmax. Os identificadores de comando também são quantidades de 32 bits. Isto cria um limite de 232 (4 bilhões) de comandos SQL dentro de uma

única transação. Na prática este limite não é um problema --- observe que este número é o limite de comandos SQL e não o número de tuplas processadas.

O último tipo de identificador utilizado pelo sistema é o tid, ou identificador de tupla. Este é o tipo de dado da coluna do sistema ctid. O identificador de tupla é o par (número do bloco, índice da tupla no bloco) que identifica a localização física da tupla dentro da tabela.

Pseudotipos

O sistema de tipos do PostgreSQL contém uma série de entradas com finalidades especiais chamadas coletivamente de pseudotipos. Um pseudotipo não pode ser utilizado como o tipo de dado de uma coluna, mas pode ser utilizado para declarar tipos de argumentos e de resultados de funções. Cada um dos pseudotipos disponíveis é útil em situações nas quais o comportamento da função não corresponde a simplesmente aceitar ou retornar o valor de um tipo de dado específico do SQL.

Tabela - Pseudotipos

Nome do tipo	Descrição
record	Identifica uma função que retorna um tipo de linha não especificado
any	Indica que a função recebe qualquer tipo de dado entrado
anyarray	Indica que a função recebe qualquer tipo de dado array
void	Indica que a função não retorna valor
trigger	Uma função de gatilho é declarada como retornando o tipo trigger
language_handler	Um tratador de chamada de linguagem procedural é declarado como retornando o tipo language_handler
cstring	Indica que a função recebe ou retorna uma cadeia de caracteres C terminada por nulo
internal	Indica que a função recebe ou retorna um tipo de dado interno do servidor
opaque	Um nome de tipo obsoleto usado anteriormente para todas as finalidades acima

As funções codificadas em C (tanto nativas quanto carregadas dinamicamente) podem ser declaradas como recebendo ou retornando qualquer um destes pseudotipos. É responsabilidade do autor da função garantir que a função se comporta com segurança quando um pseudotipo é utilizado como tipo de argumento.

As funções codificadas em linguagens procedurais podem utilizar somente os pseudotipos permitidos pela sua linguagem de implementação. Atualmente, todas as linguagens procedurais proíbem utilizar pseudotipos como tipo do argumento, permitindo apenas void como tipo do resultado (além de trigger, quando a função é utilizada como gatilho).

O pseudotipo internal é utilizado para declarar funções feitas apenas para serem chamadas internamente pelo sistema de banco de dados, e não chamadas a partir de uma consulta SQL. Se a função possui ao menos um tipo de argumento internal então não pode ser chamada por um comando SQL. Para preservar a segurança de tipo desta restrição é importante seguir esta regra de codificação: não criar nenhuma função declarada como retornando o tipo internal, a não ser que haja pelo menos um argumento do tipo internal.

Matrizes

O PostgreSQL permite que colunas de uma tabela sejam definidas como matrizes (arrays) multidimensionais de comprimento variável. Podem ser criadas matrizes de qualquer tipo de dado, nativo ou definido pelo usuário. Para ilustrar a utilização a tabela abaixo é criada:

```
CREATE TABLE sal_emp (
   nome            text,
   pagamento_semanal  integer[],
   agenda          text[][]
);
```

Conforme visto, o tipo de dado matricial é definido adicionando colchetes ([]) ao nome do tipo de dado dos elementos da matriz. O comando acima cria uma tabela chamada sal_emp, contendo uma coluna para cadeia de caracteres do tipo de dado text (nome), uma coluna contendo a matriz unidimensional do tipo de dado integer (pagamento_semanal), representando o pagamento semanal do empregado, e uma coluna contendo a matriz de duas dimensões do tipo de dado text (agenda), que representa a agenda semanal do empregado.

Agora iremos realizar alguns comandos de inserção (INSERT). Observe que um valor matricial para ser escrito deve estar entre chaves e separado por vírgulas. Aqueles que conhecem a linguagem C vão ver que não é diferente da sintaxe para inicializar estruturas (Abaixo são mostrados mais detalhes).

```
INSERT INTO sal_emp
   VALUES ('Bill',
   '{10000, 10000, 10000, 10000}',
   '{{"reunião", "almoço"}, {}}');

INSERT INTO sal_emp
   VALUES ('Carol',
   '{20000, 25000, 25000, 25000}',
   '{{"palestra", "consultoria"}, {"reunião"}}');
```

Agora podem ser feitas algumas consultas à tabela sal_emp. Primeiro será mostrado como acessar um único elemento de uma matriz de cada vez. Esta consulta obtém os nomes dos empregados cujos pagamentos mudaram na segunda semana:

```
SELECT nome FROM sal_emp WHERE pagamento_semanal[1] <> pagamento_semanal[2];

 nome
-------
 Carol
(1 row)
```

Os números dos elementos da matriz são escritos entre colchetes. Por padrão, o PostgreSQL utiliza a convenção de base um para a numeração dos elementos da matriz, ou seja, as matrizes com n elementos começam por array[1] e terminam por array[n].

A consulta abaixo obtém o pagamento da terceira semana de todos os empregados:

```
SELECT pagamento_semanal[3] FROM sal_emp;

 pagamento_semanal
------------------
          10000
          25000
(2 rows)
```

Também podem ser acessadas faixas retangulares de uma matriz, ou submatrizes. Uma faixa de uma matriz é especificada escrevendo-se limite-inferior:limite-superior para uma ou mais dimensões da matriz. A consulta abaixo obtém o primeiro item da agenda do Bill para os primeiros dois dias da semana:

```
SELECT agenda[1:2][1:1] FROM sal_emp WHERE nome = 'Bill';

      agenda
--------------------
 {{reunião},{""}}
(1 row)
```

Também é possível escrever

```
SELECT agenda[1:2][1] FROM sal_emp WHERE nome = 'Bill';
```

para obter o mesmo resultado. Uma operação para identificar elementos de uma matriz é considerada como representando uma faixa da matriz se for escrita na forma inferior:superior. Um limite inferior igual a 1 é estabelecido para qualquer índice quando somente um valor for especificado.

O valor de uma matriz pode ser inteiramente substituído:

```
UPDATE sal_emp SET pagamento_semanal = '{25000,25000,27000,27000}' WHERE nome = 'Carol';
```

ou pode ser atualizado um único elemento:

```
UPDATE sal_emp SET pagamento_semanal[4] = 15000
   WHERE nome = 'Bill';
```

ou pode ser atualizada uma faixa:

```
UPDATE sal_emp SET pagamento_semanal[1:2] = '{27000,27000}'  WHERE nome = 'Carol';
```

Uma matriz pode ser ampliada fazendo atribuição a um elemento adjacente aos já existentes, ou fazendo atribuição a uma faixa que é adjacente ou se sobrepõe aos dados já existentes. Por exemplo,

se uma matriz possui atualmente quatro elementos, esta matriz terá cinco elementos após uma atualização fazer a atribuição de array[5]. Atualmente, as ampliações desta forma somente são permitidas para matrizes unidimensionais, não sendo permitidas para matrizes multi-dimensionais.

A atribuição de fatia de matriz permite a criação de matrizes que não utilizam índices baseados em um. Por exemplo, pode ser feita a atribuição array[-2:7] para criar uma matriz com índices variando de -2 a 7.

A sintaxe do CREATE TABLE permite a criação de matrizes de comprimento fixo:

```
CREATE TABLE jogo_da_velha ( casa   integer[3][3] );
```

Entretanto, a implementação atual não impõe os limites do tamanho da matriz --- o comportamento é o mesmo das matrizes de comprimento não especificado.

Na verdade, a implementação atual também não impõe o número declarado de dimensões. As matrizes de um determinado tipo de elemento são todos considerados como sendo do mesmo tipo, não importando o tamanho ou o número de dimensões. Portanto, a declaração do número de dimensões ou dos tamanhos no comando CREATE TABLE é simplesmente uma documentação, que não afeta o comportamento em tempo de execução.

As dimensões atuais de qualquer valor de uma matriz podem ser obtidas por meio da função array_dims:

```
SELECT array_dims(agenda) FROM sal_emp WHERE nome = 'Carol';

 array_dims
------------
 [1:2][1:1]
(1 row)
```

A função array_dims produz um resultado do tipo text conveniente para as pessoas lerem, mas talvez não muito conveniente para os programas.

Para procurar um valor em uma matriz deve ser verificado cada valor da matriz. Pode ser feito à mão (se for conhecido o tamanho da matriz):

```
SELECT * FROM sal_emp WHERE pagamento_semanal[1] = 10000 OR
                pagamento_semanal[2] = 10000 OR pagamento_semanal[3] = 10000 OR
                pagamento_semanal[4] = 10000;
```

Entretanto, esta forma pode ser entediante para matrizes grandes, e não servirá se a matriz for de tamanho desconhecido. Embora não faça parte da distribuição primária do PostgreSQL, existe uma extensão disponível que define novas funções e operadores para interagir com valores de matrizes. Utilizando esta extensão, a consulta poderia ser:

```
SELECT * FROM sal_emp WHERE pagamento_semanal[1:4] *= 10000;
```

Para procurar em toda a matriz (e não apenas nas colunas especificadas), poderia ser utilizado:

SELECT * FROM sal_emp WHERE pagamento_semanal *= 10000;

Além disso, poderiam ser encontradas as linhas onde a matriz tivesse todos os valores iguais a 10 000 com:

SELECT * FROM sal_emp WHERE pagamento_semanal **= 10000;

Para instalar este módulo opcional veja o diretório contrib/array da distribuição do fonte do PostgreSQL.

Dica: Matrizes não são conjuntos; a utilização de matrizes da forma descrita no parágrafo anterior é geralmente um indicativo de um projeto ruim de banco de dados. O campo matriz deve geralmente ser separado em uma tabela a parte. As tabelas obviamente podem ser pesquisadas com facilidade.

Nota: Uma limitação da implementação atual das matrizes é que os elementos individuais de uma matriz não podem ter atribuídos o valor nulo do SQL. Toda a matriz pode ser definida como nulo, mas não é possível ter uma matriz com alguns elementos nulo e alguns não. A solução deste problema está na lista de coisas a fazer.

Sintaxe de entrada e de saída das matrizes. A representação externa do valor de uma matriz é composta por itens que são interpretados de acordo com as regras de conversão de I/O para o tipo do elemento da matriz, mais os elementos que indicam a estrutura da matriz. Estes elementos são compostos por chaves ({ e }) em torno do valor da matriz, mais os caracteres delimitadores entre os itens adjacentes. O caractere delimitador geralmente é a vírgula (,), mas pode ser outra coisa: é determinado pela definição de typdelim para o tipo de elemento da matriz (Entre os tipos de dado padrão fornecidos na distribuição do PostgreSQL, o tipo box utiliza o ponto-e-vírgula (;), mas todos os outros utilizam a vírgula). Em uma matriz multidimensional cada dimensão (linha, plano, cubo, etc.) recebe seu nível próprio de chaves, e os delimitadores devem ser escritos entre entidades de chaves adjacentes do mesmo nível. Podem ser colocados espaços antes da chave de abertura, após a chave de fechamento, ou antes de qualquer item individual cadeia de caracteres. Espaços em branco após um item não são ignorados, entretanto: após saltar os espaços em branco iniciais, tudo até a próxima chave de fechamento ou delimitador é tomado como sendo o valor do item.

Colocando elementos da matriz entre aspas. Conforme mostrado acima, ao escrever um valor de uma matriz pode-se colocar aspas em torno de qualquer elemento individual da matriz. Isto deve ser feito se o valor do elemento puder de alguma forma confundir o analisador de elemento da matriz. Por exemplo, os elementos contendo chaves, vírgulas (ou qualquer que seja o caractere delimitador), aspas, contrabarras ou espaços iniciais devem estar entre aspas. Para colocar aspas ou contrabarras no valor do elemento da matriz, estes devem ser precedidos por uma contrabarra. Como alternativa, pode ser utilizado o escape de contrabarra para proteger todos os caracteres de dado que seriam de outra forma considerados como sintaxe da matriz ou espaços em branco ignoráveis.

A rotina de saída da matriz coloca aspas em torno dos valores dos elementos caso estes sejam cadeias de caracteres vazias ou contenham chaves, caracteres delimitadores, aspas, contrabarras ou espaços em branco. Aspas ou contrabarras embutidas nos valores dos elementos têm a contrabarra

de escape. Para os tipos de dado numéricos é seguro supor que as aspas nunca vão aparecer, mas para os tipos de dado textuais deve-se estar preparado para lidar tanto com a presença quanto com a ausência das aspas (Esta é uma mudança de comportamento em relação às versões anteriores a 7.2 do PostgreSQL).

Dica: Lembre-se que quando se escreve um comando SQL este é interpretado primeiro como um literal cadeia de caracteres e depois como uma matriz. Isto dobra o número de contrabarras necessárias. Por exemplo, para inserir um valor do tipo text contendo uma contrabarra e uma aspa, deve ser escrito

```
INSERT ... VALUES ('{"\\\\","\\""}');
```

O processador de literal cadeia de caracteres remove um nível de contrabarras, portanto o que chega ao analisador de valor de matriz se parece com {"\\","\""}. Por sua vez, as cadeias de caracteres introduzidas na rotina de entrada de dado do tipo text se tornan \ e ", respectivamente (Se estivéssemos trabalhando com um tipo de dado cuja rotina de entrada também tratasse contrabarras de forma especial como, por exemplo, bytea, seriam necessárias oito contrabarras no comando para obter uma contrabarra armazenada no elemento da matriz).

CAPÍTULO 9. Funções e Operadores

O PostgreSQL disponibiliza várias funções e operadores para os tipos de dado nativos. Os usuários também podem definir suas próprias funções e operadores, conforme descrito no Guia do Programador do PostgreSQL. Os comandos \df e \do do psql podem ser utilizados para mostrar a lista de todas as funções e operadores disponíveis, respectivamente.

Havendo preocupação em relação à portabilidade, deve-se ter em mente que a maior parte das funções e operadores descritos neste capítulo não são especificadas no padrão SQL, com exceção dos operadores mais triviais para aritmética e para comparação, e de algumas funções indicadas explicitamente. Algumas destas funcionalidades estendidas estão presentes em outras implementações do SQL e, em muitos casos, a funcionalidade é compatível e consistente entre vários produtos.

Operadores lógicos

Os operadores lógicos habituais estão disponíveis:

AND
OR
NOT

O SQL utiliza a lógica booleana de três valores, onde o valor nulo representa o "desconhecido". Observe as seguintes tabelas 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

Operadores de comparação

Os operadores de comparação habituais estão disponíveis, conforme mostrado na Tabela 6-1.

Tabela - Operadores de comparação

Operador	Descrição
<	menor que
>	maior que

<=	menor que ou igual a
>=	maior que ou igual a
=	igual
<> ou !=	diferente

Nota: O operador != é convertido em <> no estágio de análise. Não é possível implementar os operadores != e <> realizando operações diferentes.

Os operadores de comparação estão disponíveis em todos os tipos de dado onde fazem sentido. Todos os operadores de comparação são operadores binários que retornam valores do tipo boolean; expressões como 1 < 2 < 3 não são válidas (porque não existe o operador < para comparar um valor booleano com 3).

Além dos operadores de comparação, a construção especial BETWEEN está disponível.

a BETWEEN x AND y

equivale a

a >= x AND a <= y

Igualmente,

a NOT BETWEEN x AND y

equivale a

a < x OR a > y

Não existe diferença entre estas duas formas, além dos ciclos de CPU necessários para reescrever a primeira forma na segunda internamente.

Para verificar se um valor é ou não nulo devem ser usadas as construções

expressão IS NULL
expressão IS NOT NULL

ou às construções equivalentes, mas fora do padrão,

expressão ISNULL
expressão NOTNULL

Não deve ser escrito expressão = NULL, porque NULL não é "igual a" NULL (O valor nulo representa o valor desconhecido, não se podendo saber se dois valores desconhecidos são iguais).

Algumas aplicações podem (incorretamente) requerer que expressão = NULL retorne verdade se a expressão resultar no valor nulo. Para apoiar estas aplicações a opção em tempo de execução transform_null_equals pode ser ativada (por exemplo, SET transform_null_equals TO ON;). Com

isso o PostgreSQL converte a cláusula x = NULL em x IS NULL. Este era o comportamento padrão nas versões de 6.5 a 7.1.

Os valores booleanos também podem ser testados utilizando as construções

expressão IS TRUE
expressão IS NOT TRUE
expressão IS FALSE
expressão IS NOT FALSE
expressão IS UNKNOWN
expressão IS NOT UNKNOWN

Estas formas são semelhantes a IS NULL porque sempre retornam falso ou verdade, nunca o valor nulo, mesmo quando o operando é nulo. A entrada nula é tratada como o valor lógico "desconhecido".

Funções e operadores matemáticos

Estão disponíveis operadores matemáticos para vários tipos de dado 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.

Tabela - Operadores matemáticos

Nome	Descrição	Exemplo	Resultado
+	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
!!	fatorial (operador de prefixo)		120
@	valor absoluto	@ -5.0	5
&	AND binário	91 & 15	11
|	OR binário	32 | 3	
#	XOR binário	17 # 5	
~	NOT binário	~1	
<<	deslocamento binário à esquerda		
>>	deslocamento binário à direita		

Os operadores "binários" também estão disponíveis para os tipos cadeia de bits BIT e BIT VARYING, conforme mostrado na Tabela 6-3. Os argumentos de cadeia de bits para &, | e # devem ter o mesmo comprimento. Ao ser feito o deslocamento de bits, o comprimento original da cadeia de bits é preservado, conforme mostrado na tabela abaixo.

Tabela - Operadores binários para cadeias de bit

Exemplo	Resultado
B'10001' & B'01101'	00001
B'10001' | B'01101'	11101
B'10001' # B'01101'	11110
~ B'10001'	01110
B'10001' << 3	01000
B'10001' >> 2	00100

Nesta tabela "dp" significa double precision. Muitas funções estão disponíveis de várias formas, possuindo argumentos com tipos de dado diferentes. Exceto onde for indicado, todas as formas das funções retornam o mesmo tipo de dado do argumento. As funções que trabalham com dados do tipo double precision são, em sua maioria, implementadas a partir da biblioteca C do sistema; a precisão e o comportamento em casos limites podem, portanto, variar dependendo do sistema hospedeiro.

Tabela - Funções matemáticas

Função	Tipo retornado	Descrição	Exemplo	Resultado
abs(x)	(o mesmo de x)	valor absoluto	abs(-17.4)	17.4
cbrt(dp)	dp	raiz cúbica	cbrt(27.0)	3
ceil(dp ou numeric)	(o mesmo da entrada)	o menor inteiro não menor que o argumento	ceil(-42.8)	-42
degrees(dp)	dp	radianos para graus	degrees(0.5)	28.6478897565412
exp(dp ou numeric)	(o mesmo da entrada)	exponenciação	exp(1.0)	2.71828182845905
floor(dp ou numeric)	(o mesmo da entrada)	o maior inteiro não maior que o argumento	floor(-42.8)	-43
ln(dp ou numeric)	(o mesmo da entrada)	logaritmo natural	ln(2.0)	0.693147180559945
log(dp ou numeric)	(o mesmo da entrada)	logaritmo na base 10	log(100.0)	2
log(b numeric, x numeric)	numeric	logaritmo na base b	log(2.0, 64.0)	6.0000000000
mod(y, x)	(o mesmo tipo dos argumentos)	resto de y/x	mod(9,4)	1
pi()	dp	constante "Pi"	pi()	3.14159265358979
pow(x dp, e dp)	dp	eleva um número ao expoente e	pow(9.0, 3.0)	729
pow(x numeric, e numeric)	numeric	eleva um número ao expoente e	pow(9.0, 3.0)	729

radians(dp) dp
graus para radianos radians(45.0)
0.785398163397448

random() dp valor randômico entre 0.0 e 1.0 random()
round(dp ou numeric) (o mesmo da entrada) arredonda para o inteiro mais próximo
round(42.4) 42
round(v numeric, s integer) numeric arredonda com s casas decimais
round(42.4382, 2) 42.44
sign(dp ou numeric) (o mesmo da entrada) sinal do argumento (-1, 0, +1) sign(-8.4)
-1
sqrt(dp ou numeric) (o mesmo da entrada) raiz quadrada sqrt(2.0) 1.4142135623731
trunc(dp ou numeric) (o mesmo da entrada) trunca com zero casas decimais
trunc(42.8) 42
trunc(v numeric, s integer) numeric trunca com s casas decimais
trunc(42.4382, 2) 42.43

Todas as funções trigonométricas recebem argumentos e retornam valores do tipo double precision.

Tabela - Funções trigonométricas

Função	Descrição
acos(x)	função inversa do cosseno
asin(x)	função inversa do seno
atan(x)	função inversa da tangente
atan2(x, y)	função inversa da tangente de x/y
cos(x)	cosseno
cot(x)	cotangente
sin(x)	seno
tan(x)	tangente

Funções e operadores para cadeias de caracteres

Esta seção descreve as funções e operadores disponíveis para examinar e manipular cadeias de caracteres. Neste contexto as cadeias de caracteres incluem valores dos tipos CHARACTER, CHARACTER VARYING e TEXT. A menos que seja dito o contrário, todas as funções listadas abaixo trabalham com todos estes tipos, mas deve ser tomado cuidado quando for utilizado o tipo CHARACTER com os efeitos em potencial do preenchimento automático. De modo geral, as funções descritas nesta seção também trabalham com dados 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. Estas funções também são implementadas utilizando a sintaxe regular de chamada de funções.

Tabela - Funções e operadores SQL para cadeias de caracteres

Função Tipo retornado Descrição Exemplo Resultado
cadeia_de_caracteres || cadeia_de_caracteres
text
Concatenação de cadeias de caracteres
'Post' || 'greSQL'
PostgreSQL

bit_length(cadeia_de_caracteres)
integer
Número de bits na cadeia de caracteres
bit_length('jose')
32

char_length(cadeia_de_caracteres) ou character_length(cadeia_de_caracteres)
integer
Número de caracteres na cadeia de caracteres
char_length('jose')
4

convert(cadeia_de_caracteres using nome_da_conversão)
text
Muda a codificação utilizando o nome de conversão especificado. As conversões podem ser definidas pelo comando CREATE CONVERSION. Além disso, existem alguns nomes de conversão pré-definidos. Consulte a Tabela 6-8 para ver os nomes de conversão disponíveis.
convert('PostgreSQL' using iso_8859_1_to_utf_8)
'PostgreSQL' na codificação Unicode (UTF-8)

lower(cadeia_de_caracteres)
text
Converte a cadeia de caracteres em letras minúsculas
lower('TOM')
tom

octet_length(cadeia_de_caracteres)
integer
Número de bytes na cadeia de caracteres
octet_length('jose')
4

overlay(cadeia_de_caracteres placing cadeia_de_caracteres from integer [for integer])
text
Inserir substring
overlay('Txxxxas' placing 'hom' from 2 for 4)
Thomas

position(caracteres in cadeia_de_caracteres)
integer
Localização dos caracteres especificados
position('om' in 'Thomas')
3

substring(cadeia_de_caracteres [from integer] [for integer])
text
Extrai parte da cadeia de caracteres
substring('Thomas' from 2 for 3)
hom

substring(cadeia_de_caracteres from expressão)
text
Extrai a parte da cadeia de caracteres correspondente à expressão regular POSIX
substring('Thomas' from '...$')
mas

substring(cadeia_de_caracteres from expressão for escape)
text
Extrai a parte da cadeia de caracteres correspondente à expressão regular SQL
substring('Thomas' from '%#"o_a#"_' for '#')
oma

trim([leading | trailing | both] [caracteres] from cadeia_de_caracteres)
text
Remove da extremidade inicial/final/ambas da cadeia_de_caracteres, a cadeia de caracteres mais longa contendo apenas os caracteres (espaço por padrão)
trim(both 'x' from 'xTomxx')
Tom

upper(cadeia_de_caracteres)
text
Converte a cadeia de caracteres em letras maiúsculas
upper('tom')
TOM

Estão disponíveis funções adicionais para manipulação de cadeias de caracteres conforme listado na Tabela abaixo. Algumas delas são utilizadas internamente para implementar funções de cadeia de caracteres padrão do SQL.

Tabela - Outras funções para cadeia de caracteres

Função	Tipo retornado	Descrição	Exemplo	Resultado
ascii(text)	integer	código ASCII do primeiro caractere do argumento		
ascii('x')	120			

btrim(cadeia_de_caracteres text, trim text) text
Remove (trim) a maior cadeia de caracteres composta apenas pelos caracteres contidos em trim, do

início e do fim da cadeia_de_caracteres
btrim('xyxtrimyyx','xy') trim
chr(integer) text Caractere com o código ASCII fornecido
chr(65) A

convert(cadeia_de_caracteres text, [codificação_origem name,] codificação_destino name)
text
Converte a cadeia de caracteres na codificação_destino. A codificação de origem é especificada por codificação_origem. Se codificação_origem for omitida, a codificação do banco de dados será utilizada.
convert('text_in_unicode', 'UNICODE', 'LATIN1')
text_in_unicode com a representação ISO 8859-1

decode(cadeia_de_caracteres text, tipo text)
bytea
Decodifica os dados binários da cadeia_de_caracteres previamente codificada com encode(). O tipo do parâmetro é o mesmo da da função encode().
decode('MTIzAAE=', 'base64')
123\000\001

encode(dados bytea, tipo text)
text
Codificar os dados binários na representação somente ASCII. Os tipos suportados são: base64, hex e escape.
encode('123\\000\\001', 'base64')
MTIzAAE=

initcap(text)
text
Converte a primeira letra de cada palavra (separadas por espaço em branco) em maiúscula
initcap('hi thomas')
Hi Thomas

length(cadeia_de_caracteres)
integer
Comprimento da cadeia de caracteres
length('jose')
4

lpad(cadeia_de_caracteres text, comprimento integer [, preenchimento text])
text
Preenche a cadeia_de_caracteres até o comprimento adicionando os caracteres de preenchimento (espaço por padrão). Se a cadeia_de_caracteres for mais longa que o comprimento então é truncada (à direita).
lpad('hi', 5, 'xy')
xyxhi

ltrim(cadeia_de_caracteres text, text text)

text
Remove do início da cadeia de caracteres, a cadeia de caracteres mais longa contendo apenas caracteres de trim
ltrim('zzzytrim','xyz')
trim

pg_client_encoding()
name
Nome da codificação atual do cliente
pg_client_encoding()
SQL_ASCII

quote_ident(cadeia_de_caracteres text)
text
Retorna a cadeia de caracteres fornecida, devidamente entre aspas, para ser utilizada como identificador em cadeia de caracteres de consulta SQL. As aspas são adicionadas somente quando há necessidade (ou seja, se a cadeia de caracteres contiver caracteres não identificadores ou se acontecer conversão de letras maiúsculas para minúsculas). As aspas internas são devidamente duplicadas.
quote_ident('Foo')
"Foo"

quote_literal(cadeia_de_caracteres text)
text
Retorna a cadeia de caracteres fornecida, devidamente entre apóstrofos, para ser utilizada como literal em cadeia de caracteres de consulta SQL. Apóstrofos e contrabarras embutidos são devidamente duplicados.
quote_literal('O\'Reilly')
'O''Reilly'

repeat(text, integer)
text
Repete o texto o número de vezes
repeat('Pg', 4)
PgPgPgPg

replace(cadeia_de_caracteres text, origem text, destino text)
text
Substitui todas as ocorrências de origem na cadeia_de_caracteres por destino
replace('abcdefabcdef', 'cd', 'XX')
abXXefabXXef

rpad(cadeia_de_caracteres text, comprimento integer [, preenchimento text])
text
Preenche a cadeia_de_caracteres até o comprimento adicionando os caracteres de preenchimento (espaço por padrão). Se a cadeia_de_caracteres for mais longa que o comprimento estão é truncada.
rpad('hi', 5, 'xy')
hixyx

rtrim(cadeia_de_caracteres text, trim text)
text
Remove do fim da cadeia de caracteres, a cadeia de caracteres mais longa contendo apenas caracteres de trim
rtrim('trimxxxx','x')
trim

split_part(cadeia_de_caracteres text, delimitador text, coluna integer)
text
Separa a cadeia_de_caracteres utilizando o delimitador, retornando a coluna especificada (1 para a primeira coluna).
split_part('abc~@~def~@~ghi','~@~',2)
def

strpos(cadeia_de_caracteres, substring)
text
Localiza a substring especificada (o mesmo que position(substring in cadeia_de_caracteres), mas deve ser observada a ordem inversa dos argumentos)
strpos('high','ig')
2

substr(cadeia_de_caracteres, origem [, contador])
text
Extrai a substring especificada (o mesmo que substring(cadeia_de_caracteres from origem for contador))
substr('alphabet', 3, 2)
ph

to_ascii(text [, codificação])
text
Converte texto em outras codificações em ASCII [a]
to_ascii('Karel')
Karel

to_hex(número integer ou bigint)
text
Converte número em sua representação hexadecimal equivalente
to_hex(9223372036854775807::bigint)
7fffffffffffffff

translate(cadeia_de_caracteres text, origem text, destino text)
text
Todo caractere da cadeia_de_caracteres, correspondente a um caractere do conjunto origem, é substituído pelo caractere correspondente do conjunto destino.
translate('12345', '14', 'ax')
a23x5

Notas:
a. A função to_ascii permite conversão de LATIN1, LATIN2 e WIN1250 apenas.

Tabela - Conversões nativas

Nome da conversão [a]	Codificação de origem	Codificação de destino
ascii_to_mic	SQL_ASCII	MULE_INTERNAL
ascii_to_utf_8	SQL_ASCII	UNICODE
big5_to_euc_tw	BIG5	EUC_TW
big5_to_mic	BIG5	MULE_INTERNAL
big5_to_utf_8	BIG5	UNICODE
euc_cn_to_mic	EUC_CN	MULE_INTERNAL
euc_cn_to_utf_8	EUC_CN	UNICODE
euc_jp_to_mic	EUC_JP	MULE_INTERNAL
euc_jp_to_sjis	EUC_JP	SJIS
euc_jp_to_utf_8	EUC_JP	UNICODE
euc_kr_to_mic	EUC_KR	MULE_INTERNAL
euc_kr_to_utf_8	EUC_KR	UNICODE
euc_tw_to_big5	EUC_TW	BIG5
euc_tw_to_mic	EUC_TW	MULE_INTERNAL
euc_tw_to_utf_8	EUC_TW	UNICODE
gb18030_to_utf_8	GB18030	UNICODE
gbk_to_utf_8	GBK	UNICODE
iso_8859_10_to_utf_8	LATIN6	UNICODE
iso_8859_13_to_utf_8	LATIN7	UNICODE
iso_8859_14_to_utf_8	LATIN8	UNICODE
iso_8859_15_to_utf_8	LATIN9	UNICODE
iso_8859_16_to_utf_8	LATIN10	UNICODE
iso_8859_1_to_mic	LATIN1	MULE_INTERNAL
iso_8859_1_to_utf_8	LATIN1	UNICODE
iso_8859_2_to_mic	LATIN2	MULE_INTERNAL
iso_8859_2_to_utf_8	LATIN2	UNICODE
iso_8859_2_to_windows_1250	LATIN2	WIN1250
iso_8859_3_to_mic	LATIN3	MULE_INTERNAL
iso_8859_3_to_utf_8	LATIN3	UNICODE
iso_8859_4_to_mic	LATIN4	MULE_INTERNAL
iso_8859_4_to_utf_8	LATIN4	UNICODE
iso_8859_5_to_koi8_r	ISO_8859_5	KOI8
iso_8859_5_to_mic	ISO_8859_5	MULE_INTERNAL

Nome da conversão [a]	Codificação de origem	Codificação de destino
iso_8859_5_to_utf_8	ISO_8859_5	UNICODE
iso_8859_5_to_windows_1251	ISO_8859_5	WIN
iso_8859_5_to_windows_866	ISO_8859_5	ALT
iso_8859_6_to_utf_8	ISO_8859_6	UNICODE
iso_8859_7_to_utf_8	ISO_8859_7	UNICODE
iso_8859_8_to_utf_8	ISO_8859_8	UNICODE
iso_8859_9_to_utf_8	LATIN5	UNICODE
johab_to_utf_8	JOHAB	UNICODE
koi8_r_to_iso_8859_5	KOI8	ISO_8859_5
koi8_r_to_mic	KOI8	MULE_INTERNAL
koi8_r_to_utf_8	KOI8	UNICODE
koi8_r_to_windows_1251	KOI8	WIN
koi8_r_to_windows_866	KOI8	ALT
mic_to_ascii	MULE_INTERNAL	SQL_ASCII
mic_to_big5	MULE_INTERNAL	BIG5
mic_to_euc_cn	MULE_INTERNAL	EUC_CN
mic_to_euc_jp	MULE_INTERNAL	EUC_JP
mic_to_euc_kr	MULE_INTERNAL	EUC_KR
mic_to_euc_tw	MULE_INTERNAL	EUC_TW
mic_to_iso_8859_1	MULE_INTERNAL	LATIN1
mic_to_iso_8859_2	MULE_INTERNAL	LATIN2
mic_to_iso_8859_3	MULE_INTERNAL	LATIN3
mic_to_iso_8859_4	MULE_INTERNAL	LATIN4
mic_to_iso_8859_5	MULE_INTERNAL	ISO_8859_5
mic_to_koi8_r	MULE_INTERNAL	KOI8
mic_to_sjis	MULE_INTERNAL	SJIS
mic_to_windows_1250	MULE_INTERNAL	WIN1250
mic_to_windows_1251	MULE_INTERNAL	WIN
mic_to_windows_866	MULE_INTERNAL	ALT
sjis_to_euc_jp	SJIS	EUC_JP
sjis_to_mic	SJIS	MULE_INTERNAL
sjis_to_utf_8	SJIS	UNICODE
tcvn_to_utf_8	TCVN	UNICODE
uhc_to_utf_8	UHC	UNICODE
utf_8_to_ascii	UNICODE	SQL_ASCII
utf_8_to_big5	UNICODE	BIG5
utf_8_to_euc_cn	UNICODE	EUC_CN
utf_8_to_euc_jp	UNICODE	EUC_JP

Nome da conversão [a]	Codificação de origem	Codificação de destino
utf_8_to_euc_kr	UNICODE	EUC_KR
utf_8_to_euc_tw	UNICODE	EUC_TW
utf_8_to_gb18030	UNICODE	GB18030
utf_8_to_gbk	UNICODE	GBK
utf_8_to_iso_8859_1	UNICODE	LATIN1
utf_8_to_iso_8859_10	UNICODE	LATIN6
utf_8_to_iso_8859_13	UNICODE	LATIN7
utf_8_to_iso_8859_14	UNICODE	LATIN8
utf_8_to_iso_8859_15	UNICODE	LATIN9
utf_8_to_iso_8859_16	UNICODE	LATIN10
utf_8_to_iso_8859_2	UNICODE	LATIN2
utf_8_to_iso_8859_3	UNICODE	LATIN3
utf_8_to_iso_8859_4	UNICODE	LATIN4
utf_8_to_iso_8859_5	UNICODE	ISO_8859_5
utf_8_to_iso_8859_6	UNICODE	ISO_8859_6
utf_8_to_iso_8859_7	UNICODE	ISO_8859_7
utf_8_to_iso_8859_8	UNICODE	ISO_8859_8
utf_8_to_iso_8859_9	UNICODE	LATIN5
utf_8_to_johab	UNICODE	JOHAB
utf_8_to_koi8_r	UNICODE	KOI8
utf_8_to_sjis	UNICODE	SJIS
utf_8_to_tcvn	UNICODE	TCVN
utf_8_to_uhc	UNICODE	UHC
utf_8_to_windows_1250	UNICODE	WIN1250
utf_8_to_windows_1251	UNICODE	WIN
utf_8_to_windows_1256	UNICODE	WIN1256
utf_8_to_windows_866	UNICODE	ALT
utf_8_to_windows_874	UNICODE	WIN874
windows_1250_to_iso_8859_2	WIN1250	LATIN2
windows_1250_to_mic	WIN1250	MULE_INTERNAL
windows_1250_to_utf_8	WIN1250	UNICODE
windows_1251_to_iso_8859_5	WIN	ISO_8859_5
windows_1251_to_koi8_r	WIN	KOI8
windows_1251_to_mic	WIN	MULE_INTERNAL
windows_1251_to_utf_8	WIN	UNICODE
windows_1251_to_windows_866	WIN	ALT
windows_1256_to_utf_8	WIN1256	UNICODE
windows_866_to_iso_8859_5	ALT	ISO_8859_5

Nome da conversão [a]	Codificação de origem	Codificação de destino
windows_866_to_koi8_r	ALT	KOI8
windows_866_to_mic	ALT	MULE_INTERNAL
windows_866_to_utf_8	ALT	UNICODE
windows_866_to_windows_1251	ALT	WIN
windows_874_to_utf_8	WIN874	UNICODE

Notas:
a. Os nomes das conversões obedecem a um esquema padrão de nomes: O nome oficial da codificação de origem com todos os caracteres não alfanuméricos substituídos por sublinhado, seguido por _to_, seguido pelo nome da codificação de destino processado da mesma forma. Portanto, os nomes podem desviar dos nomes habituais das codificações.

Funções e operadores para cadeias binárias

Esta seção descreve as funções e operadores disponíveis para examinar e manipular cadeias binárias. As cadeias binárias neste contexto significam valores do tipo BYTEA.

O SQL define algumas funções para cadeias binárias com uma sintaxe especial onde certas palavras chave, em vez de vírgulas, são utilizadas para separar os argumentos. Os detalhes estão na Tabela 6-9. Algumas funções também são implementadas utilizando a sintaxe regular de chamada de função.

Tabela 6-9. Funções e operadores SQL para cadeias binárias

Função
Tipo retornado
Descrição
Exemplo
Resultado

cadeia_de_caracteres || cadeia_de_caracteres
bytea
Concatenação de cadeias binárias
'\\\\Post'::bytea || '\\047greSQL\\000'::bytea
\\Post'greSQL\000

octet_length(cadeia_de_caracteres)
integer
Número de bytes da cadeia binária
octet_length('jo\\000se'::bytea)
5

position(substring in cadeia_de_caracteres)
integer
Localização da substring especificada
position('\\000om'::bytea in 'Th\\000omas'::bytea)
3

substring(cadeia_de_caracteres [from integer] [for integer])
bytea
Extrair substring
substring('Th\\000omas'::bytea from 2 for 3)
h\000o

trim([both] caracteres from cadeia_de_caracteres)
bytea
Remove do início/fim/ambas as extremidades da cadeia_de_caracteres, a cadeia mais longa contendo apenas os caracteres
trim('\\000'::bytea from '\\000Tom\\000'::bytea)
Tom

get_byte(cadeia_de_caracteres, deslocamento)
integer
Extrai byte da cadeia
get_byte('Th\\000omas'::bytea, 4)
109

set_byte(cadeia_de_caracteres, deslocamento, novo_valor)
bytea
Define byte na cadeia
set_byte('Th\\000omas'::bytea, 4, 64)
Th\000o@as

get_bit(cadeia_de_caracteres, deslocamento)
integer
Extrai bit da cadeia
get_bit('Th\\000omas'::bytea, 45)
1

set_bit(cadeia_de_caracteres, deslocamento, novo_valor)
bytea
Define bit na cadeia
set_bit('Th\\000omas'::bytea, 45, 0)
Th\000omAs

Tabela - Outras funções para cadeias binárias

Função Tipo retornado Descrição Exemplo Resultado

btrim(cadeia_de_caracteres bytea trim bytea)
bytea
Remove (trim) do início e do fim da cadeia_de_caracteres, a cadeia mais longa composta apenas por caracteres de trim
btrim('\\000trim\\000'::bytea,'\\000'::bytea)
trim

length(cadeia_de_caracteres)
integer
Comprimento da cadeia binária
length('jo\\000se'::bytea)
5

encode(cadeia_de_caracteres bytea, tipo text)
text
Codifica a cadeia binária para a representação somente ASCII Os tipos suportados são: base64, hex, escape.
encode('123\\000456'::bytea, 'escape')
123\000456

decode(cadeia_de_caracteres text, tipo text)
bytea
Decodifica a cadeia binária da cadeia_de_caracteres previamente codificada com encode(). O tipo do parâmetro é o mesmo do encode().
decode('123\\000456', 'escape')
123\000456

Comparação com padrão

Existem três abordagens separadas para comparação com padrão fornecidas pelo PostgreSQL: o operador LIKE do SQL, o operador mais recente SIMILAR TO do SQL99, e as expressões regulares no estilo POSIX. Além destas, a função de comparação com padrão SUBSTRING também está disponível, utilizando tanto o estilo SQL99 quanto as expressões regulares POSIX.

Dica: Havendo necessidade de comparação com padrão não atendidas, deve ser considerado o desenvolvimento de uma função definida pelo usuário em Perl ou Tcl.

LIKE

cadeia_de_caracteres LIKE filtro [ESCAPE caractere_de_escape]
cadeia_de_caracteres NOT LIKE filtro [ESCAPE caractere_de_escape]

Cada filtro 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 filtro (Como esperado, a expressão NOT LIKE retorna falso quando LIKE retorna verdade, e vice-versa. A expressão equivalente seria NOT (cadeia_de_caracteres LIKE filtro)).

Quando o filtro não possui os caracteres percentagem ou sublinhado, o filtro representa apenas a própria cadeia de caracteres; neste caso LIKE atua como o operador igual. O sublinhado (_) no filtro significa (corresponde a) qualquer caractere único; o caractere percentagem (%) corresponde a qualquer cadeia com zero ou mais caracteres.

Alguns exemplos:

'abc' LIKE 'abc' true
'abc' LIKE 'a%' true
'abc' LIKE '_b_' true
'abc' LIKE 'c' false

A comparação com padrão LIKE sempre abrange toda a cadeia de caracteres. Para haver correspondência com o padrão em qualquer posição da cadeia de caracteres, o filtro precisa começar e terminar pelo caractere percentagem.

Para corresponder ao próprio caractere sublinhado ou percentagem, e não a outros caracteres, estes caracteres no filtro devem estar precedidos pelo caractere de escape. O padrão para caractere de escape é a contrabarra, mas um outro caractere pode ser definido através da cláusula ESCAPE. Para corresponder ao próprio caractere de escape, devem ser escritos dois caracteres de escape.

Deve ser observado que a contrabarra possui significado especial nas cadeias de caracteres e, portanto, para escrever um filtro contendo uma contrabarra devem ser escritas duas contrabarras na consulta. Assim sendo, escrever um filtro correspondente à contrabarra significa escrever quatro contrabarras na consulta, o que pode ser evitado escolhendo um caractere de escape diferente na cláusula ESCAPE; assim a contrabarra deixa de ser um caractere especial para o LIKE (Mas continua sendo especial para o analisador de cadeias de caracteres e, por isso, continuam sendo necessárias duas contrabarras).

Também é possível fazer nenhum caractere servir de escape declarando ESCAPE ''. Esta declaração tem como efeito desativar o mecanismo de escape, tornando impossível evitar o significado especial dos caracteres sublinhado e percentagem presentes no filtro.

A palavra chave ILIKE pode ser utilizada no lugar do LIKE para fazer a comparação de não diferenciar letras maiúsculas de minúsculas, conforme a localização ativa. Isto não faz parte do padrão SQL, sendo extensão do PostgreSQL.

O operador ~~ equivale ao LIKE, enquanto ~~* equivale ao ILIKE. Também existem os operadores !~~ e !~~*, representando o NOT LIKE e o NOT ILIKE. Todos estes operadores são específicos do PostgreSQL.

SIMILAR TO e as expressões regulares do SQL99

cadeia_de_caracteres SIMILAR TO filtro [ESCAPE caractere_de_escape]
cadeia_de_caracteres NOT SIMILAR TO filtro [ESCAPE caractere_de_escape]

O operador SIMILAR TO retorna verdade ou falso conforme o filtro corresponda ou não à cadeia de caracteres fornecida. Este operador é muito semelhante ao LIKE, exceto por interpretar o filtro utilizando a definição de expressão regular do SQL99. As expressões regulares do SQL99 são um cruzamento curioso entre a notação do LIKE e a notação das expressões regulares comuns.

Da mesma forma que o LIKE, o operador SIMILAR TO somente tem êxito quando o filtro corresponde a toda cadeia de caracteres; é diferente da prática comum para expressões regulares, onde o filtro pode corresponder a qualquer parte da cadeia de caracteres. Também como o LIKE, o operador SIMILAR TO utiliza % e _ como caracteres curingas, representando qualquer cadeia de

caracteres e qualquer caractere único, respectivamente (sendo semelhantes aos .* e . das expressões regulares POSIX).

Além destas facilidades tomadas emprestada do LIKE, o SIMILAR TO suporta os seguintes metacaracteres para comparação com padrão tomados emprestado das expressões regulares POSIX:

| representa a alternância (uma das duas alternativas).

* representa a repetição do item anterior zero ou mais vezes.

+ representa a repetição do item anterior uma ou mais vezes.

Os parênteses () podem ser utilizados para agrupar itens num único item lógico.

A expressão de colchetes [...] especifica uma classe de caracteres, do mesmo modo que na expressão regular POSIX.

Observe que a repetição limitada (bounded repetition) (? e {...}) não está disponível, embora exista no POSIX. Além disso, o ponto (.) não é um metacaractere.

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

Alguns exemplos:

```
'abc' SIMILAR TO 'abc'      true
'abc' SIMILAR TO 'a'        false
'abc' SIMILAR TO '%(b|d)%'  true
'abc' SIMILAR TO '(b|c)%'   false
```

A função SUBSTRING com três parâmetros, SUBSTRING(cadeia_de_caracteres FROM filtro FOR escape), permite a extração da parte da cadeia de caracteres correspondente à expressão regular SQL99 no filtro. Assim como em SIMILAR TO, o filtro especificado deve corresponder a toda a cadeia de caracteres do dado, senão a função falha e retorna nulo. Para indicar a parte do filtro que deve ser retornada em caso de sucesso, o SQL99 especifica que o filtro deve conter duas ocorrências do caractere de escape seguido por aspas ("). O texto correspondente à parte do filtro entre estes marcadores é retornada.

Alguns exemplos:

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

Expressões regulares POSIX

Tabela - Operadores disponíveis para comparação padrão de expressões regulares POSIX.

Operador	Descrição	Exemplo

~	Corresponde à expressão regular, diferenciando maiúsculas/minúsculas	'thomas' ~ '.*thomas.*'
~*	Corresponde à expressão regular, não diferenciando maiúsculas/minúsculas	'thomas' ~* '.*Thomas.*'
!~	Não corresponde à expressão regular, diferenciando maiúsculas/minúsculas	'thomas' !~ '.*Thomas.*'
!~*	Não corresponde à expressão regular, não diferenciando maiúsculas/minúsculas	'thomas' !~* '.*vadim.*'

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

Uma expressão regular é uma seqüência de caracteres que é 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 filtro correspondem exatamente aos caracteres da cadeia de caracteres, a não ser que sejam caracteres especiais da linguagem da expressão regular --- porém as expressões regulares utilizam caracteres especiais diferentes dos utilizados pelo LIKE. Ao contrário do LIKE, uma expressão regular pode corresponder a qualquer parte da cadeia de caracteres, a não ser que a expressão regular esteja explicitamente ancorada ao início ou ao fim da cadeia de caracteres.

Alguns exemplos:

'abc' ~ 'abc' true
'abc' ~ '^a' true
'abc' ~ '(b|d)' true
'abc' ~ '^(b|c)' false

A função SUBSTRING com dois parâmetros, SUBSTRING(cadeia_de_caracteres FROM filtro), permite extrair a parte da cadeia de caracteres correspondente à expressão regular SQL99 no filtro. A função retorna nulo quando não há correspondência, senão retorna a parte do texto que corresponde ao filtro. Mas se o filtro contiver parênteses, a parte do texto correspondente à primeira subexpressão entre parênteses (aquela cujo abre parênteses vem primeiro) é retornada. Pode ser colocado parênteses envolvendo toda a expressão se for desejado utilizar parênteses em seu interior sem disparar esta exceção.

Alguns exemplos:

SUBSTRING('foobar' FROM 'o.b') oob
SUBSTRING('foobar' FROM 'o(.)b') o

As expressões regulares (REs), conforme definidas no POSIX 1003.2, estão presentes em duas formas: REs modernas (as do egrep aproximadamente; chamadas no 1003.2 de REs "estendidas"), e REs obsoletas (as do ed aproximadamente; REs "básicas" do 1003.2). O PostgreSQL implementa a forma moderna.

Uma expressão regular (moderna) é composta por uma ou mais subdivisões não vazias, separadas por |. A expressão corresponde a qualquer coisa que corresponda a uma das suas subdivisões.

Uma subdivisão é uma ou mais peças, concatenadas. Equivale a uma correspondência para a primeira, seguida por uma correspondência para a segunda, etc.

Uma peça é um átomo, possivelmente seguido por um único *, +, ? ou limite. Um átomo seguido por um * corresponde a uma série de 0 ou mais ocorrências do átomo. Um átomo seguido por um + corresponde a uma série de 1 ou mais ocorrências do átomo. Um átomo seguido por um ? corresponde a uma série de 0 ou 1 ocorrências do átomo.

Um limite é uma { seguida por um número decimal inteiro sem sinal, possivelmente seguido por uma , possivelmente seguida por um número decimal inteiro sem sinal, sempre seguido por uma }. Os inteiros devem estar entre 0 e RE_DUP_MAX (255) inclusive e, havendo dois deles, o primeiro não pode ser maior que o segundo. Um átomo seguido por um limite contendo um inteiro i e nenhuma vírgula, corresponde a uma série de exatamente i ocorrências do átomo. Um átomo seguido por um limite contendo um inteiro i e uma vírgula, corresponde a uma seqüência de i ou mais ocorrências do átomo. Um átomo seguido por um limite contendo dois inteiros i e j, corresponde a uma série de i até j (inclusive) ocorrências do átomo.

Nota: Um operador de repetição (?, *, + ou limites) não pode vir depois de outro operador de repetição. O operador de repetição não pode iniciar uma expressão, uma subexpressão, ou vir depois de ^ ou |.

Um átomo pode ser uma expressão regular entre () (representando a expressão regular), um conjunto vazio de () (representando a cadeia de caracteres nula), uma expressão de colchete (veja abaixo), . (representando qualquer caractere único), ^ (representando a cadeia de caracteres nula no início da cadeia de caracteres da entrada), $ (representando a cadeia de caracteres nula no final da cadeia de caracteres da entrada), uma \ seguida por um dos caracteres ^.[$()|*+?{\ (representando este caractere como sendo um caractere ordinário), uma \ seguida por qualquer outro caractere (representando este caractere como sendo um caractere ordinário, como se a \ não estivesse presente), ou um caractere único sem qualquer outro significado (representando o próprio caractere). Uma { seguida por um outro caractere que não seja um dígito é um caractere ordinário, e não o início de um limite. É ilegal terminar uma expressão regular por uma \.

Observe que a contrabarra (\) possui um significado especial nas cadeias de caracteres e, portanto, para escrever uma constante filtro contendo uma contrabarra devem ser escritas duas contrabarras na consulta.

Uma expressão entre colchetes é uma seqüência de caracteres entre []. Normalmente significa qualquer um dos caracteres da seqüência (mas veja abaixo). Se o conjunto começar por ^, significa qualquer caractere não presente no restante do conjunto (mas veja abaixo). Se dois caracteres do conjunto estiverem separados por -, isto representa a forma abreviada de todos os caracteres do intervalo delimitado por estes dois caracteres (inclusive) na seqüência de arrumação (collating sequence). Por exemplo, [0-9] em ASCII significa qualquer dígito decimal. É ilegal dois intervalos compartilharem uma mesma extremidade como, por exemplo, a-c-e. Os intervalos são dependentes da seqüência de arrumação dos caracteres, e os programas portáveis devem evitar esta dependência.

Para incluir o literal] na seqüência, deve ser feito com que seja o primeiro caractere (possivelmente após o ^). Para incluir o literal -, deve ser feito com que seja o primeiro ou o último caractere, ou a segunda extremidade de um intervalo. Para utilizar o literal - como a primeira extremidade de um

intervalo, deve-se colocá-lo entre [. e .] para torná-lo um elemento de arrumação (veja abaixo). Com estas excessões e algumas outras combinações utilizando [(veja os próximos parágrafos), todos os outros caracteres especiais, incluindo a \, perdem seu significado especial dentro de uma expressão entre colchetes.

Dentro da expressão entre colchetes, o elemento de arrumação (um caractere, uma seqüência de múltiplos caracteres arrumada como se fosse um único caractere, ou o nome de uma seqüência de arrumação) entre [. e .] representa a seqüência de caracteres deste elemento de arrumação. A seqüência é um único elemento do conjunto da expressão entre colchetes. Uma expressão entre colchetes contendo um elemento de arrumação de múltiplos caracteres pode, portanto, corresponder a mais de um caractere. Por exemplo, se a seqüência de arrumação incluir um elemento de arrumação ch, então a expressão regular [[.ch.]]*c corresponde aos cinco primeiros caracteres de chchcc.

Dentro de uma expressão entre colchetes, um elemento de arrumação entre [= e =] é uma classe de equivalência, representando as seqüências de caracteres de todos os elementos de arrumação equivalentes a este elemento, incluindo o próprio (Se não existirem outros elementos de arrumação equivalentes, o tratamento é como se os delimitadores envoltórios fossem [. e .]). Por exemplo, se o e ^ são membros de uma classe de equivalência, então [[=o=]], [[=^=]] e [o^] são todos sinônimos. Uma classe de equivalência não pode ser a extremidade de um intervalo.

Dentro de uma expressão entre colchetes, o nome de uma classe de caracteres entre [: e :] representa o conjunto de todos os caracteres pertencentes a esta classe. Os nomes das classes de caracteres padrão são: alnum, alpha, blank, cntrl, digit, graph, lower, print, punct, space, upper, xdigit. Estes nomes representam as classes de caracteres definidas em ctype. Uma localização pode fornecer outras. Uma classe de caracteres não pode ser usada como uma extremidade de um intervalo.

Existem dois casos especiais para expressões entre colchetes: as expressões entre colchetes [[:<:]] e [[:>:]] correspondem à cadeia de caracteres nula no início e no fim de uma palavra, respectivamente. Uma palavra é definida como a seqüência de caracteres de palavra que não é nem precedida nem seguida por caracteres de palavra. Um caractere de palavra é um alnum caractere (conforme definido em ctype) ou um sublinhado. Isto é uma extensão, compatível mas não especificada pelo POSIX 1003.2, devendo ser utilizada com cautela em programas onde se deseja a portabilidade para outros sistemas.

Quando a expressão regular puder corresponder a mais de uma parte da cadeia de caracteres, a expressão regular corresponderá à parte que começar primeiro na cadeia de caracteres. Se a expressão regular puder corresponder a mais de uma parte começando no mesmo ponto, corresponderá à mais longa. As subexpressões também correspondem às partes mais longas possíveis, sujeitas à restrição que toda a correspondência seja a mais longa possível, com as subexpressões começando antes na expressão regular tendo prioridade em relação às começando depois. Observe que, deste modo, as subexpressões de maior nível têm prioridade em relação às suas subexpressões componentes de nível mais baixo.

Os comprimentos das correspondências são medidos em caracteres, e não em elementos de arrumação. Uma cadeia de caracteres nula é considerada como sendo mais longa que nenhuma correspondência. Por exemplo, bb* corresponde aos três caracteres do meio de abbbc; (wee|week)(knights|nights) corresponde a todos os dez caracteres de weeknights; quando (.*).* é comparado

com abc a subexpressão entre parênteses corresponde a todos os três caracteres; e quando (a*)* é comparado com bc tanto a expressão regular completa quanto a subexpressão entre parênteses correspondem a cadeia de caracteres nula.

Se for especificada uma correspondência que não diferencie letras maiúsculas e minúsculas, o efeito é como se a distinção entre maiúsculas e minúsculas tivesse desaparecido do alfabeto. Quando um caractere alfabético que existe em maiúscula e minúscula aparece como caractere ordinário fora da expressão entre colchetes, este caractere é transformado em uma expressão entre colchetes contendo as duas representações, por exemplo, x se torna [xX]. Quando o caractere aparece dentro da expressão entre colchetes, a representação que falta é adicionada à expressão entre colchetes, portanto (por exemplo) [x] se torna [xX] e [^x] se torna [^xX].

Não existe nenhum limite determinado para o comprimento da expressão regular, a não ser o fato da memória ser limitada. A utilização de memória é aproximadamente linear em relação ao tamanho da expressão regular, e altamente independente da complexidade da expressão regular, exceto para repetições limitadas. As repetições limitadas são implementadas por meio de expansão de macro, gastando muito tempo e espaço quando os contadores são grandes ou as repetições limitadas são aninhadas. Uma expressão regular como, digamos, ((((a{1,100}){1,100}){1,100}){1,100}){1,100} irá (ao fim) fazer quase todas as máquinas existentes esgotarem suas áreas de permuta (swap). [11]

Funções para formatar tipos de dado

As funções de formatação do PostgreSQL disponibilizam um poderoso conjunto de ferramentas para converter diversos tipos de dado (date/time, integer, floating point, numeric) em cadeias de caracteres formatadas, e para converter cadeias de caracteres formatadas nos tipos de dado especificados. A tabela abaixo mostra estas funções, que seguem uma convenção de chamada comum: o primeiro argumento é o valor a ser formatado, e o segundo argumento é a máscara que define o formato da entrada ou da saída.

Tabela - Funções de formatação

Função	Retorna	Descrição	Exemplo
to_char(timestamp, text)	text	converte carimbo de tempo (time stamp) em cadeia de caracteres	to_char(timestamp 'now','HH12:MI:SS')
to_char(interval, text)	text	converte intervalo em cadeia de caracteres	to_char(interval '15h 2m 12s','HH24:MI:SS')
to_char(int, text)	text	converte inteiro em cadeia de caracteres	to_char(125, '999')
to_char(double precision, text)	text		

11Tenha em mente que isto foi escrito em 1994. Os números provavelmente são outros, mas o problema continua.

converte real e precisão dupla em cadeia de caracteres
to_char(125.8, '999D9')

to_char(numeric, text) text
converte numérico em cadeia de caracteres
to_char(numeric '-125.8', '999D99S')

to_date(text, text) date
converte cadeia de caracteres em data
to_date('05 Dec 2000', 'DD Mon YYYY')

to_timestamp(text, text) timestamp
converte cadeia de caracteres em carimbo de tempo
to_timestamp('05 Dec 2000', 'DD Mon YYYY')

to_number(text, text) numeric
converte cadeia de caracteres em numérico
to_number('12,454.8-', '99G999D9S')

Na cadeia de caracteres usada como máscara de saída, existem certos elementos que são reconhecidos e substituídos pelos dados devidamente formatados a partir do valor a ser formatado. Qualquer texto que não faça parte de um elemento da máscara é simplesmente copiado sem alteração. Da mesma forma, na cadeia de caracteres usada como máscara de entrada, os elementos da máscara identificam as partes procuradas na cadeia de caracteres da entrada, e os valores a ser encontrados nestas posições.

Tabela - Elementos para máscara de conversão de data e hora

Elemento	Descrição
HH	hora do dia (01-12)
HH12	hora do dia (01-12)
HH24	hora do dia (00-23)
MI	minuto (00-59)
SS	segundo (00-59)
MS	milissegundo (000-999)
US	microssegundo (000000-999999)
SSSS	segundos após a meia-noite (0-86399)
AM ou A.M. ou PM ou P.M.	indicador de meridiano (maiúsculas)
am ou a.m. ou pm ou p.m.	indicador de meridiano (minúsculas)
Y,YYY	ano (4 e mais dígitos) com vírgula
YYYY	ano (4 e mais dígitos)
YYY	últimos 3 dígitos do ano
YY	últimos 2 dígitos do ano
Y	último dígito do ano
BC ou B.C. ou AD ou A.D.	indicador de era (maiúscula)
bc ou b.c. ou ad ou a.d.	indicador de era (minúscula)
MONTH	nome completo do mês em maiúsculas (9 caracteres completado com espaços)

Month	nome completo do mês em maiúsculas e minúsculas (9 caracteres completado com espaços)
month	nome completo do mês em minúsculas (9 caracteres completado com espaços)
MON	nome abreviado do mês em maiúsculas (3 caracteres)
Mon	nome abreviado do mês em maiúsculas e minúsculas (3 caracteres)
mon	nome abreviado do mês em minúsculas (3 caracteres)
MM	número do mês (01-12)
DAY	nome completo do dia em maiúsculas (9 caracteres completado com espaços)
Day	nome completo do dia em maiúsculas e minúsculas (9 caracteres completado com espaços)
day	nome completo do dia em minúsculas (9 caracteres completado com espaços)
DY	nome abreviado do dia em maiúsculas (3 caracteres)
Dy	nome abreviado do dia em maiúsculas e minúsculas (3 caracteres)
dy	nome abreviado do dia em minúsculas (3 caracteres)
DDD	dia do ano (001-366)
DD	dia do mês (01-31)
D	dia da semana (1-7; SUN=1)
W	semana do mês (1-5) onde a primeira semana começa no primeiro dia do mês
WW	número da semana do ano (1-53) onde a primeira semana começa no primeiro dia do ano
IW	número da semana do ano ISO (A primeira quinta-feira do novo ano está na semana 1)
CC	século (2 dígitos)
J	Dia Juliano (dias desde 1 de janeiro de 4712 AC)
Q	trimestre
RM	mês em algarismos romanos (I-XII; I=Janeiro) - maiúsculas
rm	mês em algarismos romanos (I-XII; I=Janeiro) - minúsculas
TZ	zona horária - maiúsculas
tz	zona horária - minúsculas

Certos modificadores podem ser aplicados a qualquer elemento da máscara para alterar seu comportamento. Por exemplo, "FMMonth" é o elemento "Month" com o prefixo "FM".

Tabela - Modificadores dos elementos das máscara de conversão de data e hora

Modificador	Descrição	Exemplo
prefixo FM	modo de preenchimento (suprime completar com brancos e zeros)	FMMonth
sufixo TH	adicionar o sufixo de número ordinal em maiúsculas	DDTH
sufixo th	adicionar o sufixo de número ordinal em minúsculas	DDth
prefixo FX	opção global de formato fixo (veja nota de utilização)	FX Month DD Day

sufixo SP modo de falar (spell mode) (ainda não implementado) DDSP

Notas relativas à utilização da formatação de data e hora:

O FM suprime zeros à esquerda e espaços à direita, que de outra forma seriam adicionados para fazer o elemento da saída ter comprimento fixo.

As funções to_timestamp e to_date desprezam os múltiplos espaços em branco na cadeia de caracteres de entrada quando a opção FX não é utilizada. O FX deve ser especificado como o primeiro elemento da máscara; por exemplo to_timestamp('2000 JUN','YYYY MON') está correto, mas to_timestamp('2000 JUN','FXYYYY MON') retorna erro, porque to_timestamp espera um único espaço em branco.

Se for desejada a presença de uma contrabarra ("\") em uma constante cadeia de caracteres, devem ser escritas duas contrabarras ("\\"); por exemplo '\\HH\\MI\\SS'. Isto vale para qualquer constante cadeia de caracteres no PostgreSQL.

Texto comum é permitido nas máscaras da função to_char, sendo reproduzidos literalmente. Podem ser colocados caracteres entre aspas para serem interpretados como texto literal, mesmo contendo palavras chaves para elemento. Por exemplo, em '"Hello Year "YYYY', o YYYY é substituído pelo ano da data, porém o Y de "Year" não é.

Se for desejada uma aspa na saída esta deve ser precedida por uma contrabarra. Por exemplo '\\"YYYY Month\\"'.

A conversão YYYY de cadeia de caracteres para timestamp ou para date tem restrições quando são utilizados mais de 4 dígitos para o ano. Deve ser utilizado algum caractere que não seja um dígito ou um outro elemento após YYYY, senão o ano será sempre interpretado como tendo 4 dígitos. Por exemplo, (com o ano 20000): to_date('200001131', 'YYYYMMDD') é interpretado como um ano de 4 dígitos; o melhor é utilizar um separador que não seja um dígito após o ano, como to_date('20000-1131', 'YYYY-MMDD') ou to_date('20000Nov31', 'YYYYMonDD').

Os valores de milissegundos MS e microssegundos US, na conversão de uma cadeia de caracteres para um carimbo de tempo (time stamp), são interpretados como a sendo parte dos segundos após o ponto decimal. Por exemplo to_timestamp('12:3', 'SS:MS') não são 3 milissegundos, mas 300, porque a conversão interpreta como sendo 12 + 0.3. Isto significa que, para o formato SS:MS, os valores de entrada 12:3, 12:30 e 12:300 especificam o mesmo número de milissegundos. Para especificar três milissegundos deve ser utilizado 12:003, que na conversão é interpretado como 12 + 0.003 = 12.003 segundos.

A seguir está mostrado um exemplo mais complexo: to_timestamp ('15:12:02.020.001230','HH:MI:SS.MS.US') é interpretado como 15 horas, 12 minutos, 2 segundos + 20 milissegundos + 1230 microssegundos = 2.021230 segundos.

Tabela - Elementos para máscara de conversão numérica

Elemento	Descrição
9	valor com o número especificado de dígitos

0	valor com zeros à esquerda
. (ponto)	ponto decimal
, (vírgula)	separador de grupo (milhares)
PR	valor negativo entre chaves
S	valor negativo com o sinal de menos (utiliza a localização)
L	símbolo da moeda (utiliza a localização)
D	ponto decimal (utiliza a localização)
G	separador de grupo (utiliza a localização)
MI	sinal de menos na posição especificada (se número < 0)
PL	sinal de mais na posição especificada (se número > 0)
SG	sinal de mais/menos na posição especificada
RN	algarismos romanos (entrada entre 1 e 3999)
TH ou th	converte em número ordinal
V	desloca n dígitos (veja as notas)
EEEE	notação científica (ainda não implementada)

Notas relativas à utilização da formatação numérica:

O sinal formatado utilizando SG, PL ou MI não está ancorado no número; por exemplo, to_char (-12, 'S9999') produz ' -12', mas to_char(-12, 'MI9999') produz '- 12'. A implementação do Oracle não permite utilizar o MI antes do 9, requer que o 9 preceda o MI.

O 9 especifica um valor com o mesmo número de dígitos que o número de 9s. Se não houver um dígito disponível, é colocado um espaço.

O TH não converte valores menores que zero e não converte números decimais.

O PL, o SG e o TH são extensões do PostgreSQL.

O V efetivamente multiplica os valores de entrada por 10^n, onde n é o número de dígitos após o V. A função to_char não aceita utilizar V juntamente com o ponto decimal (Por exemplo, 99.9V99 não é permitido).

Tabela - Exemplos da função to_char

Entrada	Saída
to_char(now(),'Day, DD HH12:MI:SS')	'Tuesday , 06 05:39:18'
to_char(now(),'FMDay, FMDD HH12:MI:SS')	'Tuesday, 6 05:39:18'
to_char(-0.1,'99.99')	' -.10'
to_char(-0.1,'FM9.99')	'-.1'
to_char(0.1,'0.9')	' 0.1'
to_char(12,'9990999.9')	' 0012.0'
to_char(12,'FM9990999.9')	'0012'
to_char(485,'999')	' 485'
to_char(-485,'999')	'-485'
to_char(485,'9 9 9')	' 4 8 5'
to_char(1485,'9,999')	' 1,485'
to_char(1485,'9G999')	' 1 485'

to_char(148.5,'999.999')	' 148.500'
to_char(148.5,'999D999')	' 148,500'
to_char(3148.5,'9G999D999')	' 3 148,500'
to_char(-485,'999S')	'485-'
to_char(-485,'999MI')	'485-'
to_char(485,'999MI')	'485'
to_char(485,'PL999')	'+485'
to_char(485,'SG999')	'+485'
to_char(-485,'SG999')	'-485'
to_char(-485,'9SG99')	'4-85'
to_char(-485,'999PR')	'<485>'
to_char(485,'L999')	'DM 485
to_char(485,'RN')	' CDLXXXV'
to_char(485,'FMRN')	'CDLXXXV'
to_char(5.2,'FMRN')	V
to_char(482,'999th')	' 482nd'
to_char(485, '"Good number:"999')	'Good number: 485'
to_char(485.8,'"Pre:"999" Post:" .999')	'Pre: 485 Post: .800'
to_char(12,'99V999')	' 12000'
to_char(12.4,'99V999')	' 12400'
to_char(12.45, '99V9')	' 125'

Funções e operadores para data e hora

Os detalhes são mostrados nas próximas subseções. A Tabela abaixo ilustra o comportamento dos operadores aritméticos básicos (+, *, etc.). Deve-se estar familiarizado com os os tipos de dado para data e hora.

Todas as funções e operadores descritos abaixo, que recebem os tipos time ou timestamp como entrada, estão presentes em duas formas: uma que recebe time ou timestamp com zona horária, e outra que recebe time ou timestamp sem zona horária. Para abreviar, estas formas não são mostradas em separado.

Tabela - Operadores de data e hora

Nome	Exemplo	Resultado
+	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'
+	time '01:00' + interval '3 hours'	time '04:00'
-	timestamp '2001-09-28 23:00' - interval '23 hours'	timestamp '2001-09-28'
-	date '2001-09-28' - interval '1 hour'	timestamp '2001-09-27 23:00'
-	time '05:00' - interval '2 hours'	time '03:00'
-	interval '2 hours' - time '05:00'	time '03:00:00'
*	interval '1 hour' * int '3'	interval '03:00'
/	interval '1 hour' / int '3'	interval '00:20'

Tabela - Funções de data e hora

Nome	Tipo retornado	Descrição	Exemplo	Resultado
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
current_date	date	Data de hoje;		
current_time	time with time zone	Hora do dia;		
current_timestamp	timestamp with time zone	Data e hora;		
date_part(text, timestamp)	double precision	Obter subcampo (equivale ao extract); veja também abaixo	date_part('hour', timestamp '2001-02-16 20:38:40')	20
date_part(text, interval)	double precision	Obter subcampo (equivale ao extract); veja também abaixo	date_part('month', interval '2 years 3 months')	3
date_trunc(text, timestamp)	timestamp	Truncar na precisão especificada;	date_trunc('hour', timestamp '2001-02-16 20:38:40')	2001-02-16 20:00:00+00
extract(field from timestamp)	double precision	Obter subcampo;	extract(hour from timestamp '2001-02-16 20:38:40')	20
extract(field from interval)	double precision	Obter subcampo;	extract(month from interval '2 years 3 months')	3
isfinite (timestamp)	boolean	Testar carimbo de hora finito (nem inválido nem infinito)	isfinite(timestamp '2001-02-16 21:28:30')	true
isfinite(interval)	boolean	Testar intervalo finito	isfinite(interval '4 hours')	true
localtime	time	Hora do dia;		
localtimestamp	timestamp	Data e hora;		
now()	timestamp with time zone	Data e hora corrente (equivale ao current_timestamp);		
timeofday()	text	Data e hora corrente;	timeofday()	Wed Feb 21 17:01:13.000126 2001 EST

EXTRACT, date_part

EXTRACT (campo FROM fonte)

A função extract obtém subcampos dos valores de data e hora, como o ano ou a hora. A fonte é uma

expressão de valor avaliada aos tipos timestamp ou interval. (As expressões do tipo date e time são transformadas em timestamp, possibilitando utilizá-las da mesma forma). O campo é um identificador, ou uma cadeia de caracteres, que seleciona qual campo será extraído do valor fonte. A função extract retorna valores do tipo double precision. Abaixo são mostrados valores válidos:

century

O campo do ano dividido por 100

SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');
Resultado: 20

Observe que o resultado para o campo século é simplesmente o campo ano dividido por 100, e não a definição habitual que coloca a maior parte dos anos de 1900 no século vinte.

day

O campo do dia (do mês) (1 - 31)

SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
Resultado: 16
decade

O campo do ano dividido por 10

SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
Resultado: 200
dow

O dia da semana (0 - 6; Domingo é 0) (para os valores de timestamp apenas)

SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
Resultado: 5
doy

O dia do ano (1 - 365/366) (para os valores de timestamp apenas)

SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
Resultado: 47
epoch

Para os valores de date e timestamp, o número de segundos desde 1970-01-01 00:00:00-00 (pode ser negativo); para valores de interval, o número total de segundos do intervalo

SELECT EXTRACT(EPOCH FROM TIMESTAMP '2001-02-16 20:38:40');
Resultado: 982352320

SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');

Resultado: 442800
hour

O campo das horas (0 - 23)

SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
Resultado: 20
microseconds

O campo dos segundos, incluindo a parte fracionária, multiplicado por 1 000 000. Observe que inclui todos os segundos.

SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
Resultado: 28500000
millennium

O ano dividido por 1 000

SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
Resultado: 2

Observe que o resultado para o campo milênio é simplesmente o campo ano dividido por 1000, e não a definição habitual que coloca a maior parte dos anos de 1900 no segundo milênio.

milliseconds

O campo de segundos, incluindo a parte fracionária, multiplicado por 1000. Observe que inclui todos os segundos.

SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
Resultado: 28500
minute

O campo dos minutos (0 - 59)

SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
Resultado: 38
month

Para os valores timestamp, o número do mês do ano (1 - 12); para valores de interval o número de meses, módulo 12 (0 - 11)

SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
Resultado: 2

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');
Resultado: 3

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
Resultado: 1
quarter

O trimestre do ano (1 - 4) onde o dia se encontra (para os valores de timestamp apenas)

SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
Resultado: 1
second

O campo dos segundos, incluindo a parte fracionária (0 - 59[1])

SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');
Resultado: 40

SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
Resultado: 28.5
timezone_hour

O componente hora do deslocamento da zona horária
timezone_minute

O componente minuto do deslocamento da zona horária

week

No valor de timestamp, calcula o número da semana do ano onde o dia se encontra. Por definição (ISO 8601), a primeira semana do ano contém o dia 4 de janeiro deste ano (A semana ISO começa na segunda-feira). Em outras palavras, a primeira quinta-feira está na primeira semana do ano.

SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
Resultado: 7
year

O campo do ano

SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
Resultado: 2001

A função extract é voltada principalmente para o processamento computacional. Para formatar valores de data e hora para exibição.

A função date_part está modelada sobre o equivalente no Ingres à função extract do padrão SQL:

date_part('campo', fonte)

Observe que neste caso o parâmetro campo precisa ser um uma cadeia de caracteres, e não um nome. Os valores válidos para campo em date_part são os mesmos da função extract.

SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
Resultado: 16

SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
Resultado: 4

6.8.2. date_trunc

A função date_trunc é conceitualmente similar à função trunc para números.

date_trunc('campo', fonte)

fonte é uma expressão de valor do tipo timestamp (valores do tipo date e time são transformados automaticamente). campo seleciona a precisão a ser utilizada para truncar o valor do carimbo de tempo. O valor retornado é do tipo timestamp, com todos os campos inferiores ao valor selecionado tornados zero (ou um, para o dia do mês).

Os valores válidos para campo são:

microseconds, milliseconds, second, minut, hour, day, month, year, decade, century, millennium

Exemplos:

SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
Resultado: 2001-02-16 20:00:00+00

SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
Resultado: 2001-01-01 00:00:00+00

AT TIME ZONE

A construção AT TIME ZONE permite a conversão do carimbo de tempo para uma zona horária diferente.

Tabela - Variantes de AT TIME ZONE

Expressão	Retorna	Descrição
timestamp without time zone AT TIME ZONE zone	timestamp with time zone	Converte hora local de uma determinada zona horária para UTC
timestamp with time zone AT TIME ZONE zone	timestamp without time zone	Converte de UTC para a hora local em uma determinada zona horária
time with time zone AT TIME ZONE zona	time with time zone	Converte hora local entre zonas horárias

Nestas expressões, a zona horária desejada pode ser especificada tanto por meio de um texto em uma cadeia de caracteres (por exemplo, 'PST') quanto por um intervalo (por exemplo, INTERVAL '-08:00').

Exemplos (suponha que a TimeZone seja PST8PDT):

SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'MST';
Resultado: 2001-02-16 19:38:40-08

SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'MST';
Resultado: 2001-02-16 18:38:40

O primeiro exemplo aceita um carimbo de tempo sem zona horária e o interpreta como hora MST (GMT-7) para produzir um carimbo de tempo UTC, o qual é então rotacionado para PST (GMT-8) para ser exibido. O segundo exemplo aceita um carimbo de tempo especificado em EST (GMT-5) e converte para hora local MST (GMT-7).

A função timezone(zona, carimbo_de_tempo) equivale à construção em conformidade com o padrão SQL carimbo_de_tempo AT TIME ZONE zona.

Data e hora corrente

As seguintes funções estão disponíveis para obtenção da data e hora corrente:

CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME (precisão)
CURRENT_TIMESTAMP (precisão)
LOCALTIME
LOCALTIMESTAMP
LOCALTIME (precisão)
LOCALTIMESTAMP (precisão)

CURRENT_TIME e CURRENT_TIMESTAMP fornecem valores com zona horária; LOCALTIME e LOCALTIMESTAMP fornecem valores sem zona horária.

CURRENT_TIME, CURRENT_TIMESTAMP, LOCALTIME e LOCALTIMESTAMP podem, opcionalmente, receber um parâmetro com a precisão fazendo o resultado ser arredondado nesta quantidade de dígitos fracionários. Sem o parâmetro de precisão, o resultado é fornecido com toda a precisão disponível.

Nota: Antes do PostgreSQL 7.2, os parâmetros de precisão não estavam implementados, e o resultado era sempre fornecido em segundos inteiros.

Alguns exemplos:

```
SELECT CURRENT_TIME;
14:39:53.662522-05

SELECT CURRENT_DATE;
2001-12-23

SELECT CURRENT_TIMESTAMP;
2001-12-23 14:39:53.662522-05

SELECT CURRENT_TIMESTAMP(2);
2001-12-23 14:39:53.66-05

SELECT LOCALTIMESTAMP;
2001-12-23 14:39:53.662522
```

A função now() é o equivalente tradicional do PostgreSQL para CURRENT_TIMESTAMP.

Também existe a função timeofday(), que por motivos históricos retorna uma cadeia de caracteres e não um valor do tipo timestamp:

```
SELECT timeofday();
Sat Feb 17 19:07:32.000126 2001 EST
```

É importante perceber que CURRENT_TIMESTAMP e as funções relacionadas retornam o tempo do começo da transação corrente; seus valores não mudam durante a transação. A função timeofday () retorna a hora corrente, avançando durante as transações.

Nota: Muitos outros sistemas de banco de dados avançam estes valores mais freqüentemente.

Todos os tipos de dado para data e hora também aceitam o valor literal especial now para especificar a data e hora corrente. Portanto, os três comandos abaixo retornam o mesmo resultado:

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now';
```

Nota: Não será desejado utilizar a terceira forma ao especificar a cláusula DEFAULT na criação da tabela. O sistema converte now em timestamp tão logo a constante é analisada e, portanto, quando o valor padrão for usado a hora da criação da tabela será utilizada! As duas primeiras formas não são processadas até que o valor padrão seja utilizado, porque são chamadas de funções. Assim sendo, as duas primeiras formas fornecem o comportamento desejado quando o padrão for a hora de inserção da linha.

Notas
[1] 60 se os segundos saltados (leap) são implementados pelo sistema operacional

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.

Tabela - Operadores geométricos

Operador	Descrição	Utilização
+	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)'
#	Interseção	'((1,-1),(-1,1))' # '((1,1),(-1,-1))'
#	Número de pontos do caminho ou do polígono	# '((1,0),(0,1),(-1,0))'
##	Ponto mais próximo	point '(0,0)' ## lseg '((2,0),(0,2))'
&&	Sobrepõe? box '((0,0),(1,1))' && box '((0,0),(2,2))'	
&<	Sobrepõe para esquerda?	box '((0,0),(1,1))' &< box '((0,0),(2,2))'
&>	Sobrepõe para direita?	box '((0,0),(3,3))' &> box '((0,0),(2,2))'
<->	Distância entre	circle '((0,0),1)' <-> circle '((5,0),1)'
<<	Está à esquerda de?	circle '((0,0),1)' << circle '((5,0),1)'
<^	Está abaixo?	circle '((0,0),1)' <^ circle '((0,5),1)'
>>	Está à direita de?	circle '((5,0),1)' >> circle '((0,0),1)'
>^	Está acima?	circle '((0,5),1)' >^ circle '((0,0),1)'
?#	Cruzamento ou sobreposição	lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))'
?-	É horizontal?	point '(1,0)' ?- point '(0,0)'
?-|	É perpendicular?	lseg '((0,0),(0,1))' ?-| lseg '((0,0),(1,0))'
@-@	Comprimento ou circunferência	@-@ path '((0,0),(1,0))'
?|	É vertical?	point '(0,1)' ?| point '(0,0)'
?||	É paralela?	lseg '((-1,0),(1,0))' ?|| lseg '((-1,2),(1,2))'
@	Contida ou sobre	point '(1,1)' @ circle '((0,0),2)'
@@	Centro de	@@ circle '((0,0),10)'
~=	O mesmo que	polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'

Tabela - Funções geométricas

Função	Retorna	Descrição	Exemplo
area(object)	double precision	área do objeto	area(box '((0,0),(1,1))')
box(box, box)	box	retângulo de interseção	box(box '((0,0),(1,1))',box '((0.5,0.5),(2,2))')
center(object)	point	centro do objeto	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 do retângulo	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 do objeto	length(path '((-1,0),(1,0))')
npoints(path)	integer	número de pontos	npoints(path '[(0,0),(1,1),(2,0)]')
npoints(polygon)	integer	número de pontos	npoints(polygon '((1,1),(0,0))')

pclose(path)	path	converte caminho em caminho fechado	popen(path '[(0,0),(1,1),(2,0)]')
popen(path)	path	converte 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	width(box '((0,0),(1,1))')

Tabela - Funções de conversão de tipo geométrico

Função	Retorna	Conversão	Exemplo
box(circle)	box	círculo em retângulo	box(circle '((0,0),2.0)')
box(point, point)	box	ponto em retângulo	box(point '(0,0)', point '(1,1)')
box(polygon)	box	polígono em retângulo	box(polygon '((0,0),(1,1),(2,0))')
circle(box)	circle	retângulo em círculo	circle(box '((0,0),(1,1))')
circle(point, double precision)		circle ponto em círculo	circle(point '(0,0)', 2.0)
lseg(box)	lseg	diagonal do retângulo em lseg	lseg(box '((-1,0),(1,0))')
lseg(point, point)	lseg	ponto em lseg	lseg(point '(-1,0)', point '(1,0)')
path(polygon)	point	polígono em caminho	path(polygon '((0,0),(1,1),(2,0))')
point(circle)	point	centro	point(circle '((0,0),2.0)')
point(lseg, lseg)	point	interseção	point(lseg '((-1,0),(1,0))', lseg '((-2,-2),(2,2))')
point(polygon)	point	centro	point(polygon '((0,0),(1,1),(2,0))')
polygon(box)	polygon	polígono de 4 pontos	polygon(box '((0,0),(1,1))')
polygon(circle)	polygon	polígono de 12 pontos	polygon(circle '((0,0),2.0)')
polygon(npts, circle)	polygon	polígono de npts	polygon(12, circle '((0,0),2.0)')
polygon(path)	polygon	caminho em polígono	polygon(path '((0,0),(1,1),(2,0))')

É possível acessar os dois números que compõem um ponto, como se 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 obtém a coordenada X; UPDATE t SET p[1] = ... altera a coordenada Y. Do mesmo modo, uma coluna do tipo box ou lseg pode ser tratada como sendo uma matriz contendo dois pontos.

Funções para tipo endereço de rede

A tabela abaixo mostra os operadores disponíveis para os tipos inet e cidr. Os operadores <<, <<=, >>, >>= testam a inclusão na subrede: somente consideram as partes de rede destes dois endereços, ignorando qualquer parte do hospedeiro, determinando se a parte de rede é idêntica ou uma subrede da outra.

Tabela - Operadores cidr e inet

Operador	Descrição	Utilização
<	Menor que	inet '192.168.1.5' < inet '192.168.1.6'
<=	Maior que ou igual	inet '192.168.1.5' <= inet '192.168.1.5'
=	Igual	inet '192.168.1.5' = inet '192.168.1.5'
>=	Maior ou igual	inet '192.168.1.5' >= inet '192.168.1.5'
>	Maior	inet '192.168.1.5' > inet '192.168.1.4'
<>	Diferente	inet '192.168.1.5' <> inet '192.168.1.4'

<<	está contido em	inet '192.168.1.5' << inet '192.168.1/24'
<<=	está contido em ou é igual	inet '192.168.1/24' <<= inet '192.168.1/24'
>>	contém	inet'192.168.1/24' >> inet '192.168.1.5'
>>=	contém ou é igual	inet '192.168.1/24' >>= inet '192.168.1/24'

A tabela abaixo mostra as funções disponíveis para utilizar com os tipos de dado inet e cidr. As funções host(), text() e abbrev() são voltadas principalmente para oferecer alternativas para o formato de exibição. Um campo texto pode ser transformado em endereço de rede utilizando a sintaxe habitual de conversão: inet(expressão) ou nome_da_coluna::inet.

Tabela - Funções cidr e inet

Função	Retorna	Descrição	Exemplo	Resultado
broadcast(inet)	inet	endereço de difusão da rede	broadcast('192.168.1.5/24')	192.168.1.255/24
host(inet)	text	extrai o endereço de IP como texto	host('192.168.1.5/24')	192.168.1.5
masklen(inet)	integer	extrai o comprimento da máscara de rede	masklen('192.168.1.5/24')	24
set_masklen(inet,integer)	inet	define o comprimento da máscara de rede para o valor do tipo inet	set_masklen('192.168.1.5/24',16)	192.168.1.5/16
netmask(inet)	inet	constrói máscara de rede para a rede	netmask('192.168.1.5/24')	255.255.255.0
network(inet)	cidr	extrai a parte de rede do endereço	network('192.168.1.5/24')	192.168.1.0/24
text(inet)	text	extrai o endereço de IP e o comprimento da máscara como texto	text(inet '192.168.1.5')	192.168.1.5/32
abbrev(inet)	text	extrai a exibição abreviada como texto	abbrev(cidr '10.1.0.0/16')	10.1/16

A tabela abaixo mostra as funções disponíveis para utilizar com o tipo mac. A função trunc (macaddr) retorna um endereço MAC com os últimos 3 bytes feitos iguais a zero. Isto pode ser utilizado para associar o prefixo remanescente com o fabricante. O diretório contrib/mac da distribuição do fonte contém alguns utilitários para criar e manter este tipo de tabela de associação.

Tabela - Funções para o tipo macaddr

Função	Retorna	Descrição	Exemplo	Resultado
trunc(macaddr)	macaddr	torna os 3 últimos bytes iguais a zero	trunc(macaddr '12:34:56:78:90:ab')	12:34:56:00:00:00

O tipo macaddr também suporta os operadores relacionais padrão (>, <=, etc.) para ordenação lexicográfica.

Funções para manipulação de seqüências

Esta seção descreve as funções do PostgreSQL que operam com objetos de seqüência. Os objetos de seqüência (também chamados de geradores de seqüência ou simplesmente de seqüências), são tabelas especiais de um única linha criadas pelo comando CREATE SEQUENCE. Um objeto de seqüência normalmente é usado para gerar identificadores únicos para linhas de uma tabela. As funções de seqüência, listadas na Tabela 6-26, fornecem métodos simples, seguros para multiusuários, para obter valores sucessivos da seqüência a partir dos objetos de seqüência.

Tabela - Funções de seqüências

Função	Retorna	Descrição
nextval(text)	bigint	Avança a seqüência e retorna o novo valor
currval(text)	bigint	Retorna o valor mais recentemente obtido pelo nextval
setval(text,bigint)	bigint	Estabelece o valor corrente da seqüência
setval(text,bigint,boolean)	bigint	Estabelece o valor corrente da seqüência e sinaliza is_called

Por motivos históricos, a seqüência a ser operada pela chamada da função de seqüência é especificada por um argumento que é um texto em uma cadeia de caracteres. Para obter alguma compatibilidade com o tratamento usual dos nomes no SQL, a função de seqüência converte as letras do argumento em minúsculas, a não ser que a cadeia de caracteres esteja entre aspas. Portanto

nextval('foo') opera na seqüência foo
nextval('FOO') opera na seqüência foo
nextval('"Foo"') opera na seqüência Foo

Havendo necessidade, o nome da seqüência pode ser qualificado pelo esquema:

nextval('meu_esquema.foo') opera em meu_esquema.foo
nextval('"meu_esquema".foo') o mesmo acima
nextval('foo') percorre o caminho de procura buscando foo

É claro que o texto do argumento pode ser o resultado de uma expressão, e não somente um literal simples, o que algumas vezes é útil.

As funções de seqüência disponíveis são:

nextval

Avança o objeto de seqüência para seu próximo valor e retorna este valor. Isto é feito atomicamente: mesmo que várias sessões executem nextval concorrentemente, cada uma recebe um valor distinto da seqüência com segurança.

currval

Retorna o valor obtido mais recentemente por nextval para esta seqüência na sessão corrente (Ocasiona erro se nextval nunca tiver sido chamado para esta seqüência nesta sessão). Como é retornado o valor da sessão, uma resposta previsível é fornecida mesmo que outras sessões também estejam executando nextval.

setval

Redefine o valor do contador do objeto de seqüência. A forma com dois parâmetros define o campo last_value (último valor) da seqüência com o valor especificado, e define o campo is_called como true, indicando que o próximo nextval avançará a seqüência antes de retornar o valor. Na forma com três parâmetros, is_called pode ser definido tanto como true quanto como false. Se for definido como false, o próximo nextval retornará o próprio valor especificado, e o avanço da seqüência somente começará no nextval seguinte. Por exemplo,

```
SELECT setval('foo', 42);           o próximo nextval() retorna 43
SELECT setval('foo', 42, true);     o mesmo acima
SELECT setval('foo', 42, false);    o próximo nextval() retorna 42
```

O resultado retornado por setval é simplesmente o valor de seu segundo argumento.

Importante: Para evitar bloquear transações concorrentes ao obter valores de uma mesma seqüência, a operação nextval nunca é desfeita (rolled back); ou seja, após o valor ser obtido este passa a ser considerado como tendo sido usado, mesmo que depois a transação que executou o nextval aborte. Isto significa que as transações abortadas podem deixar "buracos" não utilizados na seqüência de valores atribuídos. Além disso, as operações setval nunca são desfeitas.

Se o objeto de seqüência tiver sido criado com os parâmetros padrão, as chamadas à nextval() retornam valores sucessivos começando por um. Outros comportamentos podem ser obtidos utilizando parâmetros especiais no comando CREATE SEQUENCE; veja a página de referência deste comando para obter mais informações.

Expressões condicionais

Esta seção descreve as expressões condicionais em conformidade com o SQL disponíveis no PostgreSQL.

Dica: Havendo alguma necessidade não atendida pelas funcionalidades destas expressões condicionais, deve ser considerado o desenvolvimento de um procedimento armazenado usando uma linguagem de programação com mais recursos.

CASE

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

A expressão CASE do SQL é uma expressão condicional genérica, semelhante às declarações if/else de outras linguagens. A cláusula CASE pode ser empregada sempre que a utilização de uma expressão for válida. A condição é uma expressão que retorna um resultado booleano. Se a condição for verdade, então o valor da expressão CASE é o resultado. Se a condição for falsa, todas as cláusulas WHEN seguintes são percorridas da mesma maneira. Se nenhuma condição WHEN for

verdade, então o valor da expressão CASE é o valor do resultado na cláusula ELSE. Se a cláusula ELSE for omitida, e nenhuma condição for satisfeita, o resultado será nulo.

Um exemplo:

```
=> 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
```

Os tipos de dado de todas as expressões para resultado devem poder ser convertidos no mesmo tipo de dado de saída.

```
CASE expressão
   WHEN valor THEN resultado
   [WHEN ...]
   [ELSE resultado]
END
```

Esta expressão CASE "simplificada" é uma variante especializada da forma geral mostrada acima. A expressão é calculada e comparada com todos os valores das cláusulas WHEN, até ser encontrado um igual. Se nenhum valor igual for encontrado, o resultado na cláusula ELSE (ou o valor nulo) é retornado. Esta forma é semelhante à declaração switch da linguagem C.

O exemplo mostrado anteriormente pode ser escrito utilizando a sintaxe simplificada do CASE:

```
=> SELECT a,
        CASE a WHEN 1 THEN 'um'
             WHEN 2 THEN 'dois'
             ELSE 'outro'
        END
   FROM teste;
 a | case
---+-------
```

1 | um
2 | dois
3 | outro

COALESCE

COALESCE(valor [, ...])

A função COALESCE retorna o primeiro de seus argumentos que não for nulo. Geralmente é útil para substituir o valor padrão dos valores nulos quando os dados são usados para exibição. Por exemplo:

SELECT COALESCE(descricao, descricao_curta, '(nenhuma)') ...

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 utilizado para realizar a operação inversa do exemplo para COALESCE mostrado acima:

SELECT NULLIF(valor, '(nenhuma)') ...

Dica: Tanto o COALESCE como o NULLIF são apenas formas abreviadas das expressões CASE. Na realidade, são convertidos em expressões CASE em um estágio bem inicial do processamento, fazendo o processamento subseqüente supor que está lidando com o CASE. Por isso, a utilização incorreta do COALESCE ou do NULLIF pode produzir uma mensagem de erro fazendo referência ao CASE.

Funções diversas

A tabela abaixo mostra diversas funções que obtêm informações da sessão e do sistema.

Tabela - Funções de informação da sessão

Nome	Tipo retornado	Descrição
current_database()	name	nome do banco de dados corrente
current_schema()	name	nome do esquema corrente
current_schemas(boolean)	name[]	nomes dos esquemas no caminho de procura incluindo, opcionalmente, os esquemas implícitos
current_user	name	nome do usuário do contexto de execução corrente
session_user	name	nome do usuário da sessão
user	name	equivale ao current_user
version()	text	informação da versão do PostgreSQL

O session_user é o usuário que estabeleceu a conexão com o banco de dados; imutável durante a

conexão. O current_user é o identificador do usuário utilizado para verificação de permissão. Normalmente é igual ao usuário da sessão, mas muda durante a execução das funções com o atributo SECURITY DEFINER. Na terminologia Unix, o usuário da sessão é o "usuário real" e o usuário corrente é o "usuário efetivo".

Nota: O current_user, o session_user e o user possuem status sintático especial no SQL: devem ser chamados sem parênteses.

O current_schema retorna o nome do esquema que está em primeiro lugar no caminho de procura (ou o valor nulo se o caminho de procura estiver vazio). Este é o esquema utilizado para qualquer tabela ou outro objeto nomeado que for criado sem especificar o esquema a ser utilizado. O current_schemas(boolean) retorna uma matriz com os nomes de todos os esquemas presentes no caminho de procura. A opção booleana determina se os esquemas incluídos implicitamente sistema, tal como pg_catalog, serão incluídos no caminho de procura retornado.

O caminho de procura pode ser alterado por uma configuração em tempo de execução. O comando a ser utilizado é o SET SEARCH_PATH 'esquema'[,'esquema']...

A função version() retorna uma cadeia de caracteres descrevendo a versão do servidor PostgreSQL.

A tabela abaixo mostra as funções disponíveis para consultar e modificar os parâmetros de configuração em tempo de execução.

Tabela - Funções de informação dos valores de configuração

Nome	Tipo retornado	Descrição
current_setting(nome_da_configuração)	text	valor atual da configuração
set_config(nome_da_configuração, novo_valor, is_local)	text	novo valor da configuração

A função current_setting é utilizada para obter o valor corrente da configuração nome_da_configuração como o resultado de uma consulta. Equivale ao comando SQL SHOW. Por exemplo:

```
select current_setting('DateStyle');
         current_setting
-------------------------------------
ISO with US (NonEuropean) conventions
(1 row)
```

A função set_config permite que a configuração nome_da_configuração seja alterada para o novo_valor. Se is_local for definido como true, o novo valor somente se aplica à transação corrente. Se for desejado que o novo valor seja aplicado para a sessão corrente, deve ser utilizado false. Equivale ao comando SQL SET. Por exemplo:

```
select set_config('show_statement_stats','off','f');
 set_config
------------
```

```
 off
(1 row)
```

Tabela - Funções de consulta a privilégios de acesso

Nome	Tipo retornado	Verifica se
has_table_privilege(usuário, tabela, acesso)	boolean	o usuário pode acessar a tabela
has_table_privilege(tabela, acesso)	boolean	o usuário corrente pode acessar a tabela
has_database_privilege(usuário, banco_de_dados, acesso)	boolean	o usuário pode acessar o banco de dados
has_database_privilege(banco_de_dados, acesso)	boolean	o usuário corrente pode acessar o banco de dados
has_function_privilege(usuário, função, acesso)	boolean	o usuário pode acessar a função
has_function_privilege(função, acesso)	boolean	o usuário corrente pode acessar a função
has_language_privilege(usuário, linguagem, acesso)	boolean	o usuário pode acessar a linguagem
has_language_privilege(linguagem, acesso)	boolean	o usuário corrente pode acessar a linguagem
has_schema_privilege(usuário, esquema, acesso)	boolean	o usuário pode acessar o esquema
has_schema_privilege(esquema, acesso)	boolean	o usuário corrente pode acessar o esquema

A função has_table_privilege verifica se o usuário pode acessar a tabela de uma determinada forma. O usuário pode ser especificado pelo nome ou pelo ID (pg_user.usesysid) ou, se o argumento for omitido, será utilizado o current_user. A tabela pode ser especificada pelo nome ou pelo OID (Portanto, existem na verdade seis variantes de has_table_privilege, as quais podem ser distinguidas pelo número e pelos tipos de seus argumentos). Quando especificado pelo nome, este pode ser qualificado pelo esquema, se for necessário. A forma de acesso desejada é especificada no texto da cadeia de caracteres, que deve ser avaliado como um dos seguintes valores: SELECT, INSERT, UPDATE, DELETE, RULE, REFERENCES ou TRIGGER (Maiúsculas e minúsculas não fazem diferença). Um exemplo é:

```
SELECT has_table_privilege('meu_esquema.minha_tabela', 'select');
```

A função has_database_privilege verifica se o usuário pode acessar o banco de dados de uma determinada forma. As possibilidades para seus argumentos são análogas às da função has_table_privilege. Os tipos de acesso desejados devem ser avaliados como CREATE, TEMPORARY ou TEMP (que equivale ao TEMPORARY).

A função has_function_privilege verifica se o usuário pode acessar a função de uma determinada forma. As possibilidades para seus argumentos são análogas às da função has_table_privilege. Ao especificar a função por meio de um texto e não pelo seu OID, a entrada permitida é a mesma que para o tipo de dado regprocedure. A forma de acesso desejada deve atualmente ser avaliada como EXECUTE.

A função has_language_privilege verifica se o usuário pode acessar a linguagem procedural de um determinado modo. As possibilidades para seus argumentos são análogas às da função has_table_privilege. A forma de acesso desejada deve atualmente ser avaliado como USAGE.

A função has_schema_privilege verifica se o usuário pode acessar o esquema de uma determinada forma. As possibilidades para seus argumentos são análogas às da função has_table_privilege. A forma de acesso desejada deve atualmente ser avaliado como CREATE ou USAGE.

A tabela abaixo mostra as funções que informam se um determinado objeto está visível no caminho de procura de esquema corrente. Uma tabela é dita visível se o esquema que a contém está no caminho de procura, e nenhuma tabela com o mesmo nome aparece antes no caminho de procura. Equivale declarar que a tabela pode ser referenciada pelo nome sem uma qualificação explícita do esquema. Por exemplo, para listar o nome de todas as tabelas visíveis:

SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);

Tabela - Funções de consulta à visibilidade do esquema

Nome	Tipo retornado	Verifica se
pg_table_is_visible(OID_tabela)	boolean	a tabela está visível no caminho de procura
pg_type_is_visible(OID_tipo)	boolean	o tipo está visível no caminho de procura
pg_function_is_visible(OID_função)	boolean	a função está visível no caminho de procura
pg_operator_is_visible(OID_operador)	boolean	o operador está visível no caminho de procura
pg_opclass_is_visible(oOID_classeop)	boolean	a classe de operador está visível no caminho de procura

A função pg_table_is_visible realiza a verificação para as tabelas (ou visões, ou qualquer outro tipo de entrada na pg_class). As funções pg_type_is_visible, pg_function_is_visible, pg_operator_is_visible e pg_opclass_is_visible realizam o mesmo tipo de verificação de visibilidade para os tipos, funções, operadores e classes de operadores, respectivamente. Para as funções e operadores, um objeto está visível no caminho de procura se não existir nenhum objeto com o mesmo nome e mesmos tipos de dado dos argumentos aparecendo antes no caminho. Para as classes de operadores, tanto o nome quanto o método de acesso do índice associado são considerados.

Todas estas funções requerem os OIDs dos objetos para identificar o objeto a ser verificado. Se for desejado testar o objeto pelo nome é conveniente utilizar os tipos aliases de OID (regclass, regtype, regprocedure ou regoperator), por exemplo

SELECT pg_type_is_visible('meu_esquema.widget'::regtype);

Observe que não faz muito sentido testar um nome não qualificado deste modo --- se o nome puder ser reconhecido então tem que ser visível.

A tabela a seguir lista as funções que extraem informações dos catálogos do sistema. As funções pg_get_viewdef(), pg_get_ruledef(), pg_get_indexdef() e pg_get_constraintdef() reconstroem, respectivamente, os comandos de criação da visão, da regra, do índice e da restrição. (Observe que esta é uma reconstrução por descompilação, e não o texto exato do comando). Atualmente a função pg_get_constraintdef() somente funciona para as restrições de chave estrangeira. A função pg_get_userbyid() obtém o nome do usuário a partir do valor do usesysid.

Tabela - Funções de informação do catálogo

Nome	Tipo retornado	Descrição
pg_get_viewdef(nome_da_visão)	text	Obtém o comando CREATE VIEW para a visão (obsoleto)
pg_get_viewdef(OID_visão)	text	Obtém o comando CREATE VIEW para a visão
pg_get_ruledef(OID_regra)	text	Obtém o comando CREATE RULE para a regra
pg_get_indexdef(OID_índice)	text	Obtém o comando CREATE INDEX para o índice
pg_get_constraintdef(OID_restrição)	text	Obtém a definição da restrição
pg_get_userbyid(ID_usuario)	name	Obtém o nome do usuário com o ID_usuário fornecido

As funções mostradas na Tabela 6-32 obtêm os comentários previamente armazenados por meio do comando COMMENT. Se nenhum comentário correspondendo aos parâmetros especificados puder ser encontrando, o valor nulo é retornado.

Tabela - Funções de informação de comentário

Nome	Tipo retornado	Descrição
obj_description(OID_objeto, nome_da_tabela)	text	Obtém o comentário para o objeto do banco de dados
obj_description(OID_objeto)	text	Obtém o comentário para o objeto do banco de dados (obsoleto)
col_description(OID_tabela, número_da_coluna)	text	Obtém o comentário para a coluna da tabela

A forma da função obj_description() com dois parâmetros retorna o comentário para o objeto do banco de dados especificado pelo seu OID, e o nome do catálogo do sistema que o contém. Por exemplo, obj_description(123456,'pg_class') obtém o comentário para a tabela com OID 123456. A forma de obj_description() com um parâmetro requer apenas o o OID do objeto. Está obsoleta porque não existe garantia dos OIDs serem únicos entre diferentes catálogos do sistema; portanto, um comentário errado pode ser retornado.

A função col_description() retorna o comentário para a coluna da tabela especificada pelo OID da tabela e pelo número da coluna. A função obj_description() não pode ser utilizada para colunas de tabela porque as colunas não possuem OIDs próprios.

Funções de agregação

As funções de agregação calculam um único valor de resultado para um conjunto de valores de entrada. Consulte o Tutorial do PostgreSQL para obter informações introdutórias adicionais.

Tabela - Funções de agregação

Função	Tipo do argumento	Tipo retornado	Descrição
avg(expressão)	smallint, integer, bigint, real, double precision, numeric ou interval.	numeric para qualquer argumento de tipo inteiro, double precision para argumento de tipo ponto flutuante, caso contrário o mesmo tipo de dado do argumento	a média (média aritmética) de todos os valores de entrada
count(*)	bigint		número de valores de entrada
count(expressão)	any	bigint	número de valores de entrada para os quais o valor da expressão não é nulo
max(expressão)	qualquer tipo de dado numérico, cadeia de caracteres, data ou hora	o mesmo tipo de dado do argumento	valor máximo da expressão entre todos os valores de entrada
min(expressão)	qualquer tipo de dado numérico, cadeia de caracteres, data ou hora	o mesmo tipo de dado do argumento	valor mínimo da expressão entre todos os valores de entrada
stddev(expressão)	smallint, integer, bigint, real, double precision ou numeric.	double precision para argumentos de ponto flutuante, caso contrário numeric.	desvio padrão da amostra dos valores de entrada
sum(expressão)	smallint, integer, bigint, real, double precision, numeric ou interval	bigint para argumentos smallint ou integer, numeric para argumentos bigint, double precision para argumentos de ponto flutuante, caso contrário o mesmo tipo de dado do argumento	somatório da expressão para todos os valores de entrada
variance(expressão)	smallint, integer, bigint, real, double precision ou numeric.	double precision para argumentos de ponto flutuante, caso contrário numeric.	variância dos valores de entrada da amostra (quadrado do desvio padrão da amostra)

Deve ser observado que, com exceção do count, estas funções retornam o valor nulo quando nenhuma linha for selecionada. Em particular, sum de nenhuma linha retorna nulo, e não zero como poderia ser esperado. A função coalesce pode ser utilizada para substituir nulo por zero quando for necessário.

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).

EXISTS

EXISTS (subconsulta)

O argumento do EXISTS é uma declaração SELECT arbitrária, ou uma subconsulta. A subconsulta é avaliada 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 fazer referência às variáveis da consulta que a envolve, que atuam como constantes durante a avaliação da subconsulta.

A subconsulta geralmente só é executada até ser determinado se pelo menos uma linha é retornada, e não até o fim. Não é sensato escrever uma subconsulta que tenha efeitos colaterais (tal como chamar uma função de seqüência); se o efeito colateral ocorrerá ou não pode ser difícil de saber.

Uma vez que o resultado depende apenas do fato de alguma linha ser retornada, e não do conteúdo desta linha, normalmente não há interesse no conteúdo da saída da subconsulta. Uma convenção usual de codificação, 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 que existam muitas linhas correspondentes em tab2:

```
SELECT col1 FROM tab1
   WHERE EXISTS(SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

IN (forma escalar)

expressão IN (valor[, ...])

O lado direito desta forma do IN é uma lista de expressões escalares entre parênteses. O resultado é "verdade" se o resultado da expressão à esquerda for igual a qualquer uma das expressões à direita. Esta é uma notação abreviada para

```
expressão = valor1
OR
expressão = valor2
OR
...
```

Quando a expressão à esquerda for nula, ou não havendo nenhum valor igual à direita, e pelo menos uma expressão à direita for nula, o resultado da construção IN será nulo, e não falso. Isto está de acordo com as regras normais do SQL para as combinações booleanas de valores nulos.

Nota: Esta forma do IN não é uma expressão de subconsulta de verdade, mas parece melhor ser documentada no mesmo local da subconsulta IN.

IN (forma de subconsulta)

expressão IN (subconsulta)

O lado direito desta forma do IN é uma subconsulta entre parênteses, que deve retornar exatamente uma coluna. A expressão à esquerda é avaliada e comparada com cada linha do resultado da subconsulta. O resultado do IN é "verdade" se uma linha igual for encontrada no resultado da subconsulta. O resultado é "falso" se nenhuma linha igual for encontrada (incluindo o caso especial onde a subconsulta não retorna nenhuma linha).

Quando a expressão à esquerda for nula, ou não havendo nenhum valor igual à direita, e pelo menos uma das linhas à direita for nula, o resultado da construção IN será nulo, e não falso. Isto está de acordo com as regras normais do SQL para as combinações booleanas de valores nulos.

Da mesma forma que em EXISTS, não é sensato supor que a subconsulta será executada até o fim.

(expressão [, expressão ...]) IN (subconsulta)

O lado direito desta forma do IN é uma subconsulta entre parênteses, que deve retornar tantas colunas quantas forem as expressões existentes na lista do lado esquerdo. As expressões do lado esquerdo são avaliadas e comparadas com cada linha do resultado da subconsulta. O resultado do IN é "verdade" se for encontrada alguma linha igual na subconsulta. O resultado é "falso" se nenhuma linha igual for encontrada (incluindo o caso especial onde a subconsulta não retorna nenhuma linha).

Como usual, os valores nulos nas expressões ou nas linhas da subconsulta são combinados conforme as regras normais do SQL para expressões booleanas. Duas linhas são consideradas iguais se todos os membros correspondentes forem iguais e não nulos; as linhas não são iguais se algum membro correspondente for diferente e não nulo; caso contrário, o resultado da comparação da linha é desconhecido (nulo). Se os resultados de todas as linhas forem diferentes ou nulos, com pelo menos um nulo, então o resultado do IN é nulo.

NOT IN (forma escalar)

expressão NOT IN (valor[, ...])

O lado direito desta forma do NOT IN é uma lista de expressões escalares entre parênteses. O resultado é "verdade" se o resultado da expressão à esquerda for diferente de todas às expressões à direita. Esta é uma notação abreviada para

expressão <> valor1
AND
expressão <> valor2
AND

...

Quando a expressão à esquerda for nula, ou não havendo nenhum valor igual à direita, e pelo menos uma expressão à direita for nula, o resultado da construção NOT IN será nulo, e não verdade como poderia ser esperado. Isto está de acordo com as regras normais do SQL para as combinações booleanas de valores nulos.

Dica: x NOT IN y equivale a NOT (x IN y) em todos os casos. Entretanto, os valores nulos têm muito mais chances de enganar os novatos quando trabalham com NOT IN em vez de IN. É melhor expressar a condição na forma positiva, quando possível.

NOT IN (forma de subconsulta)

expressão NOT IN (subconsulta)

O lado direito desta forma do NOT IN é uma subconsulta entre parênteses, que deve retornar exatamente uma coluna. A expressão do lado esquerdo é avaliada e comparada com cada linha do resultado da subconsulta. O resultado do NOT IN é "verdade" se somente linhas diferentes forem encontradas no resultado da subconsulta (incluindo o caso especial onde a subconsulta não retorna nenhuma linha). O resultado é "falso" se qualquer linha igual for encontrada.

Quando a expressão à esquerda for nula, ou não havendo nenhum valor igual à direita, e pelo menos uma linha da direita for nula, o resultado da construção NOT IN será nulo, e não verdade. Isto está de acordo com as regras normais do SQL para as combinações booleanas de valores nulos.

Do mesmo modo que no EXISTS, não é sensato supor que a subconsulta será executada até o fim.

(expressão [, expressão ...]) NOT IN (subconsulta)

O lado direito desta forma do NOT IN é uma subconsulta entre parênteses, que deve retornar tantas colunas quantas forem as expressões existentes na lista do lado esquerdo. As expressões do lado esquerdo são avaliadas e comparadas com cada linha do resultado da subconsulta. O resultado do NOT IN é "verdade" se somente linhas diferentes forem encontradas na subconsulta (incluindo o caso especial onde a subconsulta não retorna nenhuma linha. O resultado é "falso" se uma linha igual for encontrada.

Como usual, os valores nulos nas expressões ou nas linhas da subconsulta são combinados conforme as regras normais do SQL para expressões booleanas. Duas linhas são consideradas iguais se todos os membros correspondentes forem iguais e não nulos; as linhas não são iguais se algum membro correspondente for diferente e não nulo; caso contrário, o resultado da comparação da linha é desconhecido (nulo). Se todos os resultados das linhas forem diferentes ou nulos, com pelo menos um nulo, então o resultado de NOT IN é nulo.

ANY/SOME

expressão operador ANY (subconsulta)
expressão operador SOME (subconsulta)

O lado direito desta forma do ANY é uma subconsulta entre parênteses, que deve retornar exatamente uma coluna. A expressão à esquerda é avaliada e comparada com cada linha do resultado da subconsulta utilizando o operador fornecido, que deve produzir um valor booleano como resultado. O resultado do ANY é "verdade" se qualquer resultado verdade for obtido. O resultado é "falso" se nenhum resultado verdade for obtido (incluindo o caso especial onde a subconsulta não retorna nenhuma linha).

SOME é um sinônimo de ANY. IN equivale ao = ANY.

Não havendo nenhum êxito, e pelo menos uma linha da direita produzir nulo para o resultado do operador, o resultado da construção ANY será nulo, e não falso. Isto está de acordo com as regras normais do SQL para as combinações booleanas de valores nulos.

Do mesmo modo que no EXISTS, não é sensato supor que a subconsulta será executada até o fim.

(expressão [, expressão ...]) operator ANY (subconsulta)
(expressão [, expressão ...]) operator SOME (subconsulta)

O lado direito desta forma do ANY é uma subconsulta entre parênteses, que deve retornar tantas colunas quantas forem as expressões existentes na lista do lado esquerdo. As expressões do lado esquerdo são avaliadas e comparadas com cada linha do resultado da subconsulta, utilizando o operador fornecido. Atualmente, somente os operadores = e <> são permitidos em consultas ANY linha por linha. O resultado do ANY é "verdade" se for encontrada alguma linha igual ou diferente, respectivamente. O resultado é "falso" se não for encontrada nenhuma linha deste tipo (incluindo o caso especial onde a subconsulta não retorna nenhuma linha).

Como usual, os valores nulos nas expressões ou nas linhas da subconsulta são combinados conforme as regras normais do SQL para expressões booleanas. Duas linhas são consideradas iguais se todos os membros correspondentes forem iguais e não nulos; as linhas não são iguais se algum membro correspondente for diferente e não nulo; caso contrário, o resultado da comparação da linha é desconhecido (nulo). Havendo pelo menos um resultado de linha nulo, então o resultado de ANY não poderá ser falso; será verdade ou nulo.

ALL

expressão operador ALL (subconsulta)

O lado direito desta forma do ALL é uma subconsulta entre parênteses, que deve retornar exatamente uma coluna. A expressão à esquerda é avaliada e comparada com cada linha do resultado da subconsulta utilizando o operador fornecido, que deve produzir um valor booleano como resultado. O resultado do ALL é "verdade" se o resultado para todas as linhas for verdade (incluindo o caso especial onde a subconsulta não retorna nenhuma linha). O resultado é "falso" se algum resultado falso for encontrado.

NOT IN equivale ao <> ALL.

Não havendo nenhuma falha, mas pelo menos uma linha da direita produzir nulo para o resultado do operador, o resultado da construção ALL será nulo, e não verdade. Isto está de acordo com as regras

normais do SQL para as combinações booleanas de valores nulos.

Do mesmo modo que no EXISTS, não é sensato supor que a subconsulta será executada até o fim.

(expressão [, expressão ...]) operador ALL (subconsulta)

O lado direito desta forma do ALL é uma subconsulta entre parênteses, que deve retornar tantas colunas quantas forem as expressões existentes na lista do lado esquerdo. As expressões do lado esquerdo são avaliadas e comparadas com cada linha do resultado da subconsulta, utilizando o operador fornecido. Atualmente, somente os operadores = e <> são permitidos em consultas ALL linha por 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 é "falso" se for encontrada alguma linha diferente ou igual, respectivamente.

Como usual, os valores nulos nas expressões ou nas linhas da subconsulta são combinados conforme as regras normais do SQL para expressões booleanas. Duas linhas são consideradas iguais se todos os membros correspondentes forem iguais e não nulos; as linhas não são iguais se algum membro correspondente for diferente e não nulo; caso contrário, o resultado da comparação da linha é desconhecido (nulo). Havendo pelo menos um resultado de linha nulo, então o resultado de ALL não poderá ser verdade; será falso ou nulo.

Comparação linha por linha

(expressão [, expressão ...]) operador (subconsulta)
(expressão [, expressão ...]) operador (expressão [, expressão ...])

O lado esquerdo é uma lista de expressões escalares. O lado direito pode ser tanto uma lista de expressões escalares do mesmo comprimento, uma uma subconsulta entre parênteses, que deve retornar tantas colunas quantas forem as expressões existentes na lista do lado esquerdo. Além disso, a subconsulta não pode retornar mais de uma linha (Se não retornar nenhuma linha, o resultado é considerado como sendo nulo). O lado esquerdo é avaliado e comparado linha por linha com a única linha do resultado da subconsulta, ou com a lista de expressões à direita. Atualmente, somente os operadores = e <> são permitidos em comparação linha por linha. O resultado será "verdade" se as duas linhas forem iguais ou diferentes, respectivamente.

Como usual, os valores nulos nas expressões ou nas linhas da subconsulta são combinados conforme as regras normais do SQL para expressões booleanas. Duas linhas são consideradas iguais se todos os membros correspondentes forem iguais e não nulos; as linhas não são iguais se algum membro correspondente for diferente e não nulo; caso contrário, o resultado da comparação da linha é desconhecido (nulo).

CAPÍTULO 10. Conversão de tipo

Uma consulta SQL pode, intencionalmente ou não, requerer a utilização de tipos de dado diferentes na mesma expressão. O PostgreSQL possui um grande número de funcionalidades para avaliar expressões mistas.

Em muitos casos o usuário não precisará compreender os detalhes do mecanismo de conversão de tipo. Entretanto, as conversões implícitas feitas pelo PostgreSQL podem afetar o resultado da consulta. Quando for necessário, estes resultados podem ser adequados pelo usuário ou pelo programador utilizando a conversão explícita de tipo.

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

O Guia do Programador do PostgreSQL contém mais detalhes relativos aos algoritmos utilizados para conversão implícita e explícita de tipo.

Visão geral

O 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 que outras implementações do SQL. Por isso, o comportamento usual para conversão de tipo no PostgreSQL deve ser governado por regras gerais em vez de heurísticas ad hoc [12]. para permitir expressões com tipos mistos terem significado inclusive com tipos definidos pelo usuário.

A leitor/analisador do PostgreSQL decodifica os elementos léxicos em apenas cinco categorias fundamentais: inteiros, números de ponto flutuante, cadeias de caracteres, nomes e palavras chave. Os tipos estendidos são inicialmente colocados em cadeias de caracteres, em sua maioria. A definição da linguagem SQL permite especificar nomes de tipos em cadeias de caracteres, e este mecanismo pode ser utilizado pelo PostgreSQL para colocar o analisador na direção do caminho correto. Por exemplo, a consulta

```
tgl=> SELECT text 'Origin' AS "Label", point '(0,0)' AS "Value";
 Label  | Value
--------+-------
 Origin | (0,0)
(1 row)
```

possui duas constantes literais, dos tipos text e point. Se o tipo não for especificado para o literal cadeia de caracteres, o tipo polivalente unknown é atribuído inicialmente, para ser resolvido posteriormente nos estágios descritos abaixo.

12 ad hoc: para isso, para esse caso - Novo Dicionário Aurélio da Língua Portuguesa. (N.T.)

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

Operadores

O PostgreSQL permite expressões com operadores unários (um só argumento) de prefixo e sufixo, assim como operadores binários (dois argumentos).

Chamadas de função

Grande parte do sistema de tipo do PostgreSQL é construído em torno de um amplo conjunto de funções. As chamadas de função podem possuir um ou mais argumentos os quais, para uma determinada consulta, devem ser correspondidos por uma das funções disponíveis no catálogo do sistema. Uma vez que o PostgreSQL permite a sobrecarga de funções, apenas o nome da função não identifica unicamente a função a ser chamada; o analisador deve selecionar a função correta baseado nos tipos de dado dos argumentos fornecidos.

Destino dos comandos

Os comandos SQL INSERT e UPDATE colocam os resultados das expressões em tabelas. As expressões no comando devem corresponder, ou talvez serem convertidas, nos tipos de dado das colunas de destino.

Construções UNION e CASE

Uma vez que os resultados de todos os comandos SELECT da união devem retornar em um único conjunto de colunas, deve haver correspondência entre os tipos de dado do resultado de cada uma das cláusulas SELECT, para se tornar um conjunto uniforme. Do mesmo modo, os resultados das expressões da construção CASE devem ser todos convertidos no mesmo tipo de dado para a expressão CASE ter, como um todo, um tipo de dado de saída conhecido.

Grande parte das regras gerais para conversão de tipo de dado utilizam convenções simples, baseadas nas funções do PostgreSQL e tabelas de operadores do sistema. Existe alguma heurística [13] incluída nas regras de conversão para fornecer um melhor apoio às convenções dos tipos nativos do padrão SQL como smallint, integer e real.

Os catálogos do sistema armazenam informações sobre quais conversões, chamadas de casts, entre os tipos de dado são válidas, e como realizar estas conversões. Outras conversões podem ser adicionadas pelo usuário utilizando o comando CREATE CAST (Geralmente isto é feito juntamente com a definição de novos tipos de dado. O conjunto de conversões entre tipos nativos foi cuidadosamente elaborado, não devendo ser modificado).

Uma heurística adicional é incluída no analisador 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 definido pelo usuário.

13 heurística: conjunto de regras e métodos que conduzem à descoberta, à invenção e à resolução de problemas - Novo Dicionário Aurélio da Língua Portuguesa. (N.T.)

Cada uma das categorias, com exceção da definida pelo usuário, possui um tipo preferido que é preferencialmente selecionado quando há ambigüidade. Na categoria definido pelo usuário, cada tipo é o seu próprio tipo preferido. As expressões ambíguas (as que possuem muitas soluções candidatas na análise) podem, geralmente, serem resolvidas quando há vários tipos nativos possíveis, mas ocasionam erro quando existem várias escolhas para tipos definidos pelo usuário.

Todas as regras de conversão são elaboradas 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 possua nenhum conhecimento a priori, devem vir "acima" na hierarquia de tipo. Nas expressões com tipos mistos, os tipos nativos devem sempre ser convertidos para o tipo definido pelo usuário (obviamente, apenas se a conversão for necessária).

Tipos definidos pelo usuário não são relacionados. Atualmente o PostgreSQL não dispõe de informações relativas ao relacionamento entre tipos, além das heurísticas codificadas para os tipos nativos e relacionamentos implícitos baseado nas funções disponíveis no catálogo.

Não deve haver nenhum trabalho adicional do analisador ou do executor se o comando não necessitar de conversão implícita de tipo, ou seja, se o comando estiver bem formulado e os tipos já se correspondem, então o comando deve prosseguir sem perda de tempo do analisador e sem introduzir funções de conversão implícitas desnecessárias no comando.

Além disso, se o comando geralmente requer uma função implícita para conversão de tipo, e o usuário definir uma função explícita com argumentos do tipo correto, o analisador deve usar esta nova função, não fazendo mais a conversão implícita utilizando a função antiga.

Operadores

Os tipos dos operandos em chamadas de operador são resolvidos utilizando o procedimento descrito abaixo. Observe que este procedimento é indiretamente afetado pela precedência dos operadores envolvidos.

Determinação do tipo do operando

Selecionar no catálogo do sistema pg_operator os operadores a ser considerados. Se um nome não qualificado de operador for utilizado (o caso usual), os operadores considerados são aqueles com o mesmo nome e número de argumentos visíveis no caminho de procura corrente. Se um nome qualificado de operador for fornecido, somente os operadores no esquema especificado são considerados.

Se no caminho de procura forem encontrados vários operadores com argumentos do mesmo tipo, somente o que aparece primeiro no caminho é considerado. Mas os operadores com argumentos de tipos diferentes são levados em consideração com a mesma precedência, não importando a posição no caminho de procura.

Verificar se algum operador aceita os mesmos tipos de dado dos argumentos de entrada. Caso exista

(só pode haver uma correspondência exata no conjunto de operadores considerados), este é usado.

Se um dos argumentos de um operador binário for do tipo unknown (desconhecido), então pressupõe-se possuir o mesmo tipo do outro argumento nesta verificação. Outros casos envolvendo o tipo unknown nunca encontram correspondência nesta etapa.

Procurar pela melhor correspondência.

Rejeitar os operadores candidatos para os quais os tipos da entrada não correspondem e não podem ser forçados (utilizando uma função implícita de conversão) a corresponder. Pressupõe-se que os literais do tipo unknown podem ser convertidos em qualquer outro tipo para esta finalidade. Se apenas um operador candidato permanecer, então este é usado; senão continuar na próxima etapa.

Examinar todos os operadores candidatos, e manter aqueles com mais correspondências exatas com os tipos da entrada. Manter todos os candidatos se nenhum possuir alguma correspondência exata. Se apenas um candidato permanecer, este é usado; senão continuar na próxima etapa.

Examinar todos os operadores candidatos, e manter aqueles com mais correspondências exatas ou correspondências compatíveis binariamente com os tipos da entrada. Manter todos os candidatos se nenhum possuir correspondências exatas ou binariamente compatíveis. Se apenas um operador candidato permanecer, este é usado; senão continuar na próxima etapa.

Examinar todos os operadores candidatos, e manter aqueles que aceitam os tipos preferidos em mais posições onde a conversão de tipo será necessária. Manter todos os candidatos se nenhum aceitar os tipos preferidos. Se apenas um operador candidato permanecer, este é usado; senão continuar na próxima etapa.

Se algum dos argumentos de entrada for do tipo "unknown", verificar as categorias de tipo aceitas nesta posição do argumento pelos candidatos remanescentes. Em cada posição, selecionar a categoria string se qualquer um dos candidatos aceitar esta categoria (isto faz a cadeia de caracteres parecer apropriada porque todo literal de tipo desconhecido se parece com uma cadeia de caracteres). Senão, se todos os candidatos remanescentes aceitam a mesma categoria de tipo, selecionar esta categoria; senão falha porque a escolha correta não pode ser deduzida sem mais informações. Também observa se algum dos candidatos aceita um tipo de dado preferido dentro da categoria selecionada. Agora rejeita os operadores candidatos que não aceitam a categoria de tipo selecionada; além disso, se algum operador candidato aceitar o tipo preferido em uma dada posição do argumento, rejeitar os candidatos que aceitam tipos não preferidos para este argumento.

Se apenas um operador candidato permanecer, este é usado; Se nenhum candidato, ou mais de um candidato, permanecer, então falha.

Exemplos

Exemplo - Determinação do tipo em operador de exponenciação

Existe apenas um operador de exponenciação definido no catálogo, e recebe argumentos do tipo double precision. Inicialmente é atribuído o tipo integer para os dois argumentos desta expressão de consulta:

```
tgl=> SELECT 2 ^ 3 AS "Exp";
 Exp
-----
   8
(1 row)
```

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

```
tgl=> SELECT CAST(2 AS double precision) ^ CAST(3 AS double precision) AS "Exp";
 Exp
-----
   8
(1 row)
```

ou

```
tgl=> SELECT 2.0 ^ 3.0 AS "Exp";
 Exp
-----
   8
(1 row)
```

Nota: Esta última forma ocasiona menos trabalho adicional, porque nenhuma função é chamada para fazer a conversão implícita de tipo. Isto não causa problema em consultas pequenas, mas pode ter impacto no desempenho das consultas envolvendo tabelas grandes.

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

Uma sintaxe estilo cadeia de caracteres é utilizada para trabalhar com tipos string, assim como para trabalhar com tipos estendidos complexos. Cadeias de caracteres de tipo não especificado são correspondidas por praticamente todos os operadores candidatos.

Um exemplo com um argumento não especificado:

```
tgl=> SELECT text 'abc' || 'def' AS "Texto e Desconhecido";
 Texto e Desconhecido
----------------------
 abcdef
(1 row)
```

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

Concatenação de tipos não especificados:

```
tgl=> SELECT 'abc' || 'def' as "Não Especificado";
```

```
 Não Especificado
------------------
 abcdef
(1 row)
```

Neste caso não existe nenhuma informação inicial do tipo a ser usado, porque nenhum tipo foi especificado na consulta. Portanto, o analisador procura todos os operadores candidatos e descobre que existem candidatos aceitando as categorias de entrada cadeia de caracteres e cadeia de bits. Uma vez que a categoria cadeia de caracteres é a preferida quando está disponível, então o "tipo preferido" para cadeias de caracteres, text, é utilizado como o tipo específico para resolver os literais desconhecidos.

Exemplo - Determinação do tipo em operador de valor absoluto e em fatorial

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 na categoria numérica. Portanto, o PostgreSQL usa esta entrada quando confrontado com uma entrada não numérica:

```
tgl=> select @ text '-4.5' as "abs";
 abs
-----
 4.5
(1 row)
```

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

```
tgl=> select @ text '-4.5e500' as "abs";
ERROR:  Input '-4.5e500' is out of range for float8
```

Por outro lado, o operador de sufixo ! (fatorial) é definido apenas para tipos de dado inteiros, e não para float8. Portanto, se tentarmos algo semelhante usando o !, resulta em:

```
tgl=> select text '20' ! as "fatorial";
ERROR:  Unable to identify a postfix operator '!' for type 'text'
        You may need to add parentheses or an explicit cast
```

Isto acontece porque o sistema não pode decidir qual dos vários operadores ! possíveis deve ser o preferido. Pode ser dada uma ajuda usando uma transformação explícita:

```
tgl=> select cast(text '20' as int8) ! as "fatorial";
     fatorial
---------------------
 2432902008176640000
(1 row)
```

Funções

Os tipos dos argumentos das chamadas de função são resolvidos de acordo com as seguintes etapas.

Determinação do tipo em argumento de função

Selecionar no catálogo do sistema pg_proc as funções a ser consideradas. Se um nome não qualificado de função for utilizado, as funções consideradas são aquelas com o mesmo nome e número de argumentos visíveis no caminho de procura corrente. Se um nome qualificado de função for fornecido, somente as funções no esquema especificado são consideradas.

Se no caminho de procura forem encontradas várias funções com argumentos do mesmo tipo, somente a que aparece primeiro no caminho é considerada. Mas as funções com argumentos de tipos diferentes são consideradas com a mesma precedência, não importando a posição no caminho de procura.

Verificar se alguma função aceita os mesmos tipos de dado dos argumentos de entrada. Caso exista (só pode haver uma correspondência exata no conjunto de operadores examinados), esta é usada. Os casos envolvendo o tipo unknown nunca encontram correspondência nesta etapa.

Se nenhuma correspondência exata for encontrada, verificar se a chamada de função é parecida com uma requisição trivial de transformação de tipo. Isto acontece se a chamada da 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 nomeado. Quando estas condições são encontradas, o argumento da função é transformado no tipo de dado nomeado sem chamada de função explícita.

Procurar pela melhor correspondência.

Rejeitar as funções candidatas para as quais os tipos da entrada não correspondem e não podem ser forçados (utilizando uma função implícita de conversão) a corresponder. Pressupõe-se que os literais do tipo unknown podem ser convertidos em qualquer outro tipo para esta finalidade. Se apenas uma função candidata permanecer, então esta é usada; senão continuar na próxima etapa.

Examinar todas as funções candidatas, e manter aquelas com mais correspondências exatas com os tipos da entrada. Manter todas as candidatas se nenhuma possuir alguma correspondência exata. Se apenas uma função candidata permanecer, então esta é usada; senão continuar na próxima etapa.

Examinar todas as funções candidatas, e manter aquelas com mais correspondências exatas ou correspondências compatíveis binariamente com os tipos da entrada. Manter todas as candidatas se nenhuma possuir correspondências exatas ou binariamente compatíveis. Se apenas uma função candidata permanecer, então esta é usada; senão continuar na próxima etapa.

Examinar todas as funções candidatas, e manter aquelas que aceitam os tipos preferidos em mais posições onde a conversão de tipo será necessária. Manter todas as candidatas se nenhuma aceitar os tipos preferidos. Se apenas uma função candidata permanecer, então esta é usada; senão continuar na próxima etapa.

Se algum dos argumentos de entrada for do tipo "unknown", verificar as categorias de tipo aceitas nesta posição do argumento pelas candidatas remanescentes. Em cada posição, selecionar a categoria string se qualquer uma das candidatas aceitar esta categoria (isto faz a cadeia de caracteres parecer apropriada porque todo 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 falha porque a escolha correta não pode ser deduzida sem mais informações. Também observa se alguma das candidatas aceita um tipo de dado preferido dentro da categoria selecionada. Agora rejeita 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.

Se apenas uma função candidata permanecer, este é usada; Se nenhuma candidata, ou mais de uma candidata, permanecer, então falha.

Exemplos

Exemplo - Determinação do tipo em argumento de função de fatorial
Existe apenas uma função int4fac definida no catálogo pg_proc. Portanto, a consulta abaixo converte automaticamente o argumento do tipo int2 em int4:

```
tgl=> SELECT int4fac(int2 '4');
 int4fac
---------
      24
(1 row)
```

sendo na verdade transformada pelo analisador em

```
tgl=> SELECT int4fac(int4(int2 '4'));
 int4fac
---------
      24
(1 row)
```

Exemplo - Determinação do tipo em função de cadeia de caracteres
Existem duas funções substr declaradas em pg_proc. Entretanto, apenas uma recebe dois argumentos, dos tipos text e int4.
Se for chamada com uma constante cadeia de caracteres de um tipo não especificado, o tipo é convertido diretamente no tipo da única função candidata:

```
tgl=> SELECT substr('1234', 3);
 substr
--------
     34
(1 row)
```

Se a cadeia de caracteres for declarada como do tipo varchar, podendo ser o caso se for proveniente

de uma tabela, então o analisador tentará torná-la text:

```
tgl=> SELECT substr(varchar '1234', 3);
 substr
--------
    34
(1 row)
```

que é transformada pelo analisador se tornando

```
tgl=> SELECT substr(text(varchar '1234'), 3);
 substr
--------
    34
(1 row)
```

Nota: Na verdade, o analisador está ciente que os tipos text e varchar são binariamente compatíveis, significando que uma pode ser passada para a função que aceita a outra sem realizar nenhuma conversão física. Portanto, nenhuma chamada para conversão explícita de tipo é na verdade inserida neste caso.

e, se a função for chamada com o tipo int4, o analisador tenta convertê-lo para o tipo text:

```
tgl=> SELECT substr(1234, 3);
 substr
--------
    34
(1 row)
```

o que na verdade executa como

```
tgl=> SELECT substr(text(1234), 3);
 substr
--------
    34
(1 row)
```

tendo sucesso porque existe a função de conversão text(int4) no catálogo do sistema.

Destino dos comandos

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

Determinação do tipo em destino de comando

Verificar a correspondência exata com o destino.

Senão, tentar transformar a expressão no tipo de dado do destino. Isto será bem sucedido se os dois tipos forem sabidamente binariamente compatíveis, ou se houver uma função de conversão. Se a expressão for um literal de tipo desconhecido, o conteúdo da cadeia de caracteres literal será colocado na rotina de conversão de entrada para o tipo de destino.

Se o destino for de um tipo de comprimento fixo (por exemplo, char ou varchar declarado com comprimento) então tentar encontrar uma função de tamanho para o tipo de dado do destino. Uma função de tamanho é uma função com o mesmo nome do tipo recebendo dois argumentos, sendo que o primeiro é deste tipo e o segundo é um inteiro, e retorna o mesmo tipo. Se alguma for encontrada, esta é aplicada, passando o comprimento declarado da coluna como o segundo parâmetro.

Exemplo - Conversão de tipo de armazenamento character

Para uma coluna de destino declarada como character(20) a seguinte consulta garante que o destino terá o tamanho correto:

```
tgl=> CREATE TABLE vv (v character(20));
CREATE
tgl=> INSERT INTO vv SELECT 'abc' || 'def';
INSERT 392905 1
tgl=> SELECT v, length(v) FROM vv;
         v          | length
--------------------+--------
 abcdef             |     20
(1 row)
```

O que acontece realmente é que as duas literais desconhecidas são resolvidas como text por padrão, permitindo que o operador || seja resolvido como concatenação de text. Então o resultado text do operador é transformado em bpchar (caractere preenchido com brancos, ou "blank-padded char", que é o nome interno do tipo de dado caractere) para corresponder ao tipo da coluna de destino (Uma vez que o analisador sabe que text e bpchar são binariamente compatíveis, esta transformação é implícita e não insere na realidade nenhuma chamada de função). Finalmente, 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 os espaços de preenchimento.

Construções UNION e CASE

As construções UNION do SQL podem precisar fazer tipos possivelmente não similares se tornarem 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 similares do mesmo modo que UNION. A construção CASE também utiliza um algoritmo idêntico para fazer corresponder suas expressões componentes e selecionar o tipo de dado do resultado.

Determinação do tipo em UNION e CASE

Se todas as entradas forem do tipo unknown, resolve como sendo do tipo text (o tipo preferido para

a categoria cadeia de caracteres). Senão, ignorar as entradas unknown ao escolher o tipo.

Se as entradas não desconhecidas não são todas da mesma categoria de tipo, falhar.

Escolher o tipo da primeira entrada não desconhecida que for o tipo preferido nesta categoria, ou que permita todas as entradas não desconhecidas serem implicitamente transformadas neste.

Transformar todas as entradas no tipo selecionado.

Exemplos

Exemplo -. Tipos subespecificados em uma união

```
tgl=> SELECT text 'a' AS "Text" UNION SELECT 'b';
 Text
------
 a
 b
(2 rows)
```

Neste caso, o literal de tipo desconhecido 'b' é resolvido como o tipo text.
Exemplo 7-8. Conversão de tipo em uma união simples

```
tgl=> SELECT 1.2 AS "Numeric" UNION SELECT 1;
 Numeric
---------
       1
     1.2
(2 rows)
```

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

Exemplo 7-9. Conversão de tipo em uma união transposta

```
tgl=> SELECT 1 AS "Real"
tgl-> UNION SELECT CAST('2.2' AS REAL);
 Real
------
    1
  2.2
(2 rows)
```

Aqui, porque o tipo real não pode ser implicitamente transformado em integer, mas o integer pode ser implicitamente transformado em real, o tipo do resultado da união é resolvido como real.

CAPÍTULO 11. Índices

Habitualmente, os índices são utilizados para melhorar o desempenho dos bancos de dados. Um índice permite ao servidor de banco de dados encontrar e trazer linhas específicas muito mais rápido do que faria sem o índice. Mas os índices também produzem trabalho adicional para o sistema de banco de dados como um todo devendo, portanto, serem utilizados conscientemente.

Introdução

O exemplo clássico da necessidade de um índice é a existência de uma tabela semelhante a esta:

```
CREATE TABLE teste1 (  id integer,  conteudo varchar );
```

com a aplicação executando muitas consultas da forma:

```
SELECT conteudo FROM teste1 WHERE id = constante;
```

Normalmente, o sistema seria obrigado a percorrer a tabela teste1, linha por linha, para encontrar todas as entradas correspondentes. Se existirem muitas linhas em teste1, e somente poucas linhas (possivelmente uma ou nenhuma) retornadas pela consulta, então este método é claramente ineficiente. Se o sistema fosse instruído para manter um índice para a coluna id, então poderia ser utilizado um método mais eficiente para localizar as linhas correspondentes. Por exemplo, só precisaria percorrer uns poucos níveis dentro da árvore de procura.

Uma abordagem semelhante é utilizada pela maioria dos livros, fora os de ficção: Os termos e os conceitos procurados freqüentemente pelos leitores são reunidos em um índice alfabético colocado no final do livro. O leitor interessado pode percorrer o índice rapidamente e ir direto para a página desejada, sem ter que ler o livro por inteiro em busca do local onde está interessado. Assim como é uma tarefa do autor prever os itens que os leitores mais provavelmente vão procurar, é uma tarefa do programador de banco de dados prever quais índices trarão benefícios.

O comando mostrado abaixo pode ser utilizado para criar o índice na coluna id conforme foi discutido:

```
CREATE INDEX idx_teste1_id ON teste1 (id);
```

O nome idx_teste1_id pode ser escolhido livremente, mas deve ser usado algo que permita lembrar mais tarde para que serve o índice.

Para remover um índice é utilizado o comando DROP INDEX. Os índices podem ser adicionados ou removidos das tabelas a qualquer instante.

Após o índice ser criado não é necessária mais nenhuma intervenção adicional: o sistema passa a utilizar o índice quando julgar mais eficiente do que a procura seqüencial na tabela. Porém, é

necessário executar o comando ANALYZE regularmente para atualizar as estatísticas que permitem ao planejador de consultas tomar melhores decisões. Também deve ser lido o Capítulo 10 para obter informações sobre como descobrir se o índice está sendo utilizado, e quando e porque o planejador pode decidir não utilizar um índice.

Os índices podem beneficiar as atualizações (UPDATE) e as exclusões (DELETE) com condição de procura. Os índices também podem ser utilizados em consultas com junção. Portanto, um índice definido em uma coluna que faça parte da condição de junção pode acelerar, significativamente, a consulta.

Quando um índice é criado, o sistema precisa mantê-lo sincronizado com a tabela. Isto gera um trabalho adicional para as operações de manipulação de dados. Portanto, os índices não essenciais ou não utilizados devem ser removidos. Observe que uma consulta ou um comando de manipulação de dados pode utilizar no máximo um índice por tabela.

Tipos de índice

O PostgreSQL disponibiliza vários tipos de índice: B-tree (árvore B), R-tree (árvore R), GiST e Hash. Cada tipo de índice é mais apropriado para um determinado tipo de consulta devido ao algoritmo utilizado. Por padrão, o comando CREATE INDEX cria um índice B-tree, adequado para as situações mais comuns. Em particular, o otimizador de consultas do PostgreSQL levará em consideração utilizar um índice B-tree sempre que uma coluna indexada estiver envolvida numa comparação utilizando um dos seguintes operadores: <, <=, =, >=, >

Os índices R-tree são especialmente indicados para dados espaciais. Para criar um índice R-tree, deve ser utilizado um comando da forma:

CREATE INDEX nome ON tabela USING RTREE (coluna);

O otimizador de consultas do PostgreSQL levará em consideração utilizar um índice R-tree sempre que uma coluna indexada estiver envolvida numa comparação utilizando um dos seguintes operadores: <<, &<, &>, >>, @, ~=, && .

O otimizador de consultas do PostgreSQL levará em consideração utilizar um índice hash sempre que uma coluna indexada estiver envolvida numa comparação utilizando o operador=. O seguinte comando pode ser utilizado para criar um índice hash:

CREATE INDEX nome ON tabela USING HASH (coluna);

Nota: Os testes mostram que os índices hash do PostgreSQL têm desempenho semelhante ou mais lento que os índices B-tree, e que o tamanho e o tempo de construção dos índices hash são muito piores. Os índices hash também possuem um fraco desempenho sob alta concorrência. Por estas razões, a utilização dos índices hash é desestimulada.

O índice B-tree é uma implementação das árvores B de alta concorrência de Lehman-Yao. O método do índice R-tree implementa as árvores R utilizando o algoritmo de partição quadrática de Guttman. O índice hash é uma uma implementação das dispersões lineares de Litwin. São mencionados os algoritmos utilizados somente para indicar que todos estes métodos de acesso são

inteiramente dinâmicos, não necessitando de otimização periódica (como no caso de, por exemplo, métodos de acesso hash estáticos).

Índices com várias colunas

Um índice pode ser definido envolvendo mais de uma coluna. Por exemplo, se existir uma tabela da forma

CREATE TABLE teste2 (maior int, menor int, nome varchar);

(Digamos que seja mantido o diretório /dev no banco de dados...) e seja feita freqüentemente uma consulta do tipo

SELECT nome FROM teste2 WHERE maior = constante AND menor = constante;

então é apropriada a definição de um índice envolvendo as colunas maior e menor. Por exemplo,

CREATE INDEX idx_teste2_maior_menor ON teste2 (maior, menor);

Atualmente, somente as implementações de B-tree e GiST suportam índices com várias colunas. Até 32 colunas podem ser especificadas (Este limite pode ser alterado durante a geração do PostgreSQL; consulte o arquivo pg_config.h).

O otimizador de consultas pode utilizar um índice com várias colunas, para consultas que envolvam as primeiras n colunas consecutivas do índice (quando utilizado com os operadores apropriados, até o número total de colunas especificadas na definição do índice. Por exemplo, um índice incluindo (a, b, c) pode ser utilizado em consultas envolvendo a, b e c, ou em consultas envolvendo a e b, ou em consultas envolvendo apenas a, mas não em outras combinações. (Em uma consulta envolvendo a e c, o otimizador pode decidir utilizar um índice para a apenas, tratando c como uma coluna comum não indexada).

Os índices com várias colunas só podem ser utilizados se as cláusulas envolvendo as colunas indexadas forem unidas por AND. Por exemplo,

SELECT nome FROM teste2 WHERE maior = constante OR menor = constante;

não pode fazer uso do índice idx_teste2_maior_menor definido acima para procurar as duas colunas (Entretanto, pode ser utilizado para procurar a coluna maior apenas).

Os índices com várias colunas devem ser usados com moderação. Na maioria das vezes, um índice em apenas uma coluna é suficiente, economizando espaço e tempo. Um índice com mais de três colunas é quase certo não ser apropriado.

Índices únicos

Os índices também podem ser utilizados para garantir a unicidade do valor de uma coluna, ou a unicidade dos valores combinados de mais de uma coluna.

CREATE UNIQUE INDEX nome ON tabela (coluna [, ...]);

Atualmente, somente os índices B-tree poder ser declarados como únicos.

Quando um índice é declarado como único, não é permitida a existência de mais de uma linha da tabela com o mesmo valor do índice. Os valores nulos não são considerados como sendo iguais.

O PostgreSQL cria, automaticamente, índices únicos quando a tabela é declarada com restrição de unicidade ou com chave primária, nas colunas que compõem a chave primária ou nas colunas com unicidade (um índice com várias colunas, quando apropriado), para garantir esta restrição. Um índice único pode ser adicionado à tabela posteriormente para adicionar uma restrição de unicidade.

Nota: A forma preferida para adicionar restrição de unicidade a uma tabela é por meio do comando ALTER TABLE ... ADD CONSTRAINT. A utilização de índices para garantir a restrição de unicidade pode ser considerada um detalhe de implementação que não deve ser acessado diretamente.

Índices funcionais

Para um índice funcional, um índice é definido sobre o resultado de uma função aplicada a uma ou mais colunas de uma única tabela. Os índices funcionais podem ser utilizados para obter acesso rápido aos dados baseado no resultado da chamada de uma função.

Por exemplo, uma forma de fazer comparações que não sejam sensíveis a letras maiúsculas e minúsculas é utilizar a função lower:

SELECT * FROM teste1 WHERE lower(col1) = 'valor';

Esta consulta pode utilizar um índice, caso algum tenha sido definido sobre o resultado da operação lower(coluna):

CREATE INDEX idx_teste1_lower_col1 ON teste1 (lower(col1));

A função na definição do índice pode receber mais de um argumento, mas devem ser colunas da tabela, e não constantes. Os índices funcionais são sempre de uma coluna (ou seja, o resultado da função), mesmo que a função utilize mais de um parâmetro de entrada; não pode haver índice de várias colunas contendo chamada de função.

Dica: As restrições mencionadas no parágrafo anterior podem ser facilmente contornadas, definindo uma função personalizada utilizando a definição do índice para obter o resultado desejado internamente.

Classes de operador

A definição de um índice pode especificar uma classe de operador para cada coluna do índice.

CREATE INDEX nome ON tabela (coluna opclass [, ...]);

A classe de operador identifica os operadores a ser utilizados pelo índice para esta coluna. Por exemplo, um índice B-tree em inteiros de quatro bytes utiliza a classe int4_ops; esta classe de operador inclui funções de comparação para inteiros de quatro bytes. Na prática a classe de operador padrão para o tipo de dado da coluna é normalmente suficiente. O ponto principal de existir classe de operador é que, para alguns tipos de dado, pode haver mais de uma ordenação que faça sentido. Por exemplo, pode ser desejado ordenar o tipo de dado do número complexo tanto pelo valor absoluto quanto pela parte real. Isto pode ser feito definindo duas classes de operador para o tipo de dado e, então, selecionando a classe apropriada ao definir o índice. Existem, também, algumas classes de operador com finalidades especiais:

As duas classes de operador box_ops e bigbox_ops suportam índices R-tree no tipo de dado box. A diferença entre estas duas é que bigbox_ops reduz as coordenadas da caixa, para evitar excessões de ponto flutuante ao realizar multiplicação, adição e subtração de coordenadas muito grandes de ponto flutuante. Se o campo no qual os retângulos se encontram tiver aproximadamente 20 000 unidades quadradas ou mais, deve ser utilizado bigbox_ops.

A seguinte consulta exibe todas as classes de operador definidas:

```
SELECT am.amname AS acc_method,
    opc.opcname AS ops_name
  FROM pg_am am, pg_opclass opc
  WHERE opc.opcamid = am.oid
  ORDER BY acc_method, ops_name;
```

Podendo ser estendida para mostrar todos os operadores incluídos em cada classe:

```
SELECT am.amname AS acc_method,
    opc.opcname AS ops_name,
    opr.oprname AS ops_comp
  FROM pg_am am, pg_opclass opc, pg_amop amop, pg_operator opr
  WHERE opc.opcamid = am.oid AND
      amop.amopclaid = opc.oid AND
      amop.amopopr = opr.oid
  ORDER BY acc_method, ops_name, ops_comp;
```

Índices parciais

Um índice parcial é um índice construído sobre um subconjunto da tabela; o subconjunto é definido por uma expressão condicional (chamada de predicado do índice parcial). O índice contém entradas apenas para as linhas da tabela que satisfazem o predicado.

A principal motivação para os índices parciais é evitar a indexação de valores comuns. Uma vez que uma pesquisa procurando por um valor comum (um que apareça em mais do que uma pequena percentagem das linhas da tabela) não vai utilizar o índice de qualquer forma, não faz sentido em manter estas linhas no índice. Isto reduz o tamanho do índice, acelerando as consultas que utilizam este índice. Também acelera muitas operações de atualização da tabela, porque o índice não precisa ser atualizado em todos os casos. O Exemplo 8-1 mostra uma aplicação possível desta idéia.

Exemplo 8-1. Definindo um índice parcial para excluir valores comuns

Supondo que as informações relativas ao acesso ao servidor Web estejam sendo armazenadas no banco de dados, e que a maioria dos acessos se origina na faixa de endereços de IP da própria organização, mas alguns são de fora (digamos, empregados com acesso discado). Se a procura por endereços de IP for principalmente relativa ao acesso externo, provavelmente não será necessário indexar a faixa de endereços de IP correspondente à subrede da própria organização.

Assumindo que exista uma tabela igual a esta:

```
CREATE TABLE access_log (
    url      varchar,
    client_ip inet,
    ...
);
```

Para criar um índice parcial adequado para o exemplo acima, deve ser utilizado um comando como este:

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)
    WHERE NOT (client_ip > inet '192.168.100.0' AND client_ip < inet '192.168.100.255');
```

Uma consulta típica que pode utilizar este índice é:

```
SELECT * FROM access_log WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

Uma consulta que não pode utilizar este índice é:

```
SELECT * FROM access_log WHERE client_ip = inet '192.168.100.23';
```

Observe que este tipo de índice parcial requer que os valores comuns sejam conhecidos a priori. Se a distribuição dos valores for inerente (devido à natureza da aplicação) e estática (não muda com o tempo) isto não é difícil, mas se os valores comuns forem meramente devidos a carga coincidente dos dados isto pode requerer bastante trabalho para manutenção.

Outra possibilidade é excluir os valores do índice para os quais o perfil típico das consultas não esteja interessado; isto está mostrado no Exemplo 8-2. Isto resulta nas mesmas vantagens descritas acima, mas impede o acesso aos valores "que não interessam" por meio deste índice, mesmo que a procura pelo índice seja vantajosa neste caso. Obviamente, definir índice parcial para este tipo de cenário requer muito cuidado e verificação experimental.

Exemplo 8-2. Definindo um índice parcial para excluir valores que não interessam

Se existir uma tabela contendo tanto pedidos faturados quanto não faturados, onde os pedidos não faturados ocupam uma pequena parte da tabela, mas que são as linhas mais acessadas, é possível melhorar o desempenho criando um índice somente para os pedidos não faturados. O comando para criar o índice deve ficar parecido com este:

```
CREATE INDEX idx_pedidos_nao_faturados ON pedidos (num_pedido)
   WHERE faturado is not true;
```

Uma possível consulta utilizando este índice seria

```
SELECT * FROM pedidos WHERE faturado is not true AND num_pedido < 10000;
```

Entretanto, o índice também pode ser utilizado em consultas não envolvendo num_pedido como, por exemplo,

```
SELECT * FROM pedidos WHERE faturado is not true AND valor > 5000.00;
```

Embora não seja tão eficiente quanto seria um índice parcial englobando a coluna valor, porque o sistema precisa varrer o índice por inteiro, mesmo assim, havendo poucos pedidos não faturados, pode ser vantajoso utilizar este índice parcial para localizar apenas os pedidos não faturados.

Observe que a consulta abaixo não pode utilizar este índice:

```
SELECT * FROM pedidos WHERE num_pedido = 3501;
```

O pedido número 3501 pode estar entre os pedidos faturados e os não faturados.

O Exemplo 8-2 também ilustra que a coluna indexada e a coluna utilizada no predicado não precisam corresponder. O PostgreSQL suporta índice parcial com predicados arbitrários, desde que somente colunas da tabela sendo indexada estejam envolvidas. Entretanto, deve-se ter em mente que o predicado deve corresponder às condições utilizadas nas consultas que supostamente vão ser beneficiadas pelo índice. Sendo preciso, o índice parcial pode ser utilizado em uma consulta somente se o sistema puder reconhecer que a condição WHERE da consulta implica matematicamente no predicado do índice. O PostgreSQL não possui um provador de teoremas sofisticado que possa reconhecer predicados matematicamente equivalentes escritos de formas diferentes (Não apenas este provador geral de teoremas seria extremamente difícil de ser criado, como provavelmente também seria muito lento para poder ser usado na prática). O sistema pode reconhecer implicações de desigualdades simples como, por exemplo, "x < 1" implica "x < 2"; senão, a condição do predicado deve corresponder exatamente à condição WHERE da consulta, ou o índice não será reconhecido como utilizável.

Um terceiro uso possível para índices parciais não requer que o índice seja utilizado em nenhuma consulta. A idéia é criar um índice único sobre um subconjunto da tabela, como no Exemplo 8-3, garantindo a unicidade entre as linhas que satisfazem o predicado do índice, sem restringir as que não fazem parte.

Exemplo: Definindo um índice único parcial

Supondo que exista uma tabela descrevendo resultados de testes. Deseja-se garantir que exista apenas uma entrada "correta" para uma dada combinação de assunto e objetivo, mas que possa haver qualquer número de entradas "incorretas". Abaixo está mostrado um modo de fazer isto:

```
CREATE TABLE testes (assunto  text,
```

```
            objetivo text,
            correto  bool,
           ...);
CREATE UNIQUE INDEX testes_correto_constraint ON testes (assunto, objetivo)
   WHERE correto;
```

Esta é uma forma particularmente eficiente a ser utilizada quando existem poucas situações corretas e muitas incorretas.

Finalizando, um índice parcial também pode ser utilizado para prevalecer sobre a escolha do plano feito para a consulta pelo sistema. Pode ocorrer que conjuntos de dados com uma distribuição peculiar façam o sistema utilizar um índice quando na realidade não deveria. Neste caso, o índice pode ser definido de tal modo que não esteja disponível para a consulta com problema. Normalmente, o PostgreSQL realiza escolhas razoáveis relativas à utilização dos índices (por exemplo, evita-os ao buscar valores com muita ocorrência, de tal forma que o primeiro exemplo realmente economiza apenas o tamanho do índice, não sendo requerido para evitar a utilização do índice), e a escolha de planos grosseiramente incorretos é motivo para um relatório de erro.

Tenha em mente que a criação de um índice parcial indica que você sabe pelo menos tanto quanto o planejador de consultas sabe, particularmente você sabe quando um índice pode ser vantajoso. A formação deste conhecimento requer experiência e compreensão sobre como os índice no PostgreSQL funcionam. Na maioria dos casos, a vantagem de um índice parcial sobre um índice regular não é muita.

Mais informações relativas aos índices parciais podem ser obtidas em The case for partial indexes, Partial indexing in POSTGRES: research project e Generalized Partial Indexes.

Examinando a utilização do índice

Embora os índices no PostgreSQL não necessitem de manutenção e ajuste, ainda assim é importante verificar quais índices são realmente utilizados no trabalho diário. O exame da utilização dos índices é feito por meio do comando EXPLAIN;.

É difícil formular um procedimento geral para determinar quais índices devem ser criados. Existem vários casos típicos que foram mostrados através de exemplos nas seções anteriores. Muita verificação experimental é necessária na maioria dos casos. O restante desta seção dá algumas dicas para atingir este objetivo.

O comando ANALYZE sempre deve ser executado antes. Este comando coleta estatísticas relativas à distribuição dos valores na tabela. Esta informação é requerida para prever o número de linhas retornadas pela consulta, uma necessidade do planejador para atribuir custos efetivos a cada um dos planos possíveis para a consulta. Na ausência de uma estatística real, alguns valores padrão são pressupostos, sendo geralmente inadequados. O exame da utilização do índice pela aplicação sem a execução prévia do comando ANALYZE é, portanto, uma causa perdida.

Devem ser usados dados reais para a verificação experimental. O uso de dados de teste para criar índices diz quais são os índices necessários para os dados de teste, e nada além disso.

É particularmente fatal utilizar conjuntos de dados proporcionalmente reduzidos. Enquanto a seleção de 1.000 para cada 100.000 linhas pode ajudar a produzir um candidato a índice, a seleção de 1 para cada 100 linhas dificilmente ajudará, porque as 100 linhas provavelmente cabem dentro de uma página do disco, não havendo nenhum plano melhor que uma busca seqüencial em uma única página do disco.

Também deve ser tomado cuidado ao produzir os dados de teste, geralmente não disponíveis quando a aplicação ainda não está em produção. Valores muito semelhantes, completamente aleatórios, ou inseridos ordenadamente, distorcem as estatísticas em relação à distribuição que os dados reais teriam.

Quando os índices não são usados, pode ser útil como teste forçar sua utilização. Existem parâmetros em tempo de execução que podem desativar vários tipos de planos (descritos no Guia do Administrador do PostgreSQL). Por exemplo, desativar varreduras seqüenciais (enable_seqscan) e junções de laço-aninhado (nested-loop joins) (enable_nestloop), que são os planos mais básicos, forçam o sistema a utilizar um plano diferente. Se o sistema ainda escolher a varredura seqüencial ou a junção de laço-aninhado então existe, provavelmente, algum problema mais fundamental devido ao qual o índice não está sendo utilizado como, por exemplo, a condição da consulta não corresponder ao índice (Qual tipo de consulta pode utilizar qual tipo de índice é explicado na seção anterior).

Se forçar a utilização do índice não faz o índice ser usado, então existem duas possibilidades: Ou o sistema está correto e a utilização do índice não é apropriada, ou a estimativa de custo dos planos da consulta não estão refletindo a realidade. Portanto, deve ser medido o tempo da consulta com e sem índices. O comando EXPLAIN ANALYZE pode ser útil neste caso.

Se for descoberto que as estimativas de custo estão erradas existem, novamente, duas possibilidades. O custo total é calculado a partir do custo por linha de cada nó do plano vezes a seletividade estimada de cada nó do plano. Os custos dos nós do plano podem ser ajustados usando parâmetros em tempo de execução (descritos no Guia do Administrador do PostgreSQL). A estimativa imprecisa da seletividade é devida a falta de estatísticas. É possível melhorar esta situação ajustando os parâmetros de captura de estatísticas (veja o comando ALTER TABLE).

Se não for obtido sucesso no ajuste dos custos para ficarem mais apropriados, então pode ser necessário obrigar a utilização do índice explicitamente. Pode-se, também, desejar fazer contato com os desenvolvedores do PostgreSQL para examinar esta questão.

CAPÍTULO 12. Controle de concorrência

Este capítulo descreve o comportamento do sistema gerenciador de banco de dados PostgreSQL, quando duas ou mais sessões tentam acessar os mesmos dados ao mesmo tempo. O objetivo nesta situação é permitir o acesso eficiente para todas as sessões mantendo, ao mesmo tempo, uma rigorosa integridade dos dados. Todos os desenvolvedores de aplicação de banco de dados devem estar familiarizados com os tópicos cobertos por este capítulo.

Introdução

Diferentemente dos sistemas gerenciadores de banco de dados tradicionais, que usam bloqueios para realizar o controle de concorrência, o PostgreSQL mantém a consistência dos dados utilizando o modelo multiversão (Multiversion Concurrency Control, MVCC). Isto significa que ao consultar o banco de dados, cada transação enxerga um instantâneo (snapshot) dos dados (uma versão do banco de dados) conforme estes dados eram há algum tempo atrás, sem levar em consideração o estado corrente dos dados subjacentes. Este modelo impede que a transação enxergue dados inconsistentes, que poderiam ser causados por atualizações feitas por transações concorrentes nas mesmas linhas de dados, fornecendo um isolamento da transação para cada uma das sessões do banco de dados.

A diferença principal entre os modelos multiversão e de bloqueio é que, no MVCC, os bloqueios obtidos para consultar (ler) os dados não conflitam com os bloqueios obtidos para escrever os dados e, portanto, a leitura nunca bloqueia a escrita, e a escrita nunca bloqueia a leitura.

As funcionalidades de bloqueio, no nível de tabela e de linha, também estão disponíveis no PostgreSQL para aplicações que não podem se adaptar facilmente ao comportamento MVCC. Entretanto, a utilização apropriada do MVCC geralmente produz um desempenho melhor que os bloqueios.

Isolamento da transação

O padrão SQL define quatro níveis de isolamento de transação em termos de três fenômenos que devem ser evitados entre transações concorrentes. Os fenômenos não desejáveis são:

dirty read (leitura suja)

A transação lê dados não efetivados (uncommitted) escritos por uma transação concorrente.

nonrepeatable read (leitura que não pode ser repetida)

A transação lê uma segunda vez os dados, e descobre que os dados foram modificados por outra transação (que os efetivou após ter sido feita a leitura anterior).

phantom read (leitura fantasma)

A transação executa uma segunda vez uma consulta que retorna um conjunto de linhas que satisfaz uma determinada condição de procura, e descobre que o conjunto de linhas que satisfaz a condição é diferente devido a uma outra transação efetivada recentemente.

Tabela - 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

O PostgreSQL disponibiliza os níveis de isolamento read committed e serializable (serializável).

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 mudanças efetivadas durante a execução da consulta pelas transações concorrentes (Entretanto, o SELECT enxerga os efeitos das atualizações anteriores executadas dentro de sua própria transação, mesmo que ainda não tenham sido efetivadas). Na verdade, o comando SELECT enxerga um instantâneo do banco de dados, conforme este era no instante que a consulta começou a executar. Observe que dois comandos SELECTs sucessivos enxergam dados diferentes, mesmo estando dentro de uma mesma transação, se outras transações efetivarem alterações durante a execução do primeiro comando SELECT.

Os comando 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 a hora do início do comando. Entretanto, alguma linha de destino pode ter sido atualizada (ou excluída ou marcada para atualização) por outra transação concorrente, na hora que for encontrada. Neste caso, o atualizador (would-be updater) aguarda a transação de atualização que começou primeiro efetivar ou desfazer (se ainda estiver executando). Se o primeiro atualizador desfizer (rolls back), então seus efeitos são negados e o segundo atualizador pode prosseguir com a atualização da linha original encontrada. Se o primeiro atualizador efetivar, o segundo atualizador ignora a linha se esta foi excluída pelo primeiro atualizador, senão tenta aplicar esta operação na versão atualizada da linha. A condição de procura do comando (cláusula WHERE) é avaliada novamente para verificar se a versão atualizada da linha ainda corresponde à condição de procura. Se corresponder, o segundo atualizador prossegue sua operação começando a partir da versão atualizada da linha.

Devido à regra acima é possível os comandos de atualização enxergarem instantâneos inconsistentes --- podem enxergar os efeitos de comandos de atualização concorrentes que afetam as mesmas linhas que estão tentando atualizar, mas não enxergam os efeitos destes comandos em outras linhas do banco de dados. Este comportamento torna o Read Committed menos apropriado 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 tentam mudar concorrentemente o saldo da conta 12345 é claro que se deseja 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 nenhuma inconsistência problemática.

Uma vez que no modo Read Committed cada novo comando começa com um novo instantâneo, incluindo todas as transações efetivadas até este instante, os comandos seguintes na mesma

transação sempre enxergam os efeitos das transações concorrentes 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 muitas aplicações, e este modo é rápido e fácil de ser utilizado. Entretanto, para aplicações que efetuam consultas e atualizações complexas, pode ser necessário garantir uma visão consistente mais rigorosa do banco de dados que a fornecida pelo modo Read Committed.

Nível de isolamento serializável

Serializável é o nível mais rigoroso de isolamento da transação. Este nível emula a execução serial da transação, como se todas as transações fossem executadas uma após a outra, em série, em vez de concorrentemente. Entretanto, as aplicações que utilizam este nível de isolamento devem estar preparadas para tentar novamente as transações devido a falhas na serialização.

Quando uma transação está no nível serializável, o comando a SELECT enxerga apenas os dados efetivados antes da transação começar; nunca enxerga dados não efetivados ou mudanças efetivadas durante a execução da transação por transações concorrentes (Entretanto, o comando SELECT enxerga os efeitos de atualizações anteriores executadas dentro de sua própria transação, mesmo que ainda não tenham sido efetivadas). Isto é diferente do Read Committed, porque o comando SELECT enxerga um instantâneo do início da transação, e não do início do comando corrente dentro da transação. Portanto, comandos SELECTs 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 SELECT para encontrar as linhas de destino: somente encontram linhas de destino efetivadas até a hora do início da transação. Entretanto, alguma linha de destino pode ter sido atualizada (ou excluída ou marcada para atualização) por outra transação concorrente na hora que for 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 desfizer as alterações, então seus efeitos são negados e a transação serializável pode prosseguir com a atualização da linha original encontrada. Porém, se as alterações forem efetivadas (e a linha for realmente atualizada ou excluída, e não apenas selecionada para atualização), então a transação serializável é desfeita com a mensagem

ERROR: Can't serialize access due to concurrent update

porque uma transação serializável não pode modificar linhas alteradas por outra transação após ter começado.

Quando uma aplicação recebe esta mensagem de erro, deve abortar 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 modificação 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 o ponto de partida para a atualização na nova transação.

Observe que somente as transações que atualizam podem precisar de novas tentativas --- as transações apenas de leitura nunca ocasionam conflito de serialização.

O modo serializável fornece uma garantia rigorosa que cada transação enxerga apenas visões completamente consistentes do banco de dados. Entretanto, a aplicação precisa estar preparada para executar novamente as transações quando as atualizações concorrentes tornam impossível sustentar a ilusão de uma execução serial. Uma vez que 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 realiza várias consultas sucessivas que necessitam enxergar visões idênticas do banco de dados.

Bloqueio explícito

O PostgreSQL fornece vários modos de bloqueio para controlar o aceso concorrente aos dados nas tabelas. Estes modos podem ser utilizados para o bloqueio controlado pela transação, nas situações onde o MVCC não fornece o comportamento adequado. Também, a maioria dos comandos do PostgreSQL obtém, automaticamente, bloqueios com modos apropriados para garantir que as tabelas referenciadas não serão excluídas ou modificadas de forma incompatível enquanto o comando executa (Por exemplo, o comando ALTER TABLE não pode executar concorrentemente com outras operações na mesma tabela).

Bloqueios no nível de tabela

A lista abaixo mostra os modos de bloqueio disponíveis e os contextos nos quais estes modos são utilizados automaticamente pelo PostgreSQL. Lembre-se que todos estes modos de bloqueio são no nível de tabela, mesmo que o nome contenha a palavra "row" (linha). Os nomes dos modos de bloqueio são históricos. De alguma forma os nomes refletem a utilização típica de cada modo de bloqueio --- mas as semânticas são todas as mesmas. A única diferença real entre um modo de bloqueio e outro é o conjunto de modos de bloqueio com o qual cada um conflita. Duas transações não podem obter modos de bloqueio conflitantes na mesma tabela ao mesmo tempo (Entretanto, uma transação nunca conflita consigo mesma --- por exemplo, pode obter o bloqueio ACCESS EXCLUSIVE e mais tarde obter o bloqueio ACCESS SHARE na mesma tabela). Modos de bloqueio não conflitantes podem ser obtidos concorrentemente por muitas transações. Em particular, deve ser observado que alguns modos de bloqueio são auto-conflitantes (por exemplo, o modo de bloqueio ACCESS EXCLUSIVE não pode ser obtido por mais de uma transação ao mesmo tempo), enquanto outros não são auto-conflitantes (por exemplo, o modo de bloqueio ACCESS SHARE pode ser obtido por várias transações). Uma vez obtido, o modo de bloqueio é mantido até o fim da transação.

Para examinar a lista de bloqueios correntemente mantidos pelo servidor de banco de dados, deve ser utilizada a visão do sistema pg_locks. Para obter mais informações relativas ao monitoramento do status do subsistema de gerência de bloqueios consulte o Guia do Administrador do PostgreSQL.

Modos de bloqueio no nível de tabela

ACCESS SHARE

Conflita apenas com o modo de bloqueio ACCESS EXCLUSIVE.

O comando SELECT obtém um bloqueio deste modo nas tabelas referenciadas. Em geral, qualquer comando que apenas lê a tabela sem modificá-la obtém este modo de bloqueio.

ROW SHARE

Conflita com os modos de bloqueio EXCLUSIVE e ACCESS EXCLUSIVE.

O comando SELECT FOR UPDATE obtém o bloqueio neste modo na(s) tabela(s) de destino (além do bloqueio no modo ACCESS SHARE para as demais tabelas referenciadas mas não selecionadas FOR UPDATE).

ROW EXCLUSIVE

Conflita com os modos de bloqueio SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE.

os comandos UPDATE, DELETE e INSERT obtêm este modo de bloqueio na tabela de destino (além do modo de bloqueio ACCESS SHARE nas outras tabelas referenciadas). Em geral, este modo de bloqueio é obtido por todos os comandos que modificam os dados da tabela.

SHARE UPDATE EXCLUSIVE

Conflita com os modos de bloqueio SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE. Este modo protege a tabela contra mudanças concorrentes no esquema e a execução do comando VACUUM.

Obtida pelo comando VACUUM (sem a opção FULL).

SHARE

Conflita com os modos de bloqueio ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE. Este modo protege a tabela contra mudanças concorrentes nos dados.

Obtido pelo comando CREATE INDEX.

SHARE ROW EXCLUSIVE

Conflita com os modos de bloqueio ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE.

Este modo de bloqueio não é obtido automaticamente por nenhum comando do PostgreSQL.

EXCLUSIVE

Conflita com os modos de bloqueio ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE. Este modo permite apenas ACCESS SHARE concorrente, ou seja, somente leituras da tabela podem

prosseguir em paralelo com uma transação que obteve este modo de bloqueio.

Este modo de bloqueio não é obtido automaticamente por nenhum comando do PostgreSQL.

ACCESS EXCLUSIVE

Conflita com todos os modos de bloqueio (ACCESS SHARE, ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE). Este modo garante que a transação que o obteve é a única acessando a tabela de qualquer forma.

Obtido pelos comandos ALTER TABLE, DROP TABLE e VACUUM FULL. Este é, também, o modo de bloqueio padrão para o comando LOCK TABLE sem a especificação explícita do modo.

Nota: Somente o bloqueio ACCESS EXCLUSIVE bloqueia o comando SELECT (sem a cláusula FOR UPDATE).

Bloqueios no nível de linha

Além dos bloqueios no nível de tabela, existem os bloqueios no nível de linha. Um bloqueio no nível de linha, para uma linha específica, é obtido automaticamente quando a linha é atualizada (ou excluída ou marcada para atualização). O bloqueio é mantido até a transação efetivar ou desfazer as alterações. Os bloqueios no nível de linha não afetam a consulta aos dados; bloqueiam apenas escritas na mesma linha. Para obter um bloqueio no nível de linha sem na verdade modificar a linha, deve-se selecionar a linha por meio do comando SELECT FOR UPDATE. Observe que, após um determinado bloqueio ser obtido a transação pode atualizar a linha várias vezes sem que haja conflito.

O PostgreSQL não guarda nenhuma informação em memória relativa às linhas modificadas, portanto não existe limite no número de linhas bloqueadas de uma vez. Entretanto, o bloqueio de uma linha pode causar escrita no disco; por exemplo, o comando SELECT FOR UPDATE modifica as linhas selecionadas para marcá-las ocasionando escrita no disco.

Além dos bloqueios de tabela e de linha, também são utilizados bloqueios no nível de página, compartilhados e exclusivos, para controlar o acesso de leitura e gravação nas páginas da tabela no shared buffer pool. Estes bloqueios são liberados imediatamente após a tupla ser lida ou atualizada. Normalmente os desenvolvedores de aplicação não precisam se preocupar com bloqueios no nível de página, sendo mencionados para o assunto ficar completo.

Impasses

A utilização de bloqueios explícitos pode causar impasses (deadlocks), em particular quando duas (ou mais) transações mantêm bloqueios que outra deseja. Por exemplo, se a transação 1 obtém um bloqueio exclusivo na tabela A e, então, tenta obter um bloqueio exclusivo na tabela B, enquanto a transação 2 já possui um bloqueio exclusivo na tabela B, e agora tenta obter um bloqueio exclusivo na tabela A, então nenhuma das duas transações pode continuar. O PostgreSQL detecta automaticamente as situações de impasse, resolvendo-as abortando uma das transações envolvidas, permitindo que a(s) outra(s) prossiga(m) (Exatamente qual transação é abortada é difícil prever, não

se devendo confiar nesta previsão).

Geralmente, a melhor defesa contra os impasses é evitá-los tendo certeza que todas as aplicações que utilizam o banco de dados obtêm estes bloqueios em vários objetos em uma ordem consistente. Deve ser garantido, também, que o primeiro bloqueio de um objeto em uma transação seja aquele com o modo mais elevado que será necessário para este objeto. Se não for possível conhecer esta situação antecipadamente, então os impasses podem ser tratados em tempo de execução tentando novamente a execução das transações abortadas pelos impasses.

Enquanto a situação de impasse não é detectada, uma transação aguardando um bloqueio no nível de tabela ou no nível de linha fica aguardando indefinidamente pela liberação do bloqueio conflitante. Por isso, é uma péssima idéia as aplicações manterem transações abertas por longos períodos (por exemplo, aguardando a entrada de dados pelo usuário).

Verificação da consistência dos dados no nível da aplicação

Uma vez que a leitura no PostgreSQL não bloqueia os dados, sem levar em consideração o nível de isolamento da transação, os dados lidos por uma transação podem ser sobrescritos por outra transação concorrente. Em outras palavras, se uma linha é retornada pelo comando SELECT, isto não significa que esta linha ainda é a linha corrente no instante em que é retornada (ou seja, algum tempo depois que o comando corrente começou). A linha pode ter sido modificada ou excluída por uma transação já efetivada, que efetuou esta efetivação após a transação ter começado. Mesmo que a linha ainda seja válida "agora", esta linha pode ser mudada ou excluída antes da transação corrente efetivar ou desfazer suas modificações.

Outra maneira de pensar em relação a isto é que cada transação enxerga um instantâneo do conteúdo do banco de dados, e as transações executando concorrentemente podem perfeitamente enxergar instantâneos diferentes. Portanto, o próprio conceito de "agora" é de alguma forma suspeito. Normalmente isto não é um grande problema quando as aplicações cliente estão isoladas uma das outras, mas se os clientes podem se comunicar por meio de canais por fora do banco de dados, então sérias confusões podem acontecer.

Para garantir a validade corrente de uma linha e protegê-la contra atualizações concorrentes, deve ser utilizado o comando SELECT FOR UPDATE ou uma declaração LOCK TABLE apropriada (o comando SELECT FOR UPDATE bloqueia apenas as linhas selecionadas contra atualizações concorrentes, enquanto o LOCK TABLE bloqueia toda a tabela). Isto deve ser levado em consideração ao portar aplicações de outros ambientes para o PostgreSQL.

Nota: Antes da versão 6.5 o PostgreSQL utilizava bloqueios de leitura e, portanto, as considerações acima também se aplicam quando é feita uma atualização de uma versão do PostgreSQL anterior a 6.5.

As verificações globais de validade requerem considerações extras sob o MVCC. Por exemplo, uma aplicação bancária pode desejar verificar se a soma de todos os créditos em uma tabela é igual a soma de todos os débitos em outra tabela, no momento em que as duas tabelas estão sendo ativamente atualizadas. Comparar os resultados de dois comando SELECT SUM(...) sucessivos não vai funcionar confiavelmente no modo Read Committed, porque o segundo comando, provavelmente, inclui resultados de transações que não apareciam no primeiro comando. Realizar as

duas somas em uma mesma transação serializável fornece uma imagem precisa dos efeitos das transações efetivadas antes do início da transação serializável --- mas pode ser legitimamente questionado se a resposta ainda é relevante na hora que foi produzida. Se a própria transação serializável introduziu algumas mudanças antes de tentar efetuar a verificação de consistência, o valor prático da verificação fica ainda mais discutível, porque agora são incluídas algumas, mas não todas, as mudanças ocorridas após o início da transação. Em casos como este, uma pessoa cuidadosa pode desejar bloquear todas as tabelas necessárias para fazer a verificação, para obter uma imagem da situação atual acima de qualquer suspeita. Um bloqueio no modo SHARE (ou superior) garante não haver mudanças não efetivadas na tabela bloqueada, fora as mudanças efetuadas pela própria transação corrente.

Observe também que, quando a prevenção contra alterações concorrentes está baseada em bloqueios explícitos, deve ser utilizado o modo Read Committed, ou tomar-se o cuidado de obter os bloqueios antes de executar os comandos no modo serializável. Um bloqueio explícito obtido em uma transação serializável garante que nenhuma outra transação modificando a tabela está executando --- mas se o instantâneo enxergado pela transação for anterior à obtenção do bloqueio, pode ser que seja anterior a algumas mudanças na tabela que agora estão efetivadas. Um instantâneo de uma transação serializável é, na verdade, tirado no início do primeiro comando (SELECT, INSERT, UPDATE ou DELETE) sendo, portanto, possível obter o bloqueio explícito antes do instantâneo ser tirado.

Bloqueio e índices

Embora o PostgreSQL forneça acesso de leitura e gravação não bloqueante aos dados das tabelas, o acesso de leitura e gravação não bloqueante não é oferecido, atualmente, para todos os métodos de acessos dos índices implementados pelo PostgreSQL.

Os tipos de índices existentes são implementados do seguinte modo:

Índices B-tree

São utilizados bloqueios compartilhados/exclusivos no nível de página, de curta duração, para acesso de leitura/gravação. Os bloqueios são liberados imediatamente após cada tupla do índice ser lida ou inserida. Os índices B-tree fornecem a concorrência mais elevada sem condições de impasse.

Índices GiST e R-tree

São utilizados bloqueios compartilhados/exclusivos no nível de índice para acessos de leitura/gravação. Os bloqueios são liberados após a declaração (comando) ser executada.

Índices Hash

São utilizados bloqueios compartilhados/exclusivos no nível de página para acessos de leitura/gravação. Os bloqueios são liberados após a página ser processada. Os bloqueios no nível de página permitem uma concorrência melhor que o bloqueio no nível de índice, mas podem ocasionar impasses.

Em resumo, o índice B-tree é o tipo de índice recomendado para as aplicações correntes.

CAPÍTULO 13. Dicas e Macetes

1. Desempenho

O desempenho dos comandos pode ser afetado por vários motivos. Alguns destes motivos podem ser tratados pelo usuário, enquanto outros estão intrinsecamente ligados ao projeto do sistema subjacente. Este capítulo fornece algumas dicas para ajudar na compreensão e ajuste do desempenho do PostgreSQL.

Utilização do comando EXPLAIN

O PostgreSQL concebe um plano de consulta para cada consulta solicitada. A escolha do plano correto, correspondendo à estrutura da consulta e às propriedades dos dados, é absolutamente crítico para o bom desempenho. Para toda consulta pode ser utilizado o comando EXPLAIN, para ver o plano criado pelo sistema. A leitura de um plano é uma arte e merece um tutorial extenso, que este não é; neste capítulo são fornecidas algumas informações básicas.

Os números apresentados atualmente pelo EXPLAIN são:

- O custo de partida estimado (O tempo gasto antes da varredura da saída poder começar como, por exemplo, o tempo para fazer a ordenação em um nó de ordenação).
- O custo total estimado (Se todas as linhas forem buscadas, que pode não acontecer --- uma consulta contendo a cláusula LIMIT não gastará o custo total, por exemplo).
- Número de linhas de saída estimado para este nó do plano (Novamente, apenas quando executado até o fim).
- Largura média estimada (em bytes) das linhas de saída para este nó do plano.

Os custos são medidos em termos de unidades de páginas buscadas no disco (O esforço de CPU estimado é convertido em unidades de página de disco utilizando fatores estipulados altamente arbitrários. Se for desejado realizar experiências com estes fatores, deve ser consultada a lista de parâmetros de configuração em tempo de execução no Guia do Administrador do PostgreSQL.)

É importante perceber que o custo do nível mais alto inclui todos os custos de seus descendentes. Também é importante perceber que o custo reflete apenas as partes com as quais o planejador/otimizador se preocupa. Em particular, o custo não considera o tempo gasto transmitindo o resultado para o cliente --- que pode ser um fator importante no computo do tempo total gasto, mas que o planejador ignora porque não é alterado pela mudança de plano (Todo plano correto produz o mesmo conjunto de linhas, acredita-se).

Linhas produzidas é um assunto delicado porque não é o número de linhas processadas/varridas pela comando --- geralmente é menos, refletindo a seletividade estimada das restrições de certas

cláusulas WHERE aplicadas. Idealmente, as linhas de nível superior estimam de forma aproximada o número de linhas realmente retornadas, atualizadas ou excluídas pelo comando.

A seguir são apresentados alguns exemplo (utilizando o banco de dados de teste de regressão após a execução do comando VACUUM ANALYZE, e os fontes de desenvolvimento da 7.3:

```
regression=# EXPLAIN SELECT * FROM tenk1;
                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..333.00 rows=10000 width=148)
```

Este resultado é tão simples quanto parece. Se for executado

```
SELECT * FROM pg_class WHERE relname = 'tenk1';
```

será visto que tenk1 possui 10.000 linhas e ocupa 233 páginas de disco. Portanto, o custo é estimado em 233 páginas lidas, definidas como custando 1.0 cada uma, vezes 10.000 * cpu_tuple_cost que vale atualmente 0.01 (tente executar SHOW cpu_tuple_cost).

Agora a consulta será modificada para incluir uma condição WHERE:

```
regression=# EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;
                         QUERY PLAN
-------------------------------------------------------------
Seq Scan on tenk1  (cost=0.00..358.00 rows=1033 width=148)
  Filter: (unique1 < 1000)
```

A estimativa de linhas produzidas diminuiu devido à cláusula WHERE. Entretanto, a varredura ainda precisa percorrer todas as 10.000 linhas, portanto o custo não diminuiu; na verdade aumentou um pouco, para refletir o tempo a mais de CPU gasto verificando a condição WHERE.

O número correto de linhas que esta consulta deveria selecionar é 1.000, mas a estimativa é somente aproximada. Se a repetição desta experiência for tentada, provavelmente será obtida uma estimativa ligeiramente diferente; além disso, mudanças ocorrem após cada comando ANALYZE, porque as estatísticas produzidas pelo ANALYZE são obtidas a partir de amostras aleatórias na tabela.

Modificando a consulta para restringir mais ainda a condição

```
regression=# EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 50;
                               QUERY PLAN
------------------------------------------------------------------------------
Index Scan using tenk1_unique1 on tenk1  (cost=0.00..179.33 rows=49 width=148)
  Index Cond: (unique1 < 50)
```

será visto que a condição WHERE ficou bastante seletiva, e o planejador no final decidirá que a varredura do índice é mais barata que a varredura seqüencial. Este plano necessita ler apenas 50 linhas devido ao índice, saindo vencedor apesar do fato de cada busca individual ser mais cara que a leitura de toda a página do disco seqüencialmente.

Adição de outra cláusula à condição WHERE:

```
regression=# EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 50 AND
regression-# stringu1 = 'xxx';
                            QUERY PLAN
------------------------------------------------------------------------------
Index Scan using tenk1_unique1 on tenk1  (cost=0.00..179.45 rows=1 width=148)
  Index Cond: (unique1 < 50)
  Filter: (stringu1 = 'xxx'::name)
```

A cláusula adicionada stringu1 = 'xxx' reduz a estimativa de linhas produzidas mas não o custo, porque deverá ser lido o mesmo conjunto de linhas. Observe que a cláusula stringu1 não pode ser aplicada como uma condição do índice (porque o índice abrange apenas a coluna unique1). Em vez disto, esta cláusula é aplicada como um filtro nas linhas obtidas pelo índice. Portanto, o custo na verdade sobe um pouco para refletir esta verificação adicional.

A seguir é feita a junção de duas tabelas, utilizando os campos sendo discutidos:

```
regression=# EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 50
regression-# AND t1.unique2 = t2.unique2;
                          QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..327.02 rows=49 width=296)
   ->  Index Scan using tenk1_unique1 on tenk1 t1
                              (cost=0.00..179.33 rows=49 width=148)
         Index Cond: (unique1 < 50)
   ->  Index Scan using tenk2_unique2 on tenk2 t2
                              (cost=0.00..3.01 rows=1 width=148)
         Index Cond: ("outer".unique2 = t2.unique2)
```

Nesta junção de laço aninhado, a varredura externa é a mesma varredura de índice vista no penúltimo exemplo e, portanto, seu custo e quantidade de linhas são os mesmos porque está sendo aplicada a cláusula unique1 < 50 na condição WHERE neste nó. A cláusula t1.unique2 = t2.unique2 ainda não é relevante e, portanto, não afeta a quantidade de linhas da varredura externa. Para a varredura interna, o valor de unique2 da linha corrente da varredura externa é conectado na varredura interna do índice para produzir uma condição de índice do tipo t2.unique2 = constante. Portanto, seria obtido o mesmo plano e custo para a varredura interna que seria obtido por, digamos, EXPLAIN SELECT * FROM tenk2 WHERE unique2 = 42. O custo do nó do laço é então estabelecido tomando por base o custo da varredura externa, mais a repetição da varredura interna para cada linha externa (49 * 3.01, neste caso), mais um pouco de tempo de CPU para o processo de junção.

Neste exemplo, a quantidade de linhas do laço é igual ao produto da quantidade de linhas das duas varreduras, mas isto não é verdade usualmente porque, em geral, podem existir cláusulas WHERE fazendo menção às duas relações e, portanto, só podem ser aplicadas no ponto de junção, e não nas duas varreduras de entrada. Por exemplo, se fosse incluído WHERE ... AND t1.hundred < t2.hundred, faria diminuir a quantidade de linhas de saída do nó da junção, mas não mudaria

nenhuma das varreduras da entrada.

Uma forma para ver planos alternativos é forçar o planejador não levar em consideração a estratégia que seria a vencedora, ativando e desativando os sinalizadores de cada tipo de plano (Esta é uma ferramenta deselegante, mas útil.).

```
regression=# SET enable_nestloop = off;
SET
regression=# EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 50
regression-# AND t1.unique2 = t2.unique2;
                          QUERY PLAN
------------------------------------------------------------------------
 Hash Join  (cost=179.45..563.06 rows=49 width=296)
   Hash Cond: ("outer".unique2 = "inner".unique2)
   ->  Seq Scan on tenk2 t2  (cost=0.00..333.00 rows=10000 width=148)
   ->  Hash  (cost=179.33..179.33 rows=49 width=148)
         ->  Index Scan using tenk1_unique1 on tenk1 t1
                               (cost=0.00..179.33 rows=49 width=148)
               Index Cond: (unique1 < 50)
```

Este plano propõe extrair as 50 linhas que interessam de tenk1, usando a mesma varredura de índice anterior, armazenar estas linhas em uma tabela hash na memória e, então, executar uma varredura seqüencial em tenk2, comparando cada linha de tenk2 com a tabela hash para verificar possíveis correspondências de t1.unique2 = t2.unique2. O custo para ler tenk1 e preparar a tabela hash é inteiramente custo de partida para a junção hash, porque não sairá nenhuma linha até começar a leitura de tenk2. O tempo total estimado para a junção também inclui uma pesada carga de tempo de CPU para verificar a tabela hash 10.000 vezes. Entretanto deve ser observado que não está sendo computado 10.000 vezes 179.33; a montagem da tabela hash é feita somente uma vez neste tipo de plano.

É possível verificar a precisão dos custos estimados pelo planejador utilizando o comando EXPLAIN ANALYZE. Este comando na verdade executa a consulta, e depois mostra o tempo real acumulado dentro de cada nó do plano junto com os mesmos custos estimados que o comando EXPLAIN simples mostraria. Por exemplo, poderia ser obtido um resultado como este:

```
regression=# EXPLAIN ANALYZE
regression-# SELECT * FROM tenk1 t1, tenk2 t2
regression-# WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique2;
                          QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..327.02 rows=49 width=296)
                        (actual time=1.18..29.82 rows=50 loops=1)
   ->  Index Scan using tenk1_unique1 on tenk1 t1
               (cost=0.00..179.33 rows=49 width=148)
                        (actual time=0.63..8.91 rows=50 loops=1)
         Index Cond: (unique1 < 50)
   ->  Index Scan using tenk2_unique2 on tenk2 t2
               (cost=0.00..3.01 rows=1 width=148)
```

```
                    (actual time=0.29..0.32 rows=1 loops=50)
      Index Cond: ("outer".unique2 = t2.unique2)
Total runtime: 31.60 msec
```

Deve ser observado que os valores de "actual time" (tempo real) são em milissegundos de tempo real, enquanto as estimativas de "cost" (custo) são expressas em unidades arbitrárias de busca em disco; portanto, não é provável haver correspondência. É nas relações que se deve prestar atenção.

Em alguns planos de consulta é possível que o nó de um subplano seja executado mais de uma vez. Por exemplo, a varredura do índice interno é executada uma vez para cada linha externa no plano de laço aninhado acima. Nestes casos, o valor "loops" indica o número total de execuções do nó, enquanto os valores tempo real (actual time) e linhas (rows) mostram médias por execução. Isto é feito para tornar os números comparáveis com o modo como os custos estimados são mostrados. Deve ser multiplicado pelo valor "loops" para obter o tempo total realmente gasto no nó.

O Total runtime (tempo total de execução) mostrado pelo EXPLAIN ANALYZE inclui os tempos de inicialização e de finalização do executor, assim como o tempo gasto processando os resultados das linhas. Não inclui os tempos de análise, re-escrita e planejamento. Para um comando SELECT, o tempo total de execução normalmente será apenas um pouco maior que o tempo total relatado pelo nó de nível mais alto do plano. Para os comandos INSERT, UPDATE e DELETE , o tempo total de execução pode ser consideravelmente maior, porque inclui o tempo gasto processando as linhas resultantes. Nestes comandos, o tempo para o nó superior do plano é, essencialmente, o tempo gasto computando as novas linhas e/ou localizando as linhas antigas, mas não inclui o tempo gasto realizando estas alterações.

Vale a pena observar que os resultados do EXPLAIN não devem ser extrapolados para outras situações além da que está sendo testada; por exemplo, não é possível supor que os resultados para uma tabela pequena possam ser aplicados em uma tabela grande. Os custos do planejador não são lineares e, portanto, podem ser escolhidos planos diferentes para tabelas grandes e tabelas menores. Um exemplo extremo é o caso de uma tabela que só ocupa uma página do disco, onde quase sempre vence o plano de varredura seqüencial, estejam os índices disponíveis ou não. O planejador percebe que fará a leitura de uma página do disco para processar a tabela em qualquer caso e, portanto, não faz sentido fazer leituras de páginas adicionais para procurar em um índice.

Estatísticas utilizadas pelo planejador

Conforme foi visto na seção anterior, o planejador de consultas precisa estimar o número de linhas encontradas pela consulta, para poder fazer boas escolhas dos planos de consulta. Esta seção fornece uma breve visão das estatísticas utilizadas pelo sistema para realizar estas estimativas.

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. Isto pode ser visto utilizando consultas semelhantes à mostrada abaixo:

```
regression=# SELECT relname, relkind, reltuples, relpages FROM pg_class
regression-# WHERE relname LIKE 'tenk1%';
   relname    | relkind | reltuples | relpages
```

```
--------------+---------+-----------+----------
 tenk1        | r       |     10000 |      233
 tenk1_hundred | i      |     10000 |       30
 tenk1_unique1 | i      |     10000 |       30
 tenk1_unique2 | i      |     10000 |       30
(4 rows)
```

Pode ser visto que tenk1 contém 10.000 linhas, assim como seus índices, mas que os índices são (sem surpresa) muito menores que a tabela.

Por razões de eficiência, as colunas reltuples e relpages não são atualizadas dinamicamente e, portanto, usualmente contêm valores aproximados apenas (que são suficientemente bons para as finalidades do planejador). Estas colunas são inicializadas com valores fictícios (atualmente 1.000 e 10, respectivamente) quando a tabela é criada. São atualizadas por alguns comandos, atualmente VACUUM, ANALYZE e CREATE INDEX. Um comando autônomo ANALYZE, que não faça parte do VACUUM, coloca um valor aproximado em reltuples porque não são lidas todas as linhas da tabela.

A maioria das consultas retorna apenas uma parte das linhas da tabela, devido à cláusula WHERE que restringe as linhas a ser examinadas. Portanto, o planejador precisa fazer uma estimativa da seletividade das cláusulas do WHERE, ou seja, a fração das linhas correspondendo a cada cláusula da condição WHERE. A informação utilizada para esta tarefa é armazenada no catálogo do sistema pg_statistic. As entradas em pg_statistic são atualizadas pelos comandos ANALYZE e VACUUM ANALYZE, sendo sempre aproximadas, mesmo logo após serem atualizadas.

Em vez de olhar pg_statistic diretamente, é melhor olhar sua visão pg_stats para examinar as estatísticas manualmente. A visão pg_stats é projetada para ser lida mais facilmente. Além disso, pg_stats pode ser lida por todos, enquanto pg_statistic somente pode ser lida pelo superusuário. (Isto impede que usuários não privilegiados obtenham informações relativas ao conteúdo da tabela de outras pessoas a partir de suas estatísticas. A visão pg_stats mostra somente linhas relativas às tabelas que o usuário corrente pode ler). Por exemplo, poderia ser executado:

regression=# SELECT attname, n_distinct, most_common_vals FROM pg_stats WHERE tablename = 'road';

attname	n_distinct	most_common_vals
name	-0.467008	{"I- 580 Ramp","I- 880 Ramp","Sp Railroad ","I- 580 ","I- 680 Ramp","I- 80 Ramp","14th St ","5th St ","Mission Blvd","I- 880 "}
thepath	20	{"[(-122.089,37.71),(-122.0886,37.711)]"}

(2 rows)
regression=#

Tabela - Colunas da visão pg_stats

Nome	Tipo	Descrição
tablename	name	Nome da tabela contendo a coluna
attname	name	Coluna descrita por esta linha
null_frac	real	Fração das entradas nulas na coluna
avg_width	integer	Largura média em bytes das entradas da coluna
n_distinct	real	

Se for maior que zero, o número estimado de valores distintos presentes na coluna. Se for menor que zero, o negativo do número de valores distintos dividido pelo número de linhas (A forma negativa é utilizada quando o ANALYZE acredita que o número de valores distintos possivelmente aumentará junto com o crescimento da tabela; a forma positiva é utilizada quando a coluna parece ter um número fixo de valores possíveis). Por exemplo, -1 indica uma coluna com restrição de unicidade onde o número de valores distintos é o mesmo que a quantidade de linhas.
most_common_vals text[] A lista dos valores mais comuns da coluna (Omitido se nenhum valor parece ser mais comum que os outros).

most_common_freqs real[] A lista das freqüências dos valores mais comuns, ou seja, o número de ocorrências de cada um dividido pelo número total de linhas.

histogram_bounds text[]
A lista dos valores que dividem os valores das colunas em grupos com populações aproximadamente iguais. Se most_common_vals existir, estes valores são omitidos no cálculo do histograma (Omitido se o tipo de dado da coluna não possui o operador <, ou se a lista most_common_vals for toda a população).

Correlation real
Correlação estatística entre a ordem física das linhas e a ordenação lógica dos valores da coluna. O intervalo é de -1 a +1. Quando o valor está próximo de -1 ou de +1, uma varredura de índice na coluna será estimada como sendo mais barata do que quando estiver próximo de zero, devido à redução de acessos aleatórios no disco (Omitido se o tipo de dado da coluna não possui o operador <).

O número máximo de entradas nas matrizes most_common_vals e histogram_bounds podem ser definidos coluna por coluna utilizando o comando ALTER TABLE SET STATISTICS. Atualmente, o limite padrão é de 10 entradas. Aumentar o valor pode permitir o planejador realizar estimativas mais precisas, em particular para colunas com distribuição de dados irregular, ao custo de consumir mais espaço em pg_statistic e um pouco mais de tempo para calcular as estimativas. Inversamente, um limite mais baixo pode ser apropriado para colunas com distribuição simples dos dados.

Controlando o planejador com cláusulas JOIN explícitas

A partir do PostgreSQL 7.1 é possível algum controle sobre o planejador de consultas utilizando a sintaxe do JOIN. 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 consulta juntando A e 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 WHERE. Poderia, também, juntar B e C e depois juntar A ao resultado. Ou, também, juntar A e C e depois juntar B --- mas isto não seria eficiente, uma vez que o produto Cartesiano completo de A e C deveria ser produzido, porque não existe nenhuma cláusula na condição WHERE que permita a otimização desta junção (Todas as junções no executor do PostgreSQL acontecem entre duas tabelas de entrada sendo, portanto, necessária a construção do resultado a partir de uma ou outra destas modalidades). O ponto a ser destacado é que estas diferentes possibilidades de junção produzem resultados semanticamente equivalentes, mas com custos de execução muito diferentes. Portanto, o planejador explora todas as possibilidades tentando encontrar o plano de consulta mais eficiente.

Quando uma consulta envolve apenas duas ou três tabelas não existem muitas possibilidades de junção com que se preocupar. Porém, o número de possibilidades de junção cresce exponencialmente quando o número de tabelas aumenta. Acima de dez tabelas de entrada não é mais prático efetuar uma procura exaustiva de todas as possibilidades, e mesmo para seis ou sete tabelas o planejamento pode levar um longo tempo. Quando existem muitas tabelas de entrada, o planejador do PostgreSQL pode alternar de uma procura exaustiva para uma procura probabilística genética com um número limitado de possibilidades (O ponto de mudança é definido pelo parâmetro em tempo de execução GEQO_THRESHOLD descrito no Guia do Administrador do PostgreSQL). A procura genética leva menos tempo, mas não encontra necessariamente o melhor plano possível.

Quando a consulta envolve junções externas, o planejador tem muito menos liberdade que com a junções comuns (internas). Por exemplo, considerando-se:

SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);

Embora as restrições desta consulta sejam superficialmente semelhantes às do exemplo anterior, as semânticas são diferentes porque uma linha deve ser gerada para cada linha de A que não possua linha correspondente na junção de B com C. Portanto, o planejador não pode escolher a junção neste caso: deve juntar B e C e depois juntar A ao resultado. Portanto, esta consulta leva menos tempo para ser planejada que a consulta anterior.

O planejador de consultas do PostgreSQL trata todas as sintaxes explícitas de JOIN como restrições das possibilidades de junção, embora não seja logicamente necessário fazer esta restrição para junções internas. Portanto, embora todas as consultas abaixo produzam o mesmo resultado

SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);

a segunda e a terceira levam menos tempo para serem planejadas que a primeira. Não há motivo para se preocupar com este efeito para duas ou três tabelas, mas pode ser de grande ajuda quando

existem muitas tabelas.

Não é necessário eliminar totalmente as possibilidades de junção para diminuir o tempo de procura, porque é correto utilizar operadores JOIN no FROM comum. Por exemplo,

SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;

força o planejador juntar A e B antes de juntá-las às outras tabelas, mas não restringe as possibilidades além disto. Neste exemplo, o número de possibilidades possíveis de junção é reduzida por um fator de cinco.

Havendo um misto de junções internas e externas em uma consulta complexa, pode não ser desejado restringir a procura do planejador por uma boa ordem nas junções internas dentro da junção externa. Isto não pode ser feito diretamente por meio da sintaxe do JOIN, mas a limitação da sintaxe pode ser contornada utilizando subconsultas. No exemplo

```
SELECT * FROM d LEFT JOIN
    (SELECT * FROM a, b, c WHERE ...) AS ss
    ON (...);
```

a junção com D será a última etapa do plano da consulta, mas o planejador está livre para considerar as diversas possibilidades de junção entre A, B e C.

Restringir as procuras do planejador desta forma é uma técnica útil tanto para reduzir o tempo de planejamento quanto para direcionar o planejador para um bom plano. Se o planejador escolher uma ordem de junção ruim por padrão, pode ser forçado a escolher uma ordem melhor por meio da sintaxe do JOIN --- assumindo o conhecimento uma ordem melhor. Recomenda-se a realização de experiências.

Carga dos dados no banco

Pode ser necessário efetuar um grande número de inserções em tabelas para fazer a carga inicial do banco de dados. Abaixo são mostradas algumas dicas e técnicas para tornar esta carga a mais eficiente possível.

Desativar a auto-efetivação

Desativar a auto-efetivação (autocommit) e fazer apenas uma efetivação no final (Em SQL puro e simples isto significa executar o BEGIN no começo e o COMMIT no fim. Algumas bibliotecas cliente fazem isto automaticamente e, neste caso, é preciso ter certeza que a biblioteca está realmente fazendo quando é necessário ser feito). Se for permitido efetivar cada inserção separadamente, o PostgreSQL terá muito trabalho cada vez que uma linha for inserida. Um benefício adicional de fazer todas as inserções em uma transação é que, quando a inserção de uma linha falha a transação é desfeita, desconsiderando todas as linhas inseridas até este ponto, não produzindo uma carga parcial dos dados.

Utilização do COPY FROM

Em vez de utilizar vários comandos INSERT, deve ser utilizado o COPY FROM STDIN para carregar todas as linhas em um único comando, obtendo uma redução substancial de esforço gerado pela análise, planejamento, etc. Se isto for feito então não será necessário desativar a auto-efetivação, porque se trata de um único comando.

Remoção dos índices

Se estiver sendo carregada uma tabela recém criada, a maneira mais rápida é criar a tabela, carregar os dados com o COPY e, então, criar todos os índices necessários para a tabela. Criar um índice sobre dados pré-existentes é mais rápido que fazer de forma incremental durante a carga de cada linha.

Para carregar uma tabela existente, pode ser executado o comando DROP INDEX, carregar a tabela e, então, recriar o índice. É claro que o desempenho do banco de dados para os outros usuários será afetado negativamente durante o tempo que o índice não existir. Deve-se pensar duas vezes antes de remover um índice único, porque a verificação de erro efetuada pela restrição de unicidade não existirá enquanto o índice não for criado novamente.

Executar o comando ANALYZE depois

É aconselhado executar o comando ANALYZE ou VACUUM ANALYZE sempre que forem adicionados ou excluídos muitos dados, inclusive logo após a carga inicial da tabela. Este procedimento garante que o planejador dispõe de estatísticas atualizadas relativas à tabela. Sem estatísticas, ou com estatísticas obsoletas, o planejador pode fazer escolhas ruins para os planos das consultas, ocasionando um mau desempenho das consultas que utilizam a tabela.

2. Tratamento de arquivos com delimitadores

Gerando arquivo texto com delimitadores

COPY <tabela> TO '<arquivo>' DELIMITERS '|';

Trazendo o conteúdo de arquivo para o banco

COPY <tabela> FROM '<arquivo>' DELIMITERS '|';

3. Tratamento de Data / Horas

Tipos de dados do postgresql:

Formato	Exemplo
ISO	2001-06-25 12:24:00-07
SQL	06/25/2001 12:24:00.00 PDT
Postgres	Mon 25 Jun 12:24 2001 PDT
German	25/06/2001 12:24:00.00 PDT

Exemplo:

```
SET DATESTYLE TO SQL;
SET DATESTYLE DO GERMAN;
SELECT CURRENT_DATE;
SELECT NOW();
SHOW DATESTYLE;
```

Para que fique permanente edite o arquivo postgresql.conf

EXTRACT

Extrai informações de determinada data.

extract (<campo> FROM <origem>)

Exemplos:

Campo	Exemplo (Base 19/07/2005 11:20:00)	Resultado
Século	SELECT EXTRACT(CENTURY FROM now());	20
Dia	SELECT EXTRACT(DAY FROM now());	16
Dia da semana	SELECT EXTRACT(DOW FROM now());	5
Dia do ano	SELECT EXTRACT(DOY FROM now());	200
Época	SELECT EXTRACT(EPOCH FROM now()); Número de segundos desde 1970-01-01 00:00:00:00-00	4070908800
Hora	SELECT EXTRACT(HOUR FROM now());	11
Minuto	SELECT EXTRACT(MINUTE FROM now());	20
Mês	SELECT EXTRACT(MONTH FROM now());	7
Ano	SELECT EXTRACT(YEAR FROM now());	2005

4. Banco Modelo

Herdando todos os seus objetos

```
createdb -U postgres -T <nome_do_modelo> <nome_do_banco>
```

template1 – serve de modelo/base para todos os bancos criados no servidor. Quando houver a necessidade de uma tabela ser herdada por todos oso bancos criados basta inserir essa tabela no template1. Um bom_exemplo é inserir PL/PgSQL no template1 para que todos os bancos criados a partir de então herdem a PL.

5. Transações

BEGIN - abre uma transação;

COMMIT - fecha a transação confirmando as alterações;

ROLLBACK - fecha a transação cancelando as alterações.

Exemplo:

```
BEGIN
        SELECT...
        DELETE ...
COMMIT
ou
ROLLBACK
```

6. Replicação[14]

O recurso utilizado para replicação é o pacotre de contribuição dblink, que não vem ativado por padrão. Para ativá-lo devemos importar seu arquivo.

psql -U postgres <nome_do_banco> </usr/share/pgsql/contrib/dblink.sql>
ou no modo console:
\i /usr/share/pgsql/contrib/dblink.sql

dblink - para SELECT
dblinkexec - para INSERT, UPDATE, DELETE (remotos).

Exemplo: Select

```
SELECT * FROM dblink
        (
                'dbname=pgteste
                hostaddr=200.174.40.63
                user=paulo
                password=paulo
                port=5432,
                'SELECT nome FROM clientes'
        ) AS t1 (nome varchar(30));
```

Exemplo: Insert

```
SELECT * FROM dblink_exec
        (
                'dbname=pgteste
                hostaddr=200.174.40.63
                user=paulo
                password=paulo
                port=5432,
                'INSERT INTO clientes(nome) values (“roger”)'
        ) ;
```

14 Veja tutorial no sitio web www.dbexperts.com.br

7. Stored Procedures

Stored Procedures no PostgreSQL são funções definidas em alguma linguagem de procedimentos suportada. Atualmente existem: PL/PgSQL, PL/Java, Pl/PHP, entre outras.

Exemplo 1:

```
CREATE FUNCTION populate () RETURNS integer AS '
        DECLARE
                -- declarations
        BEGIN
                perform my_function();
        END;
'LANGUAGE plpgsql;
```

Ligando o suporte à linguagem plpgsql: #createlang -U postgres <nome_do_banco>

As procedures em PlpgSQL são formadas por:
- Functions
- Comandos em SQL
- Comandos em PlpgSQL

Quando usar apenas funções ou usar a PlpgSQL ?

Quando precisamos usar estruturas “if, for, ...”

A PL/PgSQL foi projetada para:

- Criar funções e trigger procedures;
- Adicionar estruturas de controle para a SQL Language;
- Executar cálculos complexos;
- Herdar todos os user-defined types, functions e operators;
- Ser definida para ser confiável pelo servidor;
- Fácil de usar.

Exemplos de declarações de variáveis:

```
user_id integer;
quantity numeric(5);
url varchar(20);
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

Exemplo 2:

```
CREATE FUNCTION  sales_tax(real) RETURN real as '
        DECLARE
                subtotal   ALIAS FOR $1;
        BEGIN
                RETURN subtotal * 0.06;
        END;
' LANGUAGE plpgsql;
```

Exemplo 3:

```
CREATE FUNCTION instr(varchar, integer) RETURN integer AS '
        DECLARE
        v_string ALIAS FOR $1;
        index ALIAS FOR $2;
        BEGIN
                -- cálculos
        END;
' LANGUAGE plpgsql;
```

Exemplo 4:

```
CREATE FUNCTION concatenar ( tablename ) RETURN TEXT AS '
        DECLARE
                in_t ALIAS FOR $1;
        BEGIN
                RETURN int_t.f1 || int_t.f5 || int_t.f7;
        END;
' LANGUAGE plpgsql;
```

Exemplo 5:

```
CREATE FUNCTION exemplo4 ( anyelement, anyelement, anyelement ) RETURN anyelement AS
'       DECLARE
                result ALIAS  FOR   $0;
                first ALIAS FOR $1;
                second ALIAS FOR $2;
                third ALIAS  FOR $3;
        BEGIN
                result := firts + second + third;
                RETURN result;
        END;
' LANGUAGE plpgsql;
```

8. Triggers

Gatilhos que disparam os eventos dos objetos dos bancos.

Podem ser escritas em C ou em qualquer das demais linguagens procedurais disponíveis.

Atualmente não é possível escrever triggers na linguagem SQL.

Uma trigger pode ser definida para executar antes ou após uma operação de INSERT, UPDATE ou DELETE, para modificar o registro ou executar uma cláusula SQL.

A função trigger precisa ser definida antes da própria trigger. A função da trigger precisa ser declarada como uma função sem nenhum argumento, e que retorne um tipo de trigger.

Uma função de trigger pode ser usada por diversas trigger.

Tipos de trigger:

per-row - a função da trigger é invocada cada vez que a row é afetada pelos procedimentos da trigger;

per-statement - é invocado soment quando o apropriado statement é executado.

A palavara reservada para representar “triggers”, que é opaque, já foi adequadamente substituida por “triggers” na versão 7.4.

TRIGGERS PROCEDURES - Em PL/PgSQL

```
CREATE TABLE emp (
        empname text,
        salary integer,
        last_data timestamp,
        last_user text
);
```

```
CREATE FUNCTION emp_stmp ()  RETURN trigger AS '
        BEGIN
                --  check that empname and salary are given
                IF NEW.empname IS NULL THEN
                        RAISE EXCEPTION "empname cannot be null";
                END IF;
                IF NEW.salary IS NULL THEN
                        RAISE EXCEPTION "% cannot have null salary",
                        NEW.empname;
```

```
                END IF;
                -- who works for us when she must pay for it ?
                IF NEW.salary < 0 THEN
                        RAISE EXCEPTION "% cannot have a negative salary",
                        NEW.empname;
                END IF;
                -- Remember who changed the payroll when
                NEW.las_date :=  "now";
                NEW.last_user := current_user;
                RETURN NEW;
        END;
' LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON empname
        FOR EACH ROW EXECUTE PROCEDURE emp_stamp();

CREATE CONSTRAINTS TRIGGER name
        AFTER events ON
        tablename constraint attribute
        FOR EACH ROW EXECUTE PROCEDURE funcname ( args )
```

9. Schemas

Bancão
public
banco1 Schemas (como sub bancos)
banco2
...

Vantagens: organização, segurança e relacionamentos.

De bancos diferentes não podemos relacionar tabelas, mas de dentro de um mesmo banco com schemas diferentes podemos.

Criando SCHEMA

```
CREATE SCHEMA cristina authorization Cristina;
CREATE SCHEMA ana authorization ana;
CREATE SCHEMA katia authorization katia;
```

Cria schema cristina e torna o user cristina o dono

```
SELECT * FROM cristina.clientes  UNION SELECT  * FROM katia.clientes;
```

10. Usuários e Grupos

Um usuário comum será considerado super-usuário do sistema quando puder criar outros usuários e outros database.

Criação

CREATE USER <usuario> PASSWORD '<senha>' CREATEDB|NOCREATEDB CREATEUSER| NOCREATEUSER IN GROUP <nome_do_grupo>;

Removendo ...

DROP USER <usuario>;

Alterando ...

ALTER GROUP ADD USER <usuario>;

Removendo usuários do grupo ...

ALTER GROUP DROP USER<usuario>;

ALTER USER <usuario> PASSWORD '<senha>' CREATEDB | NOCREATEDB CREATEUSER | NOCREATEUSER IN GROUP <grupo>

Alterando permissões para grupos ou usuários:

GRANT SEL|INS|UPD|DEL|RULE|REFERENCES|TRIGGER|ALL ON <tabela>
TO <usuário> ou TO GROUP <grupo>

Removendo permissões:

REVOKE SEL|INS|UPD|DEL|RULE|REFERENCES|TRIGGER|ALL ON <tabela>
FROM <usuário> ou FROM GROUP <grupo>

ALTER USER RENAME <usuário> TO <usuário_novo>
GRANT USAGE ON <schema> TO <usuário>

REVOKE

Usuário cadastrado em dois bancos acumula os privilégios dos dois bancos no segundo banco onde foi adicionado. O ideal é trabalhar com grupos ao invés de usuários.
Exemplos:

```
CREATE GROUP adm;
CREATE GROUP mkt;
CRETE USER <usuario> ENCRYPTED PASSWORD '<senha>' IN GROUP adm;
CRETE USER <usuario2> ENCRYPTED PASSWORD '<senha>' IN GROUP adm;
```

11. Herança

```
CREATE TABLE pessoas (
        id SERIAL PRIMARY KEY,
        nome VARCHAR(30),         enderecos VARCHAR(80)
)
CREATE TABLE pf (
        cpf VARCHAR(14),  rc VARCHAR(20)
) INHERITS (pessoas);

-- inserindo em pessoas
INSERT INTO pessoas (nome, endereco) VALUES ( 'Maria','Rua A');
INSERT INTO pessoas (nome, endereco) VALUES ( 'José','Rua B');

-- inserindo em pf
INSERT INTO pessoas (nome, endereco, cpf, rc) VALUES ( 'Maria','Rua A','123','567');

-- selecionando somente pessoas
SELECT * FROM ONLY pessoas;
```

CAPÍTULO 14. O PostgreSQL 8

Inovações[15]:

- Instalador para Windows (Nativo no Windows, sem CygWin);
- Recuperação "point-in-time", que permite recuperar todos os dados em caso de falha do disco;
- Aprimoramento no gerenciamento de discos, permitindo selecionar os sistemas de arquivos escolhidos para armazenar as base de dados, esquemas, tabelas e índices;
- Melhora no gerenciamento do buffer;
- Os tipos de coluna poderão ser modificados, usando-se "Alter Table";
- Nova versão da linguagem de servidor PLPerl, que agora suporta triggers;
- COPY poderá ler arquivos separados por vírgula;
- Conceito de Tablespace;
- Maior velocidade nas consultas.

Tablespace

Como sabemos, uma das features do PostgreSQL 8 é uma super melhoria no gerenciamento de discos, permitindo assim, selecionar os sistemas de arquivos que iremos armazenar as informações (isso inclui esquemas, tabelas e índices). Em outras palavras, isso significa que eu posso escolher uma determinada PASTA no servidor a qual será utilizada para armazenar uma determinada informação.

O Conceito é muito semelhante ao de tablespace do Oracle, porém, uma tablespace no Oracle é um arquivo localizado em um determinado sistema de arquivos. Já no PostgreSQL, uma tablespace é apenas um lugar no sistema de arquivos onde serão armazenados os objetos (no PostgreSQL, os objetos como tabela e índices são arquivos sem extensão e nomeados numericamente – OID – Object ID).

Este conceito é muito útil para:

Tuning: Podemos criar uma tablespace chamada table_index a qual está vinculada ao secundo HD e nesta tablespace armazenaremos TODOS os índices de nosso banco de dados. Com isso, o acesso à informação é mais rápido e diminui a sobrecarga do HD (e todo o conjunto lógico/físico envolvido) para ler dados e índices;

Gerenciamento de Espaço: Através deste conceito é possível adicionar um outro HD e criar uma nova tablespace para que as novas tabelas e índices sejam colocadas neste novo HD com o intuito de não sofrer com problemas de espaço em disco.

As tablespaces podem ter diversas utilidades, dependendo da necessidade do Administrador do SGDB. Vejamos agora como fazer isso no PostgreSQL 8.0.

15 Adaptado de Carlos Eduardo Smanioto (http://www.sqlmagazine.com.br/Colunistas/smanioto/04_Tablespace.asp).

1) Vamos criar uma pasta chamada DB_SQLMAGAZINE em \db:

2) Agora vamos ao psql. Como ainda não tenho nenhum banco de dados criado no meu PostgreSQL, utilizarei o banco TEMPLATE1

O comando digitado foi:

CREATE DATABASE "DB_SQL" TEMPLATE=template0 TABLESPACE=space_sqlmagazine;

Onde:

- "DB_SQL": Nome do Banco de Dados criado;
- TEMPLATE=template0: Todo banco de dados PostgreSQL necessita de um banco de exemplo (template). Quando não especificado TEMPLATE=, o PostgreSQL utiliza o banco template1. O fato de estar conectado no template1 para criar a tablespace (imagens acima), faz com que este banco não possa ser usado como template, por isso, especifiquei o template0;
- TABLESPACE=space_sqlmagazine: Finalmente, esta é a nova feature do PostgreSQL. Estou definindo aqui ONDE ficarão os objetos do DB_SQL. Neste caso, \db\DB_SQLMAGAZINE.

3) Vamos agora exemplificar a criação de um objeto nesta Tablespace

O comando digitado foi:

CREATE TABLESPACE space_sqlmagazine location '\db\DB _SQLMAGAZINE';

Onde:

- sql_magazine: Nome da Tablespace;
- C:/DB_SQLMAGAZINE é o local que irá receber os dados. Observe que a barra está no padrão UNIX para ser compatível com qualquer Sistema Operacional.

Vejamos o conteúdo:

Veja o número 17234 (fictício), ele é o OID para DB_SQL, ou seja, criamos o banco DB_SQL dentro da pasta DB_SQLMAGAZINE!

Eu poderia fazer agora a criação de uma tabela do DB_SQL em OUTRA TABLESPACE através do comando:

CREATE TABLE "SQL_TABELA" ("ID" serial) TABLESPACE outra_tablespace;

Observe que para qualquer objeto a ser criado, é possível definir onde será armazenado o objeto no sistema de arquivos através do parâmetro TABLESPACE nome_tablespace.

É isso aí pessoal! Espero que tenham gostado desta dica de utilização de uma FEATURE muito interessante que podemos usufruir no PostgreSQL 8.0!

APÊNDICE A. Suporte a data e hora

O PostgreSQL utiliza um analisador heurístico interno para apoiar a entrada da data e da hora. Data e hora são entradas como cadeias de caracteres, e divididas em campos distintos baseado na determinação preliminar do tipo de informação que pode estar contida no campo. Cada campo é interpretado e, em seguida, atribuído um valor numérico, ou é ignorado, ou é rejeitado. O analisador possui tabelas internas de procura para todos os campos textuais, incluindo meses, dias da semana e zonas horárias.

Este apêndice inclui informações relativas às tabelas de procura, e descreve os passos utilizados pelo analisador para decodificar data e hora.

Interpretação da entrada de data e hora

Todos os tipos de data e hora são decodificados utilizando um conjunto comum de rotinas.

Interpretação da entrada de data e hora

Dividir a cadeia de caracteres entrada em partes (tokens), categorizando cada parte como sendo uma cadeia de caracteres, hora, zona horária ou número.

Se a parte numérica contiver dois-pontos (:), então é uma cadeia de caracteres de hora. Incluir todos os dígitos e dois-pontos seguintes.

Se a parte numérica contiver hífen (-), barra (/), ou dois ou mais pontos (.), então é uma cadeia de caracteres de data que pode conter o mês na forma de texto.

Se a parte contiver apenas números, então é apenas um campo único, ou uma data ISO 8601 concatenada (por exemplo, 19990113 para 13 de janeiro de 1999), ou hora (por exemplo, 141516 para 14:15:16).

Se a parte começa por um sinal de mais (+) ou de menos (-), então se trata de uma zona horária ou de um campo especial.

Se a parte for uma cadeia de caracteres, fazer correspondência com os valores possíveis.

Fazer pesquisa binária da parte na tabela de procura tentando correspondência com uma cadeia de caracteres especial (por exemplo, today), um dia da semana (por exemplo, Thursday), um mês (por exemplo, January), ou uma palavra ruído (por exemplo, at, on).

Colocar os valores do campo e a máscara de bit para os campos. Por exemplo, colocar ano, mês e dia para today e, adicionalmente, colocar hora, minuto e segundo para now.

Se não houver correspondência, realizar uma pesquisa binária semelhante na tabela de procura

tentando correspondência com zona horária.

Se não for encontrado, gerar erro.

A parte é um número ou campo numérico.

Havendo mais de 4 dígitos, e nenhum outro campo de data foi lido anteriormente, então interpretar como uma "data concatenada" (por exemplo, 19990118). Se contiver 8 ou 6 dígitos interpretar como ano, mês e dia, enquanto com 7 ou 5 dígitos interpretar como ano e dia do ano, respectivamente.

- Se a parte contiver 3 dígitos e um ano já tiver sido decodificado, então interpretar como dia do ano.
- Se contiver 4 ou 6 dígitos, e o ano já tiver sido lido, então interpretar como hora.
- Se contiver 4 ou mais dígitos, então interpretar como ano.
- Se estiver no modo de data European, e o campo dia ainda não tiver sido lido, e o valor for menor ou igual a 31, então interpretar como dia.
- Se o campo do mês ainda não tiver sido lido, e o valor for menor ou igual a 12, então interpretar como mês.
- Se o campo do dia ainda não tiver sido lido, e o valor for menor ou igual 31, então interpretar como dia.
- Se tiver dois, quatro ou mais dígitos, então interpretar como ano.
- Senão, gerar erro.

Se BC for especificado, tornar o ano negativo e adicionar um para armazenamento interno (Não existe ano zero no Calendário Gregoriano [16] , portanto, numericamente 1BC se torna o ano zero).
Se BC não for especificado, e o campo do ano tiver comprimento de dois dígitos, então adjustar o ano para quatro dígitos. Se o campo for inferior a 70 adicionar 2000, senão adicionar 1900.

Dica: Os anos Gregorianos AD 1-99 [17] podem ser entrados utilizando 4 dígitos com zeros à esquerda (por exemplo, 0099 é AD 99). As versões anteriores do PostgreSQL aceitavam anos com três dígitos e com um dígito, mas a partir da versão 7.0 as regras ficaram mais rigorosas para reduzir a possibilidade de ambigüidade.

Palavras chave para data e hora

Tabela A-1. Abreviaturas dos nomes dos meses

Mês	Abreviatura
Abril	Apr
Agosto	Aug
Dezembro	Dec

16 São dois os Calendário Cristãos ainda em uso no mundo: O Calendário Juliano foi proposto por Sosígenes, astrônomo de Alexandria, e introduzido por Julio César em 45 AC. Foi usado pelas igrejas e países cristãos até o século XVI, quando começou a ser trocado pelo Calendário Gregoriano. Alguns países, como a Grécia e a Rússia, o usaram até o século passado. Ainda é usado por algumas Igrejas Ortodoxas, entre elas a Igreja Russa; O Calendário Gregoriano foi proposto por Aloysius Lilius, astrônomo de Nápoles, e adotado pelo Papa Gregório XIII, seguindo as instruções do Concílio de Trento (1545-1563). O decreto instituindo esse calendário foi publicado em 24 de fevereiro de 1582. - Calendários (N.T.)

17 AD = Annus Domini = DC (N.T.)

Fevereiro	Feb
Janeiro	Jan
Julho	Jul
Junho	Jun
Março	Mar
Novembro	Nov
Outubro	Oct
Setembro	Sep, Sept

Nota: O mês de maio (May) não possui abreviatura explícita, por razões óbvias. [18]

Tabela A-2. Abreviatura dos dias da semana

Dia	Abreviatura
Domingo	Sun
Segunda	Mon
Terça	Tue, Tues
Quarta	Wed, Weds
Quinta	Thu, Thur, Thurs
Sexta	Fri
Sábado	Sat

Tabela A-3. Campos modificadores de data e hora

Identificador	Descrição
ABSTIME	Palavra chave ignorada
AM	Hora antes das 12:00
AT	Palavra chave ignorada
JULIAN, JD, J	O próximo campo é dia Juliano
ON	Palavra chave ignorada
PM	Hora após ou às 12:00
T	O próximo campo é hora

A palavra chave ABSTIME é ignorada por razões históricas; nas versões muito antigas do PostgreSQL campos inválidos do tipo abstime eram emitidos como Invalid Abstime. Entretanto, este não é mais o caso, e esta palavra chave provavelmente será retirada em uma versão futura.

A Tabela A-4 mostra as abreviaturas de zona horária reconhecidas pelo PostgreSQL. O PostgreSQL possui informação tabular interna para decodificar a zona horária, porque não existe uma interface padrão do sistema operacional para fornecer acesso a informações gerais para cruzamento de zona horária. Entretanto, o sistema operacional é utilizado para fornecer informação de zona horária para a saída.

A tabela é organizada pelo deslocamento da zona horária relativo à UTC, em vez de alfabeticamente; tem por finalidade facilitar a correspondência entre a utilização local e as abreviaturas reconhecidas nos casos em que forem diferentes.

18 Razões óbvias somente em inglês, obviamente! (N.T.)

Tabela A-4. Abreviaturas de zona horária

Zona horária	Deslocamento da UTC	Descrição
NZDT	+13:00	New Zealand Daylight Time
IDLE	+12:00	International Date Line, East
NZST	+12:00	New Zealand Standard Time
NZT	+12:00	New Zealand Time
AESST	+11:00	Australia Eastern Summer Standard Time
ACSST	+10:30	Central Australia Summer Standard Time
CADT	+10:30	Central Australia Daylight Savings Time
SADT	+10:30	South Australian Daylight Time
AEST	+10:00	Australia Eastern Standard Time
EAST	+10:00	East Australian Standard Time
GST	+10:00	Guam Standard Time, USSR Zone 9
LIGT	+10:00	Melbourne, Australia
SAST	+09:30	South Australia Standard Time
CAST	+09:30	Central Australia Standard Time
AWSST	+09:00	Australia Western Summer Standard Time
JST	+09:00	Japan Standard Time,USSR Zone 8
KST	+09:00	Korea Standard Time
MHT	+09:00	Kwajalein Time
WDT	+09:00	West Australian Daylight Time
MT	+08:30	Moluccas Time
AWST	+08:00	Australia Western Standard Time
CCT	+08:00	China Coastal Time
WADT	+08:00	West Australian Daylight Time
WST	+08:00	West Australian Standard Time
JT	+07:30	Java Time
ALMST	+07:00	Almaty Summer Time
WAST	+07:00	West Australian Standard Time
CXT	+07:00	Christmas (Island) Time
MMT	+06:30	Myannar Time
ALMT	+06:00	Almaty Time
MAWT	+06:00	Mawson (Antarctica) Time
IOT	+05:00	Indian Chagos Time
MVT	+05:00	Maldives Island Time
TFT	+05:00	Kerguelen Time
AFT	+04:30	Afganistan Time
EAST	+04:00	Antananarivo Savings Time
MUT	+04:00	Mauritius Island Time
RET	+04:00	Reunion Island Time
SCT	+04:00	Mahe Island Time
IRT, IT	+03:30	Iran Time
EAT	+03:00	Antananarivo, Comoro Time
BT	+03:00	Baghdad Time
EETDST	+03:00	Eastern Europe Daylight Savings Time
HMT	+03:00	Hellas Mediterranean Time (?)

BDST	+02:00	British Double Standard Time
CEST	+02:00	Central European Savings Time
CETDST	+02:00	Central European Daylight Savings Time
EET	+02:00	Eastern Europe, USSR Zone 1
FWT	+02:00	French Winter Time
IST	+02:00	Israel Standard Time
MEST	+02:00	Middle Europe Summer Time
METDST	+02:00	Middle Europe Daylight Time
SST	+02:00	Swedish Summer Time
BST	+01:00	British Summer Time
CET	+01:00	Central European Time
DNT	+01:00	Dansk Normal Tid
FST	+01:00	French Summer Time
MET	+01:00	Middle Europe Time
MEWT	+01:00	Middle Europe Winter Time
MEZ	+01:00	Middle Europe Zone
NOR	+01:00	Norway Standard Time
SET	+01:00	Seychelles Time
SWT	+01:00	Swedish Winter Time
WETDST	+01:00	Western Europe Daylight Savings Time
GMT	+00:00	Greenwich Mean Time
UT	+00:00	Universal Time
UTC	+00:00	Universal Time, Coordinated
Z	+00:00	Same as UTC
ZULU	+00:00	Same as UTC
WET	+00:00	Western Europe
WAT	-01:00	West Africa Time
NDT	-02:30	Newfoundland Daylight Time
ADT	-03:00	Atlantic Daylight Time
AWT	-03:00	(unknown)
NFT	-03:30	Newfoundland Standard Time
NST	-03:30	Newfoundland Standard Time
AST	-04:00	Atlantic Standard Time (Canada)
ACST	-04:00	Atlantic/Porto Acre Summer Time
ACT	-05:00	Atlantic/Porto Acre Standard Time
EDT	-04:00	Eastern Daylight Time
CDT	-05:00	Central Daylight Time
EST	-05:00	Eastern Standard Time
CST	-06:00	Central Standard Time
MDT	-06:00	Mountain Daylight Time
MST	-07:00	Mountain Standard Time
PDT	-07:00	Pacific Daylight Time
AKDT	-08:00	Alaska Daylight Time
PST	-08:00	Pacific Standard Time
YDT	-08:00	Yukon Daylight Time
AKST	-09:00	Alaska Standard Time
HDT	-09:00	Hawaii/Alaska Daylight Time
YST	-09:00	Yukon Standard Time

MART	-09:30	Marquesas Time
AHST	-10:00	Alaska-Hawaii Standard Time
HST	-10:00	Hawaii Standard Time
CAT	-10:00	Central Alaska Time
NT	-11:00	Nome Time
IDLW	-12:00	International Date Line, West

Zona horárias da Austrália. Existem três nomes em conflito entre as zonas horárias da Austrália e as zonas horárias utilizadas normalmente na América do Sul e do Norte: ACST, CST e EST. Se a opção em tempo de execução AUSTRALIAN_TIMEZONES estiver definida como verdade então ACST, CST, EST e SAT são interpretados como nomes de zonas horárias da Austrália, conforme mostrado na Tabela A-5. Se for falso (que é o padrão), então ACST, CST e EST são tomados como nomes de zonas horárias das Américas, e SAT é interpretado como palavra ruído indicando Sábado.

Tabela A-5. Abreviaturas das zonas horárias da Austrália

Zona horária	Deslocamento da UTC	Descrição
ACST	+09:30	Central Australia Standard Time
CST	+10:30	Australian Central Standard Time
EST	+10:00	Australian Eastern Standard Time
SAT	+09:30	South Australian Standard Time

História das unidades

Nota: Contribuição de José Soares (<jose@sferacarta.com>)

O Dia Juliando foi inventado pelo estudioso francês Joseph Justus Scaliger (1540-1609) e, provavelmente, recebeu este nome devido ao pai do Scaliger, o estudioso italiano Julius Caesar Scaliger (1484-1558). Os astrônomos têm utilizado o período Juliano para atribuir um número único para cada dia a partir de 1 de janeiro de 4713 AC. Este é o tão falado Dia Juliano (DJ). DJ 0 designa as 24 horas desde o meio-dia UTC de 1 de janeiro de 4713 AC até meio-dia UTC de 2 de janeiro de 4713 AC.

O "Dia Juliano" é diferente da "Data Juliana". A Data Juliana diz respeito ao Calendário Juliano, que foi introduzido por Julius Caesar em 45 AC. Ficou em uso corrente até 1582, quando os países começaram a mudar para o Calendário Gregoriano. No Calendário Juliano, o ano tropical é aproximado como 365 1/4 dias = 365.25 dias. Isto ocasiona um erro de aproximadamente 1 dia a cada 128 anos.

O erro acumulado fez o Papa Gregório XIII reformar o calendário de acordo com as instruções do Concílio de Trento. No Calendário Gregoriano o ano tropical é aproximado como 365 + 97 / 400 days = 365.2425 dias. Portanto, leva aproximadamente 3300 anos para o ano tropical se deslocar um dia em relação ao Calendário Gregoriano.

A aproximação 365+97/400 é obtida colocando 97 anos bissextos a cada 400 anos, utilizando as seguintes regras:

Cada ano divisível por 4 é um ano bissexto.

Entretanto, cada ano divisível por 100 não é um ano bissexto.

Entretanto, todo ano divisível por 400 é um ano bissexto sempre.

Portanto, 1700, 1800, 1900, 2100 e 2200 não são anos bissextos. Porém, 1600, 2000 e 2400 são anos bissextos. Contrapondo, no Calendário Juliano antigo todos os anos divisíveis por 4 eram bissextos.

A Bula Papal de fevereiro de 1582 decretou que 10 dias deveriam ser excluídos de outubro de 1582, fazendo com que 15 de outubro viesse imediatamente após 4 de outubro. Foi respeitado na Itália, Polônia, Portugal e Espanha. Outros países católicos seguiram logo após, mas os países protestantantes relutaram em seguir, e os países ortodoxos gregos não mudaram até o início do século 20. A reforma foi seguida pela Inglaterra e seus domínios (incluido o que agora são os EUA) em 1752. Assim, 2 de setembro de 1752 foi seguido por 14 de setembro de 1752. Por isso, nos sistemas Unix, o programa cal produz a seguinte informação:

```
$ cal 9 1752
   setembro 1752
Do Se Te Qu Qu Se Sá
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

Nota: O padrão SQL declara que "Dentro da definição do 'literal data/hora', os 'valores de data/hora' são restritos pelas regras naturais para data e hora de acordo com o Calendário Gregoriano". As datas entre 1752-09-03 e 1752-09-13, embora excluídas em alguns países pela Bula Papal, estão em conformidade com as "regras naturais" sendo, portanto, datas válidas.

Calendários diferentes foram elaborados em muitas partes do mundo, vários anteriores as sistema Gregoriano. Por exemplo, o início do Calendário Chinês remonta ao século 14 AC. Diz a lenda que o Imperador Huangdi inventou o calendário em 2637 AC. A República Popular da China utiliza o Calendário Gregoriano para as finalidades civis. O Calendário Chinês é utilizado para determinar as festividades.

APÊNDICE B. Palavras chave do SQL

A Tabela B-1 lista todas os termos (tokens) que são palavras chave do SQL no PostgreSQL 7.3.4.

O padrão SQL faz distinção entre palavras chave reservadas e não reservadas. De acordo com o padrão, as palavras chave reservadas são as únicas palavras chave reais; nunca podem ser utilizadas como identificadores. As palavras chave não reservadas somente possuem significado especial em alguns contextos, sendo possível utilizá-las como identificador em outros contextos. Em sua maioria, as palavras chave não reservadas são, na verdade, nomes de tabelas nativas e funções especificadas pelo padrão SQL. Essencialmente, o conceito de palavra chave não reservada existe apenas para declarar a associação desta palavra com um significado predefinido em alguns contextos.

A vida do analisador do PostgreSQL é um pouco mais complicada. Existem várias classes diferentes de termos, indo desde aqueles que nunca podem ser utilizados como identificador, até aqueles que não possuem nenhum status especial no analisador se comparado com um identificador comum (Geralmente, este último é o caso das funções especificadas pelo padrão SQL). Mesmo as palavras chave reservadas não são totalmente reservadas no PostgreSQL, sendo possível utilizá-las como títulos de colunas (por exemplo, SELECT 55 AS CHECK, embora CHECK seja uma palavra chave reservada).

Na coluna PostgreSQL da Tabela B-1 são classificadas como "não reservadas" as palavras chave explicitamente reconhecidas pelo analisador mas permitidas na maioria ou em todos os contextos onde um identificador é esperado. Existem algumas palavras chave não reservadas que não podem ser utilizadas como nome de função ou de tipo de dado, estando devidamente indicado (Em sua maioria, estas palavras representam funções nativas ou tipos de dado com sintaxe especial. A função ou o tipo continuam disponíveis, mas não podem ser redefinidos pelo usuário). Na coluna "reservadas" estão os termos permitidos apenas como rótulos de coluna utilizando o "AS" (e, talvez, em muito poucos outros contextos). Algumas palavras chave reservadas são permitidas como nome de função; isto também está indicado na tabela.

Como regra geral, acontecendo erros indevidos do analisador em comandos contendo como identificador qualquer uma das palavras chave listadas, deve ser tentado colocar o identificador entre aspas para ver se o problema desaparece.

Antes de estudar a Tabela B-1, é importante compreender o fato de uma palavra chave não ser reservada no PostgreSQL não significa que a funcionalidade associada a esta palavra chave não está implementada. Inversamente, a presença de uma palavra chave não indica a implementação da funcionalidade.

Tabela B-1. palavras chave do SQL

Palavra chave	PostgreSQL	SQL 99	SQL 92
ABORT	não-reservada		
ABS		não-reservada	
ABSOLUTE	não-reservada	reservada	reservada
ACCESS	não-reservada		
ACTION	não-reservada	reservada	reservada
ADA		não-reservada	não-reservada
ADD	não-reservada	reservada	reservada
ADMIN		reservada	
AFTER	não-reservada	reservada	
AGGREGATE	não-reservada	reservada	
ALIAS		reservada	
ALL	reservada	reservada	reservada
ALLOCATE		reservada	reservada
ALTER	não-reservada	reservada	reservada
ANALYSE	reservada		
ANALYZE	reservada		
AND	reservada	reservada	reservada
ANY	reservada	reservada	reservada
ARE		reservada	reservada
ARRAY	reservada	reservada	
AS	reservada	reservada	reservada
ASC	reservada	reservada	reservada
ASENSITIVE		não-reservada	
ASSERTION	não-reservada	reservada	reservada
ASSIGNMENT	não-reservada	não-reservada	
ASYMMETRIC		não-reservada	
AT	não-reservada	reservada	reservada
ATOMIC		não-reservada	
AUTHORIZATION	reservada (pode ser função)	reservada	reservada
AVG		não-reservada	reservada
BACKWARD	não-reservada		
BEFORE	não-reservada	reservada	
BEGIN	não-reservada	reservada	reservada
BETWEEN	reservada (pode ser função)	não-reservada	reservada
BIGINT	não-reservada (não pode ser função ou tipo)		
BINARY	reservada (pode ser função)	reservada	
BIT	não-reservada (não pode ser função ou tipo)	reservada	reservada
BITVAR		não-reservada	
BIT_LENGTH		não-reservada	reservada
BLOB		reservada	

Palavra chave	PostgreSQL	SQL 99	SQL 92
BOOLEAN	não-reservada (não pode ser função ou tipo)	reservada	
BOTH	reservada	reservada	reservada
BREADTH		reservada	
BY	não-reservada	reservada	reservada
C		não-reservada	não-reservada
CACHE	não-reservada		
CALL		reservada	
CALLED	não-reservada	não-reservada	
CARDINALITY		não-reservada	
CASCADE	não-reservada	reservada	reservada
CASCADED		reservada	reservada
CASE	reservada	reservada	reservada
CAST	reservada	reservada	reservada
CATALOG		reservada	reservada
CATALOG_NAME		não-reservada	não-reservada
CHAIN	não-reservada	não-reservada	
CHAR	não-reservada (não pode ser função ou tipo)	reservada	reservada
CHARACTER	não-reservada (não pode ser função ou tipo)	reservada	reservada
CHARACTERISTICS	não-reservada		
CHARACTER_LENGTH		não-reservada	reservada
CHARACTER_SET_CATALOG		não-reservada	não-reservada
CHARACTER_SET_NAME		não-reservada	não-reservada
CHARACTER_SET_SCHEMA		não-reservada	não-reservada
CHAR_LENGTH		não-reservada	reservada
CHECK	reservada	reservada	reservada
CHECKED		não-reservada	
CHECKPOINT	não-reservada		
CLASS	não-reservada	reservada	
CLASS_ORIGIN		não-reservada	não-reservada
CLOB		reservada	
CLOSE	não-reservada	reservada	reservada
CLUSTER	não-reservada		
COALESCE	não-reservada (não pode ser função ou tipo)	não-reservada	reservada
COBOL		não-reservada	não-reservada
COLLATE	reservada	reservada	reservada
COLLATION		reservada	reservada
COLLATION_CATALOG		não-reservada	não-reservada
COLLATION_NAME		não-reservada	não-reservada
COLLATION_SCHEMA		não-reservada	não-reservada
COLUMN	reservada	reservada	reservada

Palavra chave	PostgreSQL	SQL 99	SQL 92
COLUMN_NAME		não-reservada	não-reservada
COMMAND_FUNCTION		não-reservada	não-reservada
COMMAND_FUNCTION_CODE		não-reservada	
COMMENT	não-reservada		
COMMIT	não-reservada	reservada	reservada
COMMITTED	não-reservada	não-reservada	não-reservada
COMPLETION		reservada	
CONDITION_NUMBER		não-reservada	não-reservada
CONNECT		reservada	reservada
CONNECTION		reservada	reservada
CONNECTION_NAME		não-reservada	não-reservada
CONSTRAINT	reservada	reservada	reservada
CONSTRAINTS	não-reservada	reservada	reservada
CONSTRAINT_CATALOG		não-reservada	não-reservada
CONSTRAINT_NAME		não-reservada	não-reservada
CONSTRAINT_SCHEMA		não-reservada	não-reservada
CONSTRUCTOR		reservada	
CONTAINS		não-reservada	
CONTINUE		reservada	reservada
CONVERSION	não-reservada		
CONVERT	não-reservada (não pode ser função ou tipo)	não-reservada	reservada
COPY	não-reservada		
CORRESPONDING		reservada	reservada
COUNT		não-reservada	reservada
CREATE	reservada	reservada	reservada
CREATEDB	não-reservada		
CREATEUSER	não-reservada		
CROSS	reservada (pode ser função)	reservada	reservada
CUBE		reservada	
CURRENT		reservada	reservada
CURRENT_DATE	reservada	reservada	reservada
CURRENT_PATH		reservada	
CURRENT_ROLE		reservada	
CURRENT_TIME	reservada	reservada	reservada
CURRENT_TIMESTAMP	reservada	reservada	reservada
CURRENT_USER	reservada	reservada	reservada
CURSOR	não-reservada	reservada	reservada
CURSOR_NAME		não-reservada	não-reservada
CYCLE	não-reservada	reservada	
DATA		reservada	não-reservada
DATABASE	não-reservada		
DATE		reservada	reservada
DATETIME_INTERVAL_CODE		não-reservada	não-reservada

Palavra chave	PostgreSQL	SQL 99	SQL 92
DATETIME_INTERVAL_PRECISION		não-reservada	não-reservada
DAY	não-reservada	reservada	reservada
DEALLOCATE	não-reservada	reservada	reservada
DEC	não-reservada (não pode ser função ou tipo)	reservada	reservada
DECIMAL	não-reservada (não pode ser função ou tipo)	reservada	reservada
DECLARE	não-reservada	reservada	reservada
DEFAULT	reservada	reservada	reservada
DEFAULTS	não-reservada		
DEFERRABLE	reservada	reservada	reservada
DEFERRED	não-reservada	reservada	reservada
DEFINED		não-reservada	
DEFINER	não-reservada	não-reservada	
DELETE	não-reservada	reservada	reservada
DELIMITER	não-reservada		
DELIMITERS	não-reservada		
DEPTH		reservada	
DEREF		reservada	
DESC	reservada	reservada	reservada
DESCRIBE		reservada	reservada
DESCRIPTOR		reservada	reservada
DESTROY		reservada	
DESTRUCTOR		reservada	
DETERMINISTIC		reservada	
DIAGNOSTICS		reservada	reservada
DICTIONARY		reservada	
DISCONNECT		reservada	reservada
DISPATCH		não-reservada	
DISTINCT	reservada	reservada	reservada
DO	reservada		
DOMAIN	não-reservada	reservada	reservada
DOUBLE	não-reservada	reservada	reservada
DROP	não-reservada	reservada	reservada
DYNAMIC		reservada	
DYNAMIC_FUNCTION		não-reservada	não-reservada
DYNAMIC_FUNCTION_CODE		não-reservada	
EACH	não-reservada	reservada	
ELSE	reservada	reservada	reservada
ENCODING	não-reservada		
ENCRYPTED	não-reservada		
END	reservada	reservada	reservada
END-EXEC		reservada	reservada
EQUALS		reservada	

Palavra chave	PostgreSQL	SQL 99	SQL 92
ESCAPE	não-reservada	reservada	reservada
EVERY		reservada	
EXCEPT	reservada	reservada	reservada
EXCEPTION		reservada	reservada
EXCLUDING	não-reservada		
EXCLUSIVE	não-reservada		
EXEC		reservada	reservada
EXECUTE	não-reservada	reservada	reservada
EXISTING		não-reservada	
EXISTS	não-reservada (não pode ser função ou tipo)	não-reservada	reservada
EXPLAIN	não-reservada		
EXTERNAL	não-reservada	reservada	reservada
EXTRACT	não-reservada (não pode ser função ou tipo)	não-reservada	reservada
FALSE	reservada	reservada	reservada
FETCH	não-reservada	reservada	reservada
FINAL		não-reservada	
FIRST	não-reservada	reservada	reservada
FLOAT	não-reservada (não pode ser função ou tipo)	reservada	reservada
FOR	reservada	reservada	reservada
FORCE	não-reservada		
FOREIGN	reservada	reservada	reservada
FORTRAN		não-reservada	não-reservada
FORWARD	não-reservada		
FOUND		reservada	reservada
FREE		reservada	
FREEZE	reservada (pode ser função)		
FROM	reservada	reservada	reservada
FULL	reservada (pode ser função)	reservada	reservada
FUNCTION	não-reservada	reservada	
G		não-reservada	
GENERAL		reservada	
GENERATED		não-reservada	
GET		reservada	reservada
GLOBAL	não-reservada	reservada	reservada
GO		reservada	reservada
GOTO		reservada	reservada
GRANT	reservada	reservada	reservada
GRANTED		não-reservada	
GROUP	reservada	reservada	reservada
GROUPING		reservada	
HANDLER	não-reservada		
HAVING	reservada	reservada	reservada
HIERARCHY		não-reservada	

Palavra chave	PostgreSQL	SQL 99	SQL 92
HOLD	não-reservada	não-reservada	
HOST		reservada	
HOUR	não-reservada	reservada	reservada
IDENTITY		reservada	reservada
IGNORE		reservada	
ILIKE	reservada (pode ser função)		
IMMEDIATE	não-reservada	reservada	reservada
IMMUTABLE	não-reservada		
IMPLEMENTATION		não-reservada	
IMPLICIT	não-reservada		
IN	reservada (pode ser função)	reservada	reservada
INCLUDING	não-reservada		
INCREMENT	não-reservada		
INDEX	não-reservada		
INDICATOR		reservada	reservada
INFIX		não-reservada	
INHERITS	não-reservada		
INITIALIZE		reservada	
INITIALLY	reservada	reservada	reservada
INNER	reservada (pode ser função)	reservada	reservada
INOUT	não-reservada	reservada	
INPUT	não-reservada	reservada	reservada
INSENSITIVE	não-reservada	não-reservada	reservada
INSERT	não-reservada	reservada	reservada
INSTANCE		não-reservada	
INSTANTIABLE		não-reservada	
INSTEAD	não-reservada		
INT	não-reservada (não pode ser função ou tipo)	reservada	reservada
INTEGER	não-reservada (não pode ser função ou tipo)	reservada	reservada
INTERSECT	reservada	reservada	reservada
INTERVAL	não-reservada (não pode ser função ou tipo)	reservada	reservada
INTO	reservada	reservada	reservada
INVOKER	não-reservada	não-reservada	
IS	reservada (pode ser função)	reservada	reservada
ISNULL	reservada (pode ser função)		
ISOLATION	não-reservada	reservada	reservada
ITERATE		reservada	
JOIN	reservada (pode ser função)	reservada	reservada
K		não-reservada	
KEY	não-reservada	reservada	reservada
KEY_MEMBER		não-reservada	
KEY_TYPE		não-reservada	
LANCOMPILER	não-reservada		

Palavra chave	PostgreSQL	SQL 99	SQL 92
LANGUAGE	não-reservada	reservada	reservada
LARGE		reservada	
LAST	não-reservada	reservada	reservada
LATERAL		reservada	
LEADING	reservada	reservada	reservada
LEFT	reservada (pode ser função)	reservada	reservada
LENGTH		não-reservada	não-reservada
LESS		reservada	
LEVEL	não-reservada	reservada	reservada
LIKE	reservada (pode ser função)	reservada	reservada
LIMIT	reservada	reservada	
LISTEN	não-reservada		
LOAD	não-reservada		
LOCAL	não-reservada	reservada	reservada
LOCALTIME	reservada	reservada	
LOCALTIMESTAMP	reservada	reservada	
LOCATION	não-reservada		
LOCATOR		reservada	
LOCK	não-reservada		
LOWER		não-reservada	reservada
M		não-reservada	
MAP		reservada	
MATCH	não-reservada	reservada	reservada
MAX		não-reservada	reservada
MAXVALUE	não-reservada		
MESSAGE_LENGTH		não-reservada	não-reservada
MESSAGE_OCTET_LENGTH		não-reservada	não-reservada
MESSAGE_TEXT		não-reservada	não-reservada
METHOD		não-reservada	
MIN		não-reservada	reservada
MINUTE	não-reservada	reservada	reservada
MINVALUE	não-reservada		
MOD		não-reservada	
MODE	não-reservada		
MODIFIES		reservada	
MODIFY		reservada	
MODULE		reservada	reservada
MONTH	não-reservada	reservada	reservada
MORE		não-reservada	não-reservada
MOVE	não-reservada		
MUMPS		não-reservada	não-reservada
NAME		não-reservada	não-reservada
NAMES	não-reservada	reservada	reservada
NATIONAL	não-reservada	reservada	reservada

Palavra chave	PostgreSQL	SQL 99	SQL 92
NATURAL	reservada (pode ser função)	reservada	reservada
NCHAR	não-reservada (não pode ser função ou tipo)	reservada	reservada
NCLOB		reservada	
NEW	reservada	reservada	
NEXT	não-reservada	reservada	reservada
NO	não-reservada	reservada	reservada
NOCREATEDB	não-reservada		
NOCREATEUSER	não-reservada		
NONE	não-reservada (não pode ser função ou tipo)	reservada	
NOT	reservada	reservada	reservada
NOTHING	não-reservada		
NOTIFY	não-reservada		
NOTNULL	reservada (pode ser função)		
NULL	reservada	reservada	reservada
NULLABLE		não-reservada	não-reservada
NULLIF	não-reservada (não pode ser função ou tipo)	não-reservada	reservada
NUMBER		não-reservada	não-reservada
NUMERIC	não-reservada (não pode ser função ou tipo)	reservada	reservada
OBJECT		reservada	
OCTET_LENGTH		não-reservada	reservada
OF	não-reservada	reservada	reservada
OFF	reservada	reservada	
OFFSET	reservada		
OIDS	não-reservada		
OLD	reservada	reservada	
ON	reservada	reservada	reservada
ONLY	reservada	reservada	reservada
OPEN		reservada	reservada
OPERATION		reservada	
OPERATOR	não-reservada		
OPTION	não-reservada	reservada	reservada
OPTIONS		não-reservada	
OR	reservada	reservada	reservada
ORDER	reservada	reservada	reservada
ORDINALITY		reservada	
OUT	não-reservada	reservada	
OUTER	reservada (pode ser função)	reservada	reservada
OUTPUT		reservada	reservada
OVERLAPS	reservada (pode ser função)	não-reservada	reservada
OVERLAY	não-reservada (não pode ser função ou tipo)	não-reservada	
OVERRIDING		não-reservada	

Palavra chave	PostgreSQL	SQL 99	SQL 92
OWNER	não-reservada		
PAD		reservada	reservada
PARAMETER		reservada	
PARAMETERS		reservada	
PARAMETER_MODE		não-reservada	
PARAMETER_NAME		não-reservada	
PARAMETER_ORDINAL_POSITION		não-reservada	
PARAMETER_SPECIFIC_CATALOG		não-reservada	
PARAMETER_SPECIFIC_NAME		não-reservada	
PARAMETER_SPECIFIC_SCHEMA		não-reservada	
PARTIAL	não-reservada	reservada	reservada
PASCAL		não-reservada	não-reservada
PASSWORD	não-reservada		
PATH	não-reservada	reservada	
PENDANT	não-reservada		
PLACING	reservada		
PLI		não-reservada	não-reservada
POSITION	não-reservada (não pode ser função ou tipo)	não-reservada	reservada
POSTFIX		reservada	
PRECISION	não-reservada	reservada	reservada
PREFIX		reservada	
PREORDER		reservada	
PREPARE	não-reservada	reservada	reservada
PRESERVE	não-reservada	reservada	reservada
PRIMARY	reservada	reservada	reservada
PRIOR	não-reservada	reservada	reservada
PRIVILEGES	não-reservada	reservada	reservada
PROCEDURAL	não-reservada		
PROCEDURE	não-reservada	reservada	reservada
PUBLIC		reservada	reservada
READ	não-reservada	reservada	reservada
READS		reservada	
REAL	não-reservada (não pode ser função ou tipo)	reservada	reservada
RECHECK	não-reservada		
RECURSIVE		reservada	
REF		reservada	
REFERENCES	reservada	reservada	reservada
REFERENCING		reservada	
REINDEX	não-reservada		
RELATIVE	não-reservada	reservada	reservada

Palavra chave	PostgreSQL	SQL 99	SQL 92
RENAME	não-reservada		
REPEATABLE		não-reservada	não-reservada
REPLACE	não-reservada		
RESET	não-reservada		
RESTART	não-reservada		
RESTRICT	não-reservada	reservada	reservada
RESULT		reservada	
RETURN		reservada	
RETURNED_LENGTH		não-reservada	não-reservada
RETURNED_OCTET_LENGTH		não-reservada	não-reservada
RETURNED_SQLSTATE		não-reservada	não-reservada
RETURNS	não-reservada	reservada	
REVOKE	não-reservada	reservada	reservada
RIGHT	reservada (pode ser função)	reservada	reservada
ROLE		reservada	
ROLLBACK	não-reservada	reservada	reservada
ROLLUP		reservada	
ROUTINE		reservada	
ROUTINE_CATALOG		não-reservada	
ROUTINE_NAME		não-reservada	
ROUTINE_SCHEMA		não-reservada	
ROW	não-reservada (não pode ser função ou tipo)	reservada	
ROWS	não-reservada	reservada	reservada
ROW_COUNT		não-reservada	não-reservada
RULE	não-reservada		
SAVEPOINT		reservada	
SCALE		não-reservada	não-reservada
SCHEMA	não-reservada	reservada	reservada
SCHEMA_NAME		não-reservada	não-reservada
SCOPE		reservada	
SCROLL	não-reservada	reservada	reservada
SEARCH		reservada	
SECOND	não-reservada	reservada	reservada
SECTION		reservada	reservada
SECURITY	não-reservada	não-reservada	
SELECT	reservada	reservada	reservada
SELF		não-reservada	
SENSITIVE		não-reservada	
SEQUENCE	não-reservada	reservada	
SERIALIZABLE	não-reservada	não-reservada	não-reservada
SERVER_NAME		não-reservada	não-reservada
SESSION	não-reservada	reservada	reservada
SESSION_USER	reservada	reservada	reservada
SET	não-reservada	reservada	reservada

Palavra chave	PostgreSQL	SQL 99	SQL 92
SETOF	não-reservada (não pode ser função ou tipo)		
SETS		reservada	
SHARE	não-reservada		
SHOW	não-reservada		
SIMILAR	reservada (pode ser função)	não-reservada	
SIMPLE	não-reservada	não-reservada	
SIZE		reservada	reservada
SMALLINT	não-reservada (não pode ser função ou tipo)	reservada	reservada
SOME	reservada	reservada	reservada
SOURCE		não-reservada	
SPACE		reservada	reservada
SPECIFIC		reservada	
SPECIFICTYPE		reservada	
SPECIFIC_NAME		não-reservada	
SQL		reservada	reservada
SQLCODE			reservada
SQLERROR			reservada
SQLEXCEPTION		reservada	
SQLSTATE		reservada	reservada
SQLWARNING		reservada	
STABLE	não-reservada		
START	não-reservada	reservada	
STATE		reservada	
STATEMENT	não-reservada	reservada	
STATIC		reservada	
STATISTICS	não-reservada		
STDIN	não-reservada		
STDOUT	não-reservada		
STORAGE	não-reservada		
STRICT	não-reservada		
STRUCTURE		reservada	
STYLE		não-reservada	
SUBCLASS_ORIGIN		não-reservada	não-reservada
SUBLIST		não-reservada	
SUBSTRING	não-reservada (não pode ser função ou tipo)	não-reservada	reservada
SUM		não-reservada	reservada
SYMMETRIC		não-reservada	
SYSID	não-reservada		
SYSTEM		não-reservada	
SYSTEM_USER		reservada	reservada
TABLE	reservada	reservada	reservada
TABLE_NAME		não-reservada	não-reservada
TEMP	não-reservada		

Palavra chave	PostgreSQL	SQL 99	SQL 92
TEMPLATE	não-reservada		
TEMPORARY	não-reservada	reservada	reservada
TERMINATE		reservada	
THAN		reservada	
THEN	reservada	reservada	reservada
TIME	não-reservada (não pode ser função ou tipo)	reservada	reservada
TIMESTAMP	não-reservada (não pode ser função ou tipo)	reservada	reservada
TIMEZONE_HOUR		reservada	reservada
TIMEZONE_MINUTE		reservada	reservada
TO	reservada	reservada	reservada
TOAST	não-reservada		
TRAILING	reservada	reservada	reservada
TRANSACTION	não-reservada	reservada	reservada
TRANSACTIONS_COMMITTED		não-reservada	
TRANSACTIONS_ROLLED_BACK		não-reservada	
TRANSACTION_ACTIVE		não-reservada	
TRANSFORM		não-reservada	
TRANSFORMS		não-reservada	
TRANSLATE		não-reservada	reservada
TRANSLATION		reservada	reservada
TREAT	não-reservada (não pode ser função ou tipo)	reservada	
TRIGGER	não-reservada	reservada	
TRIGGER_CATALOG		não-reservada	
TRIGGER_NAME		não-reservada	
TRIGGER_SCHEMA		não-reservada	
TRIM	não-reservada (não pode ser função ou tipo)	não-reservada	reservada
TRUE	reservada	reservada	reservada
TRUNCATE	não-reservada		
TRUSTED	não-reservada		
TYPE	não-reservada	não-reservada	não-reservada
UNCOMMITTED		não-reservada	não-reservada
UNDER		reservada	
UNENCRYPTED	não-reservada		
UNION	reservada	reservada	reservada
UNIQUE	reservada	reservada	reservada
UNKNOWN	não-reservada	reservada	reservada
UNLISTEN	não-reservada		
UNNAMED		não-reservada	não-reservada
UNNEST		reservada	
UNTIL	não-reservada		

Palavra chave	PostgreSQL	SQL 99	SQL 92
UPDATE	não-reservada	reservada	reservada
UPPER		não-reservada	reservada
USAGE	não-reservada	reservada	reservada
USER	reservada	reservada	reservada
USER_DEFINED_TYPE_CATALOG		não-reservada	
USER_DEFINED_TYPE_NAME		não-reservada	
USER_DEFINED_TYPE_SCHEMA		não-reservada	
USING	reservada	reservada	reservada
VACUUM	não-reservada		
VALID	não-reservada		
VALIDATOR	não-reservada		
VALUE		reservada	reservada
VALUES	não-reservada	reservada	reservada
VARCHAR	não-reservada (não pode ser função ou tipo)	reservada	reservada
VARIABLE		reservada	
VARYING	não-reservada	reservada	reservada
VERBOSE	reservada (pode ser função)		
VERSION	não-reservada		
VIEW	não-reservada	reservada	reservada
VOLATILE	não-reservada		
WHEN	reservada	reservada	reservada
WHENEVER		reservada	reservada
WHERE	reservada	reservada	reservada
WITH	não-reservada	reservada	reservada
WITHOUT	não-reservada	reservada	
WORK	não-reservada	reservada	reservada
WRITE	não-reservada	reservada	reservada
YEAR	não-reservada	reservada	reservada
ZONE	não-reservada	reservada	reservada

APÊNDICE C. Conformidade com o SQL

Esta seção tem por objetivo dar uma visão geral até que ponto o PostgreSQL está em conformidade com o padrão SQL. A conformidade total com o padrão, ou a declaração completa relativa à conformidade com o padrão, é complicada e não é particularmente útil. Portanto, esta seção só pode mostrar uma visão geral.

O nome formal do padrão SQL é ISO/IEC 9075 "Database Language SQL". A versão revisada do padrão é liberada de tempos em tempos; a mais recente surgiu em 1999. Esta versão é referida como ISO/IEC 9075:1999 ou, informalmente, como SQL99. A versão anterior a esta foi a SQL92. O desenvolvimento do PostgreSQL tenta manter a conformidade com a última versão oficial do padrão, quando esta conformidade não contradiz as funcionalidades tradicionais ou o senso comum. Enquanto este texto é escrito, uma votação se encontra em andamento para realizar uma nova revisão do padrão, a qual, se aprovada, ao terminar se tornará a referência de conformidade para o desenvolvimento das versões futuras do PostgreSQL.

O SQL92 definiu três conjuntos de funcionalidades para conformidade: Entrada, Intermediário e Completo. A maioria dos produtos que clamavam conformidade com este padrão SQL estavam em conformidade apenas com o nível de Entrada, porque o conjunto completo de funcionalidades nos níveis Intermediário e Completo era muito volumoso, ou em conflito com comportamentos legados.

O SQL99 define um conjunto maior de funcionalidades individuais em vez dos três níveis amplos não efetivos encontrados no SQL92. Um subconjunto grande destas funcionalidades representa as funcionalidades "núcleo", que toda implementação SQL em conformidade deve atender. As demais funcionalidades são inteiramente opcionais. Algumas funcionalidades opcionais são agrupadas em "pacotes", com os quais as implementações SQL podem clamar conformidade, portanto clamando conformidade com um determinado grupo de funcionalidades.

O padrão SQL99 está dividido em 5 partes: Framework, Foundation, Call Level Interface, Persistent Stored Modules e Host Language Bindings. O PostgreSQL somente cobre as partes 1, 2 e 5. A parte 3 é semelhante à interface ODBC, e a parte 4 é semelhante à linguagem de programação PL/pgSQL, mas a conformidade exata não é pretendida em nenhum destes dois casos.

A seguir é fornecida uma lista das funcionalidades suportadas pelo PostgreSQL, seguida por uma lista de funcionalidades definidas no SQL99 ainda não são suportadas pelo PostgreSQL. As duas listas são aproximadas: pode haver pequenos detalhes não em conformidade para uma funcionalidade listada como suportada, e grande parte de uma funcionalidade não suportada pode, na verdade, estar implementada. O corpo principal da documentação sempre contém informações mais precisas relativas ao que funciona e que não funciona.

Nota: Códigos de funcionalidade contendo hífen são subfuncionalidades. Portanto, se uma determinada subfuncionalidade não for suportada, a funcionalidade principal também não é suportada, mesmo que outras subfuncionalidades sejam suportadas.

ANEXOS

Tabela – Limitações físicas

Descrição	Limite
Tamanho máximo de um DB	Ilimitado
Tamanho máximo de uma tabela	64 TB
Tamanho máximo de um registro	Ilimitado para a versão 7.1 e posteriores
Tamanho máximo de um campo	1GB para a versão 7.1 e posteriores
Máximo de linhas numa tabela	Ilimitado
Máximo de colunas numa tabela	1600 (Depende do Tipo varia de 250 a 1600)
Máximo de índices numa tabela	Ilimitado
Máximo de campos em chave primária	32

Tabela - Funcionalidades suportadas

Identificador	Pacote	Descrição
B012	Core	Embedded C
B021		Direct SQL
E011	Core	Numeric data types
E011-01	Core	INTEGER and SMALLINT data types
E011-02	Core	REAL, DOUBLE PRECISION, and FLOAT data types
E011-03	Core	DECIMAL and NUMERIC data types
E011-04	Core	Arithmetic operators
E011-05	Core	Numeric comparison
E011-06	Core	Implicit casting among the numeric data types
E021	Core	Character data types
E021-01	Core	CHARACTER data type
E021-02	Core	CHARACTER VARYING data type
E021-03	Core	Character literals
E021-04	Core	CHARACTER_LENGTH function
E021-05	Core	OCTET_LENGTH function
E021-06	Core	SUBSTRING function
E021-07	Core	Character concatenation
E021-08	Core	UPPER and LOWER functions
E021-09	Core	TRIM function
E021-10	Core	Implicit casting among the character data types
E021-11	Core	POSITION function
E021-12	Core	Character comparison
E031	Core	Identifiers

Identificador	Pacote	Descrição
E031-01	Core	Delimited identifiers
E031-02	Core	Lower case identifiers
E031-03	Core	Trailing underscore
E051	Core	Basic query specification
E051-01	Core	SELECT DISTINCT
E051-02	Core	GROUP BY clause
E051-04	Core	GROUP BY can contain columns not in <select list>
E051-05	Core	Select list items can be renamed. AS is required
E051-06	Core	HAVING clause
E051-07	Core	Qualified * in select list
E051-08	Core	Correlation names in the FROM clause
E051-09	Core	Rename columns in the FROM clause
E061	Core	Basic predicates and search conditions
E061-01	Core	Comparison predicate
E061-02	Core	BETWEEN predicate
E061-03	Core	IN predicate with list of values
E061-04	Core	LIKE predicate
E061-05	Core	LIKE predicate ESCAPE clause
E061-06	Core	NULL predicate
E061-07	Core	Quantified comparison predicate
E061-08	Core	EXISTS predicate
E061-09	Core	Subqueries in comparison predicate
E061-11	Core	Subqueries in IN predicate
E061-12	Core	Subqueries in quantified comparison predicate
E061-13	Core	Correlated subqueries
E061-14	Core	Search condition
E071	Core	Basic query expressions
E071-01	Core	UNION DISTINCT table operator
E071-02	Core	UNION ALL table operator
E071-03	Core	EXCEPT DISTINCT table operator
E071-05	Core	Columns combined via table operators need not have exactly the same data type
E071-06	Core	Table operators in subqueries
E081-01	Core	SELECT privilege
E081-02	Core	DELETE privilege
E081-03	Core	INSERT privilege at the table level
E081-04	Core	UPDATE privilege at the table level
E081-06	Core	REFERENCES privilege at the table level
E081-08	Core	WITH GRANT OPTION
E091	Core	Set functions
E091-01	Core	AVG
E091-02	Core	COUNT
E091-03	Core	MAX
E091-04	Core	MIN
E091-05	Core	SUM

Identificador	Pacote	Descrição
E091-06	Core	ALL quantifier
E091-07	Core	DISTINCT quantifier
E101	Core	Basic data manipulation
E101-01	Core	INSERT statement
E101-03	Core	Searched UPDATE statement
E101-04	Core	Searched DELETE statement
E111	Core	Single row SELECT statement
E121-01	Core	DECLARE CURSOR
E121-02	Core	ORDER BY columns need not be in select list
E121-03	Core	Value expressions in ORDER BY clause
E121-04	Core	OPEN statement
E121-08	Core	CLOSE statement
E121-10	Core	FETCH statement implicit NEXT
E121-17	Core	WITH HOLD cursors
E131	Core	Null value support (nulls in lieu of values)
E141	Core	Basic integrity constraints
E141-01	Core	NOT NULL constraints
E141-02	Core	UNIQUE constraints of NOT NULL columns
E141-03	Core	PRIMARY KEY constraints
E141-04	Core	Basic FOREIGN KEY constraint with the NO ACTION default for both referential delete action and referential update action
E141-06	Core	CHECK constraints
E141-07	Core	Column defaults
E141-08	Core	NOT NULL inferred on PRIMARY KEY
E141-10	Core	Names in a foreign key can be specified in any order
E151	Core	Transaction support
E151-01	Core	COMMIT statement
E151-02	Core	ROLLBACK statement
E152	Core	Basic SET TRANSACTION statement
E152-01	Core	SET TRANSACTION statement: ISOLATION LEVEL SERIALIZABLE clause
E152-02	Core	SET TRANSACTION statement: READ ONLY and READ WRITE clauses
E161	Core	SQL comments using leading double minus
E171	Core	SQLSTATE support
F021	Core	Basic information schema
F021-01	Core	COLUMNS view
F021-02	Core	TABLES view
F021-03	Core	VIEWS view
F021-04	Core	TABLE_CONSTRAINTS view
F021-05	Core	REFERENTIAL_CONSTRAINTS view
F021-06	Core	CHECK_CONSTRAINTS view
F031	Core	Basic schema manipulation
F031-01	Core	CREATE TABLE statement to create persistent base tables
F031-02	Core	CREATE VIEW statement
F031-03	Core	GRANT statement

Identificador	Pacote	Descrição
F031-04	Core	ALTER TABLE statement: ADD COLUMN clause
F031-13	Core	DROP TABLE statement: RESTRICT clause
F031-16	Core	DROP VIEW statement: RESTRICT clause
F031-19	Core	REVOKE statement: RESTRICT clause
F032		CASCADE drop behavior
F033		ALTER TABLE statement: DROP COLUMN clause
F034		Extended REVOKE statement
F034-01		REVOKE statement performed by other than the owner of a schema object
F034-02		REVOKE statement: GRANT OPTION FOR clause
F034-03		REVOKE statement to revoke a privilege that the grantee has WITH GRANT OPTION
F041	Core	Basic joined table
F041-01	Core	Inner join (but not necessarily the INNER keyword)
F041-02	Core	INNER keyword
F041-03	Core	LEFT OUTER JOIN
F041-04	Core	RIGHT OUTER JOIN
F041-05	Core	Outer joins can be nested
F041-07	Core	The inner table in a left or right outer join can also be used in an inner join
F041-08	Core	All comparison operators are supported (rather than just =)
F051	Core	Basic date and time
F051-01	Core	DATE data type (including support of DATE literal)
F051-02	Core	TIME data type (including support of TIME literal) with fractional seconds precision of at least 0
F051-03	Core	TIMESTAMP data type (including support of TIMESTAMP literal) with fractional seconds precision of at least 0 and 6
F051-04	Core	Comparison predicate on DATE, TIME, and TIMESTAMP data types
F051-05	Core	Explicit CAST between datetime types and character types
F051-06	Core	CURRENT_DATE
F051-07	Core	LOCALTIME
F051-08	Core	LOCALTIMESTAMP
F052	Enhanced datetime facilities	Intervals and datetime arithmetic
F081	Core	UNION and EXCEPT in views
F111-02		READ COMMITTED isolation level
F131	Core	Grouped operations
F131-01	Core	WHERE, GROUP BY, and HAVING clauses supported in queries with grouped views
F131-02	Core	Multiple tables supported in queries with grouped views
F131-03	Core	Set functions supported in queries with grouped views
F131-04	Core	Subqueries with GROUP BY and HAVING clauses and grouped views
F131-05	Core	Single row SELECT with GROUP BY and HAVING clauses and grouped views
F171		Multiple schemas per user
F191	Enhanced integrity management	Referential delete actions
F201	Core	CAST function

Identificador	Pacote	Descrição
F221	Core	Explicit defaults
F222		INSERT statement: DEFAULT VALUES clause
F231		Privilege Tables
F231-01		TABLE_PRIVILEGES view
F231-02		COLUMN_PRIVILEGES view
F231-03		USAGE_PRIVILEGES view
F251		Domain support
F261	Core	CASE expression
F261-01	Core	Simple CASE
F261-02	Core	Searched CASE
F261-03	Core	NULLIF
F261-04	Core	COALESCE
F271		Compound character literals
F281		LIKE enhancements
F302	OLAP facilities	INTERSECT table operator
F302-01	OLAP facilities	INTERSECT DISTINCT table operator
F302-02	OLAP facilities	INTERSECT ALL table operator
F304	OLAP facilities	EXCEPT ALL table operator
F311-01	Core	CREATE SCHEMA
F311-02	Core	CREATE TABLE for persistent base tables
F311-03	Core	CREATE VIEW
F311-05	Core	GRANT statement
F321		User authorization
F361		Subprogram support
F381		Extended schema manipulation
F381-01		ALTER TABLE statement: ALTER COLUMN clause
F381-02		ALTER TABLE statement: ADD CONSTRAINT clause
F381-03		ALTER TABLE statement: DROP CONSTRAINT clause
F391		Long identifiers
F401	OLAP facilities	Extended joined table
F401-01	OLAP facilities	NATURAL JOIN
F401-02	OLAP facilities	FULL OUTER JOIN
F401-03	OLAP facilities	UNION JOIN
F401-04	OLAP facilities	CROSS JOIN
F411	Enhanced datetime facilities	Time zone specification
F421		National character
F431		Read-only scrollable cursors
F431-01		FETCH with explicit NEXT
F431-02		FETCH FIRST
F431-03		FETCH LAST
F431-04		FETCH PRIOR
F431-05		FETCH ABSOLUTE
F431-06		FETCH RELATIVE
F441		Extended set function support
F471	Core	Scalar subquery values

Identificador	Pacote	Descrição
F481	Core	Expanded NULL predicate
F491	Enhanced integrity management	Constraint management
F501	Core	Features and conformance views
F501-01	Core	SQL_FEATURES view
F501-02	Core	SQL_SIZING view
F501-03	Core	SQL_LANGUAGES view
F502		Enhanced documentation tables
F502-01		SQL_SIZING_PROFILES view
F502-02		SQL_IMPLEMENTATION_INFO view
F502-03		SQL_PACKAGES view
F511		BIT data type
F531		Temporary tables
F555	Enhanced datetime facilities	Enhanced seconds precision
F561		Full value expressions
F571		Truth value tests
F591	OLAP facilities	Derived tables
F611		Indicator data types
F651		Catalog name qualifiers
F701	Enhanced integrity management	Referential update actions
F711		ALTER domain
F761		Session management
F771		Connection management
F781		Self-referencing operations
F791		Insensitive cursors
F801		Full set function
S071	Enhanced object support	SQL paths in function and type name resolution
S111	Enhanced object support	ONLY in query expressions
S211	Enhanced object support, SQL/MM support	User-defined cast functions
T031		BOOLEAN data type
T141		SIMILAR predicate
T151		DISTINCT predicate
T171		LIKE clause in table definition
T191	Enhanced integrity management	Referential action RESTRICT
T201	Enhanced integrity management	Comparable data types for referential constraints
T211-01	Enhanced integrity management, Active database	Triggers activated on UPDATE, INSERT, or DELETE of one base table
T211-02	Enhanced integrity management, Active database	BEFORE triggers
T211-03	Enhanced integrity management, Active database	AFTER triggers
T211-04	Enhanced integrity management, Active database	FOR EACH ROW triggers

Identificador	Pacote	Descrição
T211-07	Enhanced integrity management, Active database	TRIGGER privilege
T212	Enhanced integrity management	Enhanced trigger capability
T231		SENSITIVE cursors
T241		START TRANSACTION statement
T312		OVERLAY function
T321-01	Core	User-defined functions with no overloading
T321-03	Core	Function invocation
T321-06	Core	ROUTINES view
T321-07	Core	PARAMETERS view
T322	PSM, SQL/MM support	Overloading of SQL-invoked functions and procedures
T323		Explicit security for external routines
T351		Bracketed SQL comments (/*...*/ comments)
T441		ABS and MOD functions
T501		Enhanced EXISTS predicate
T551		Optional key words for default syntax
T581		Regular expression substring function
T591		UNIQUE constraints of possibly null columns

BIBLIOGRAFIA

Referências e artigos selecionados para o SQL e para o PostgreSQL.

Alguns relatórios oficiais e técnicos da equipe original de desenvolvimento do POSTGRES estão disponíveis no sítio na Web do Departamento de Ciência da Computação da Universidade da Califórnia em Berkeley.

Livros de Referência sobre o SQL

Judith Bowman, Sandra Emerson, e Marcy Darnovsky, The Practical SQL Handbook: Using Structured Query Language, Third Edition, Addison-Wesley, ISBN 0-201-44787-8, 1996.

C. J. Date e Hugh Darwen, A Guide to the SQL Standard: A user's guide to the standard database language SQL, Fourth Edition, Addison-Wesley, ISBN 0-201-96426-0, 1997.

C. J. Date, An Introduction to Database Systems, Volume 1, Sixth Edition, Addison-Wesley, 1994.

Ramez Elmasri e Shamkant Navathe, Fundamentals of Database Systems, 3rd Edition, Addison-Wesley, ISBN 0-805-31755-4, August 1999.

Jim Melton e Alan R. Simon, Understanding the New SQL: A complete guide, Morgan Kaufmann, ISBN 1-55860-245-3, 1993.

Jeffrey D. Ullman, Principles of Database and Knowledge: Base Systems, Volume 1, Computer Science Press, 1988.

Documentação específica do PostgreSQL

Stefan Simkovics, Enhancement of the ANSI SQL Implementation of PostgreSQL, Department of Information Systems, Vienna University of Technology, November 29, 1998.

Discusses SQL history and syntax, and describes the addition of INTERSECT and EXCEPT constructs into PostgreSQL. Prepared as a Master's Thesis with the support of O. Univ. Prof. Dr. Georg Gottlob and Univ. Ass. Mag. Katrin Seyr at Vienna University of Technology.

A. Yu e J. Chen, The POSTGRES Group, The Postgres95 User Manual, University of California, Sept. 5, 1995.

Zelaine Fong, The design and implementation of the POSTGRES query optimizer, University of California, Berkeley, Computer Science Department.

Conferências e Artigos

Nels Olson, Partial indexing in POSTGRES: research project, University of California, UCB Engin T7.49.1993 O676, 1993.

L. Ong e J. Goh, "A Unified Framework for Version Modeling Using Production Rules in a Database System", ERL Technical Memorandum M90/33, University of California, April, 1990.

L. Rowe e M. Stonebraker, "The POSTGRES data model", Proc. VLDB Conference, Sept. 1987.

P. Seshadri e A. Swami, "Generalized Partial Indexes ", Proc. Eleventh International Conference on Data Engineering, 6-10 March 1995, IEEE Computer Society Press, Cat. No.95CH35724, 1995, p. 420-7.

M. Stonebraker e L. Rowe, "The design of POSTGRES", Proc. ACM-SIGMOD Conference on Management of Data, May 1986.

M. Stonebraker, E. Hanson, e C. H. Hong, "The design of the POSTGRES rules system", Proc. IEEE Conference on Data Engineering, Feb. 1987.

M. Stonebraker, "The design of the POSTGRES storage system", Proc. VLDB Conference, Sept. 1987.

M. Stonebraker, M. Hearst, e S. Potamianos, "A commentary on the POSTGRES rules system", SIGMOD Record 18(3), Sept. 1989.

M. Stonebraker, "The case for partial indexes", SIGMOD Record 18(4), Dec. 1989, p. 4-11.

M. Stonebraker, L. A. Rowe, e M. Hirohama, "The implementation of POSTGRES", Transactions on Knowledge and Data Engineering 2(1), IEEE, March 1990.

M. Stonebraker, A. Jhingran, J. Goh, e S. Potamianos, "On Rules, Procedures, Caching and Views in Database Systems", Proc. ACM-SIGMOD Conference on Management of Data, June 1990.

Links

www.postgresql.org

www.postgresql.org.br

http://geocities.yahoo.com.br/halleypo/pg74/

EXEMPLOS

1. Seleciona registros não duplicados de informações patrimoniais de um equipamento com ligações ao cadastro de materiais e de responsáveis.

```
SELECT DISTINCT
        bens.registro, bens.responsavel, responsaveis.descricao, bens.material, material.descricao,
bens.descricao, bens.valor
        FROM bens
                INNER JOIN  materiais ON
                        ( bens.material =  materiais.material)
                INNER JOIN  responsaveis ON
                        ( bens.responsavel =  responsaveis.responsavel)
        WHERE bens.registro= '$w_equipamento';
```

ou executado por *default*

```
SELECT DISTINCT
        bens.registro, bens.responsavel, responsaveis.descricao, bens.material, material.descricao,
bens.descricao, bens.valor
        FROM bens, materiais, responsaveis
        WHERE       bens.registro= '$w_equipamento' AND   -- $w_equipamento = material
                    bens.material =  materiais.material AND
                    bens.responsavel =  responsaveis.responsavel;
```

EXERCÍCIOS

1. Criar o DB e a tabela:

Db=escolas (dono=postgres, sql_ascii); table=alunos;

Colunas	aluno	curso	nome	endereco	bairro	cep	cidade	estado	ddd	telefone	celular	email	data_inclusao	data_alteracao
Tipo	String	String	String	String	String	String	String	String	Integer	String	String	String	Date	Date
Tamanho	10	4	50	50	20	8	50	2	5	10	10	20	-	-
Restrições	NN	NN	NN	NN	-	-	-	NN	-	NN	-	-	NN	-
Obs	UC	UC	UC	-	-	-	-	UC	-	-	-	-	-	-

Observações: Restrições=NN (Not Nul); Obs=UC (Letras maiúsculas); data_inclusao e data_alteracao deverão ser preenchidas com a data do sistema.

2. Realize as seguintes operações:
a) Inclua 20 registros aleatoriamente na tabela, ddd em branco;
b) Inclua mais dois registro com código de aluno repetido;
c) Selecione todos os registros cujo estado não seja 'CE';
d) Selecione todos os registros do estado CE e da cidade de Fortaleza;
e) Selecione o aluno, curso e nome de determinado curso em ordem alfabética;
f) Efetue um backup dos dados do sistema no arquivo backup.dump;
g) Efetue um backup dos dados e esquemas da tabela no arquivo backup.pgdumpall;

3. Altere a tabela anterior fazendo com que o campo aluno seja chave primária;
4. Repita o passo 2.
5. Realize as seguintes operações:
a) Altere a coluna de ddd de acordo com o estado (CE=85, PE=81, BA=71, DF=61, PI=86, etc);
b) Selecione todos as linhas de alunos que comecem com a letra A;
c) Delete todas as linhas cujos estados não sejam da região nordeste;

6. Faça um backup geral de seus dados e envie para o seu e-mail;
7. Crie a tabela de cursos seguindo o modelo abaixo:

```
cursos varchar(4) not null, chave primária;
descricao varchar(30) not null,
carga_horaria integer,
vagas_total     integer(2),
vagas_usadas  integer(2),
valor_unitario float4,
valor_subtotal float4,
data_inclusao timestamp,
data_alteracao timestamp.
```

8. Inclua linhas na tabela de cursos segundo sua imaginação;
9. Realize operações diversas (livres).
10. Efetue um backup geral.

2. Dado o arquivo anexo agenda.sql (Dados dos médicos da UNIMED de Fortaleza).

1. Crie um banco teste cujo dono é postgres e ENCODING=SQL_ASCII;

2. Conecte-se ao banco e importe o arquivo agenda.sql

3. Verifique o conteúdo da tabela;

4. Transforme todo o conteúdo do assunto em letras minúsculas;

5. Transforme todo o conteúdo do subassunto em letras minúsculas;

6. Crie uma VIEW cuja execução selecionará da tabela as colunas: registro, assunto, subassunto, nome em ordem de nome;

7. Crie uma tabela <estados> do Brasil (Colunas estado, descricao, capital) e insira os respectivos estados;

8. Crie uma VIEW agenda_estados onde serão exibidos as colunas registro, nome, assunto, subassunto, ddd, telefone, estado, estados.descricao, cidade;

9. Verifique se na tabela <agenda> existem registros com nomes nulos ou em branco e suprima-os;

10. Altere a tabela <agenda> para restringir nomes, assunto, subassunto não nulos e diferentes de '';

11. Verifique se não há números de registro repetidos. Transforme a coluna registro em chave primária;

12. Selectione os 50 primeiros registros em ordem alfabética cujo nome tenha 'silva' no conteúdo;

13. Selecione todos os registros de cardiologistas;

14. Verifique quantos registros existem em cada subassunto;

15. Tire uma cópia da estrutura das tabelas no arquivo estrutura.sql

16. Tire uma cópia dos dados no arquivo dados.sql

17. Tire uma cópia total de estruturas e dados no arquivo geral.sql

18. Crie um novo banco chamado agenda e coloque todos os dados do banco teste;

19. Na tabela agenda o campo estado deve corresponder a um registro válido da tabela estados (integridade referencial);

20. Realiza um backup geral dos dois bancos.

PROJETOS

1. Agência de consultoria

A empresa Consultores Associados Ltda. funciona como uma espécie de cooperativa de especialistas em todas as áreas do conhecimento. Cada consultor, na verdade, é um associado da empresa, recebendo proventos relativos ao número de horas trabalhadas e ao valor de sua hora de especialidade.

Tendo reconhecida atuação na região de sua sede, a CAL resolveu aumentar sua área de atuação, desejando atuar a nível nacional. Para tanto, precisa melhorar o controle sobre informações dos consultores associados e dos clientes fixos ou potenciais, tanto para fins financeiros quanto para planejar o atendimento mais rápido e eficiente dos clientes. Assim, foram levantados os seguintes requisitos para um sistema de informações:

- Devem ser mantidos todos os dados relevantes dos consultores associados, como uma espécie de currículo. Assim, para cada consultor deve-se saber o nome, endereço completo, telefones de contato (fixo, celular, fax e comercial), Identidade, CPF e registro no INSS (para efetuar os pagamentos e desconto de impostos), número da conta bancária, agência e banco utilizados para o pagamento, e-mail e icq. Além dos dados cadastrais básicos, é necessário armazenar dados sobre a formação do consultor, como cursos de formação realizados (técnico, graduação, especialização, mestrado, doutorado, pós-doutorado), cursos de aperfeiçoamento e qualificação (qual curso, assunto, duração do curso), e experiência profissional (empresa, datas de entrada e saída, funções exercidas na empresa). Para facilitar pesquisas sobre áreas de atuação dos consultores, existe ainda uma relação de termos padronizados para cada consultor, relativo às especialidades em que presta serviços. Por fim, são relacionadas também atividades que o consultor pode oferecer (consultoria, instrução, planejamento, projeto, inspeção, auditoria).
- Para cada cliente, são armazenados seus dados característicos, como nome, endereço completo e fones de contato, CPF e RG (pessoa física), CNPJ e Inscrição estadual (pessoa jurídica).
- A cada abertura de um pedido de consultoria por parte de um cliente, o sistema procura localizar um consultor que já tenha atuado na empresa (e que o cliente tenha um bom conceito em relação a ele), na área de consultoria solicitada, e que esteja o mais próximo do cliente possível (se não na mesma cidade, na cidade mais próxima), nessa ordem. Selecionado um consultor, é feita uma ordem de serviço, onde são registrados todos os encontros do consultor com o cliente, o número de horas empregadas e a atividade realizada. Ao final do serviço de consultoria, o cliente dá seu aval e preenche um questionário de avaliação, onde é dada uma nota de 5 a 10 para o consultor. Ao chegar na CAL, é emitida uma nota fiscal, que é cobrada do cliente.
- Os recursos recebidos pela CAL são repassados aos consultores, sendo retidos 10% para despesas operacionais, pagamento de funcionários administrativos e fundo para viagens e emergências, além de pagos os impostos relativos à prestação de serviços e taxas sindicais. O valor líquido a ser percebido pelo consultor fica registrado como um saldo de conta bancária, podendo este ser retirado a qualquer instante pelo consultor em dinheiro ou

transferido para uma conta bancária a escolha deste. Todas as transações são registradas para balanço futuro.

2. Site de culinária

Teus serviços foram contratados para desenvolver um website sobre culinária. Para tanto, querem que o usuário, após rápido cadastro gratuito, tenha acesso a um banco de receitas *oficiais* (de mestre-cucas profissionais ou reconhecidos) e *amadoras* (enviadas pelos associados do site). As receitas podem ser pesquisadas em função dos ingredientes utilizados, grau de dificuldade, categoria (sopas, massas, carnes, sobremesas,...).

Toda receita tem um nome, autor, relação dos ingredientes com suas quantidades e a relação dos passos a serem seguidos. Opcionalmente, pode-se ter um link com algum passo ou ingrediente, que leva ao aparecimento de uma dica ou curiosidade culinária. Além disso, cada receita pode ter uma referência a outras receitas complementares (se carne, por exemplo, o que pode acompanhar; ou que sobremesa combinaria com o prato principal).

É interessante manter uma biografia resumida sobre os cozinheiros. Outra funcionalidade do site é manter uma lista com as 10 receitas mais acessadas e as 10 mais recentes.

3. Videolocadora

A videolocadora **Locadora de Histórias** está necessitando organizar seus dados, a fim de viabilizar maior eficiência e novos serviços aos seus clientes. Para tanto, contratou nossos serviços para elaboração de um sistema que não apenas resolva suas necessidades atuais, mas possibilite expansão de seus serviços em pouco tempo.

Esta videolocadora trabalha com fitas de vídeo e dvds, tendo títulos em ambos os formatos de mídia. Além disso, oferece aos seus clientes a possibilidade de locar CD's de trilhas sonoras de filmes. Cada título de filme pode ter várias cópias em fita de vídeo e DVD, além de cópias em CD da trilha sonora. Todas as cópias são identificadas unicamente.

Para facilitar pesquisas por parte dos clientes, cada título tem relacionado dados como Nome do filme, duração, gênero, diretor, companhia cinematográfica, ano de lançamento, atores principais, prêmios alcançados (se for o caso) e uma sinopse do mesmo, normalmente copiada da própria capa do título. Como a locadora tem como missão ser uma especializada em cinéfilos, também guarda dados sobre diretores e atores, além de relacionar filmes similares para facilitar a seleção.

Dependendo do filme, pode haver diferença de valores das locações. Existem as locações promoção, catálogo, lançamento e estréia, cada uma com um valor e uma duração diferenciada em dias. A cada locação, são registrados as cópias de títulos lavadas pelo cliente, e na devolução o valor a ser pago é calculado em função do tipo de locação vezes o número de períodos de cada título levado. Dependendo do tipo de promoção, pode haver um desconto, como no caso de aniversário do cliente, onde uma locação estréia é gratuita (ou duas outras), e a cada 50 locações, onde o cliente não paga a 50ª.

Em relação aos clientes, são armazenados dados cadastrais comuns, além da data de inscrição na videolocadora e dados de dependentes (número indefinido, desde que com relação familiar comprovada). Os clientes, além de locações e devoluções, podem fazer reservas, que devem ser retiradas até 4 horas após o filme ter chegado na locadora. Nesses casos, os clientes são informados

pelo telefone de contato. Após 4 horas sem ter sido retirado, o filme volta à prateleira e a reserva é desativada.

Para a gerência, interessa saber os principais títulos locados, e em que mídia. Também interessa saber quantas reservas são realizadas e quantas destas resultam em locações. É interessante também um relatório mensal de cópias desativadas, e por que motivo. Também interessa saber o volume de locações feitas por cada funcionário, pois é dado um bônus mensal aos três funcionários que mais atendem clientes. Deve-se ter em mente que não há planos de expansão da locadora, ou seja, ela deve permanecer em uma única sede.

4. Supermercado

O Supermercado **É Barato** resolveu implantar um sistema de vendas on-line para os clientes ocupados que utilizam a Internet em seu escritório, disponibilizando mais uma comodidade a seus assíduos fregueses.

O sistema baseia-se em uma lista de compras virtual, onde o cliente pode incluir diversos itens disponíveis no estoque do supermercado, junto com sua quantidade. Ao terminar sua lista de compras virtual, o valor total da compra é calculado e pergunta-se o endereço de entrega.

O cliente pode optar por buscar no supermercado as compras ou entregar em um endereço, por default associado ao endereço residencial do cliente. Neste último caso, agrega-se uma taxa de entrega ao valor final, que depende do bairro do endereço de entrega. Devem ser observados os seguintes detalhes:

- Cada produto tem uma descrição, fabricante, preço de venda e quantidade em estoque. Os produtos são organizados em categorias, e cada categoria é subdividida em produtos gerais (ex.: Enlatados como categoria tem Molhos e extratos de tomate como produto geral, onde o Extrato de Tomate Elefante se encaixa).
- Para cada cliente, são armazenados o nome e o endereço completos, além do telefone residencial, telefone comercial, telefone celular e e-mail (assume-se que os clientes serão somente pessoas físicas). Para fins de acompanhamento futuro, será necessária guardar a data de ativação do cadastro, e o ranking do cliente (classificação do mesmo como especial, preferencial, comum, regular, irregular e devedor).
- Cada lista de compras deve conter, além dos itens do pedido, a data e a hora em que foi iniciada e concluída, qual era o cliente, a forma de pagamento e a forma de entrega, além do valor da entrega, se for o caso. Também deve armazenar a data e a hora da entrega (ou se o cliente foi buscar, a da retirada), quem realizou a entrega, o número da nota fiscal correspondente e um espaço para observações.
- Os itens de compra devem conter o código do produto e sua quantidade, observando que pode-se considerar produtos a granel (onde são medidas o kg, o litro) e por unidade (5 latas, 2 garrafas, 5 bananas).

5. Escolas

A Secretaria Municipal de Educação da cidade de Cafundó do Judas, cidade com grande área rural e pequena área urbana, enfrenta um problema para o planejamento de vagas a oferecer para o ensino fundamental e ensino médio para a população, especialmente a população rural. Para tanto, contratou um consultor que levantou os seguintes requisitos para uma solução para esse problema:

- As escolas estão espalhadas por todo o município, e é importante localizar geograficamente

sua posição. Cada escola pode oferecer ensino fundamental até 4a série, ensino fundamental até 8a série e ensino médio;

- O sistema deve compreender o cadastro das famílias da cidade, considerando para cada família sua localização e cadastro dos membros em idade escolar (menores de 18 anos), e em que série deveriam cursar no momento, bem como dados dos responsáveis pela família;
- Deve-se ter, também, a relação de professores, bem como das disciplinas que os mesmos estão aptos para exercer (até 4a série, por área, demais séries, por disciplina); é importante que se tenha a localização da residência do professor.

O sistema deve poder relacionar as escolas e os alunos do sistema municipal de ensino, visando localizar as escolas mais próximas das residências dos alunos, e de forma que os alunos que são irmãos fiquem na mesma escola; O sistema deve possibilitar, na medida do possível, relacionar os professores das disciplinas de cada série que se situam próximos à cada escola.

Índice Alfabético

Totalmente editado com softwares livres.