Administração de Banco de Dados

Fabio Caiut

A RNP – Rede Nacional de Ensino e Pesquisa – é qualificada como uma Organização Social (OS), sendo ligada ao Ministério da Ciência, Tecnologia e Inovação (MCTI) e responsável pelo Programa Interministerial RNP, que conta com a participação dos ministérios da Educação (MEC), da Saúde (MS) e da Cultura (MinC). Pioneira no acesso à Internet no Brasil, a RNP planeja e mantém a rede Ipê, a rede óptica nacional acadêmica de alto desempenho. Com Pontos de Presença nas 27 unidades da federação, a rede tem mais de 800 instituições conectadas. São aproximadamente 3,5 milhões de usuários usufruindo de uma infraestrutura de redes avançadas para comunicação, computação e experimentação, que contribui para a integração entre o sistema de Ciência e Tecnologia, Educação Superior, Saúde e Cultura.

RNP

Ministério da **Cultura**

Ministério da **Saúde**

Ministério da **Educação**

Ministério da **Ciência, Tecnologia e Inovação**

Administração de Banco de Dados

Fabio Caiut

Administração de Banco de Dados

Fabio Caiut

Rio de Janeiro
Escola Superior de Redes
2015

Rua Lauro Müller, 116 sala 1103
22290-906 Rio de Janeiro, RJ

Diretor Geral
Nelson Simões

Diretor de Serviços e Soluções
José Luiz Ribeiro Filho

Escola Superior de Redes

Coordenação
Luiz Coelho

Edição
Lincoln da Mata

Coordenador Acadêmico da Área de Desenvolvimento de Sistemas
John Lemos Forman

Equipe ESR (em ordem alfabética)
Adriana Pierro, Celia Maciel, Cristiane Oliveira, Derlinéa Miranda, Edson Kowask, Elimária Barbosa, Evellyn Feitosa, Felipe Nascimento, Lourdes Soncin, Luciana Batista, Luiz Carlos Lobato, Renato Duarte e Yve Abel Marcial.

Capa, projeto visual e diagramação
Tecnodesign

Versão
1.0.0

Distribuição
Escola Superior de Redes
Rua Lauro Müller, 116 – sala 1103
22290-906 Rio de Janeiro, RJ
http://esr.rnp.br
info@esr.rnp.br

Dados Internacionais de Catalogação na Publicação (CIP)

C138a Caiut, Fábio
Administração de banco de dados / Fábio Caiut. – Rio de Janeiro: RNP/ESR, 2015.
182 p. : il. ; 28 cm.

Bibliografia: p.165.
ISBN 978-85-63630-51-3

1. Banco de dados – administração, gerenciamento. 2. Sistema de gerenciamento de banco de dados (SGBD). 3. PostgreSQL. I. Titulo.

CDD 005.7406

Sumário

Escola Superior de Redes

A Escola Superior de Redes (ESR) é a unidade da Rede Nacional de Ensino e Pesquisa (RNP) responsável pela disseminação do conhecimento em Tecnologias da Informação e Comunicação (TIC). A ESR nasce com a proposta de ser a formadora e disseminadora de competências em TIC para o corpo técnico-administrativo das universidades federais, escolas técnicas e unidades federais de pesquisa. Sua missão fundamental é realizar a capacitação técnica do corpo funcional das organizações usuárias da RNP, para o exercício de competências aplicáveis ao uso eficaz e eficiente das TIC.

A ESR oferece dezenas de cursos distribuídos nas áreas temáticas: Administração e Projeto de Redes, Administração de Sistemas, Segurança, Mídias de Suporte à Colaboração Digital e Governança de TI.

A ESR também participa de diversos projetos de interesse público, como a elaboração e execução de planos de capacitação para formação de multiplicadores para projetos educacionais como: formação no uso da conferência web para a Universidade Aberta do Brasil (UAB), formação do suporte técnico de laboratórios do Proinfo e criação de um conjunto de cartilhas sobre redes sem fio para o programa Um Computador por Aluno (UCA).

A metodologia da ESR

A filosofia pedagógica e a metodologia que orientam os cursos da ESR são baseadas na aprendizagem como construção do conhecimento por meio da resolução de problemas típicos da realidade do profissional em formação. Os resultados obtidos nos cursos de natureza teórico-prática são otimizados, pois o instrutor, auxiliado pelo material didático, atua não apenas como expositor de conceitos e informações, mas principalmente como orientador do aluno na execução de atividades contextualizadas nas situações do cotidiano profissional.

A aprendizagem é entendida como a resposta do aluno ao desafio de situações-problema semelhantes às encontradas na prática profissional, que são superadas por meio de análise, síntese, julgamento, pensamento crítico e construção de hipóteses para a resolução do problema, em abordagem orientada ao desenvolvimento de competências.

Dessa forma, o instrutor tem participação ativa e dialógica como orientador do aluno para as atividades em laboratório. Até mesmo a apresentação da teoria no início da sessão de aprendizagem não é considerada uma simples exposição de conceitos e informações. O instrutor busca incentivar a participação dos alunos continuamente.

As sessões de aprendizagem onde se dão a apresentação dos conteúdos e a realização das atividades práticas têm formato presencial e essencialmente prático, utilizando técnicas de estudo dirigido individual, trabalho em equipe e práticas orientadas para o contexto de atuação do futuro especialista que se pretende formar.

As sessões de aprendizagem desenvolvem-se em três etapas, com predominância de tempo para as atividades práticas, conforme descrição a seguir:

Primeira etapa: apresentação da teoria e esclarecimento de dúvidas (de 60 a 90 minutos).
O instrutor apresenta, de maneira sintética, os conceitos teóricos correspondentes ao tema da sessão de aprendizagem, com auxílio de slides em formato PowerPoint. O instrutor levanta questões sobre o conteúdo dos slides em vez de apenas apresentá-los, convidando a turma à reflexão e participação. Isso evita que as apresentações sejam monótonas e que o aluno se coloque em posição de passividade, o que reduziria a aprendizagem.

Segunda etapa: atividades práticas de aprendizagem (de 120 a 150 minutos).
Esta etapa é a essência dos cursos da ESR. A maioria das atividades dos cursos é assíncrona e realizada em duplas de alunos, que acompanham o ritmo do roteiro de atividades proposto no livro de apoio. Instrutor e monitor circulam entre as duplas para solucionar dúvidas e oferecer explicações complementares.

Terceira etapa: discussão das atividades realizadas (30 minutos).
O instrutor comenta cada atividade, apresentando uma das soluções possíveis para resolvê-la, devendo ater-se àquelas que geram maior dificuldade e polêmica. Os alunos são convidados a comentar as soluções encontradas e o instrutor retoma tópicos que tenham gerado dúvidas, estimulando a participação dos alunos. O instrutor sempre estimula os alunos a encontrarem soluções alternativas às sugeridas por ele e pelos colegas e, caso existam, a comentá-las.

Sobre o curso

Este curso abrange os conceitos teóricos e práticos de Administração de Banco de Dados para o PostgreSQL, SGBD open source altamente avançado e robusto. Descreve a arquitetura do PostgreSQL, apresentando as funções de cada componente, define e demonstra atividades comuns de administração como: instalação, configuração, backup, recuperação e outras rotinas de manutenção essenciais a saúde das bases de dados.

Aprofunda no gerenciamento do PostgreSQL abordando e trabalhando conceitos avançados como tuning de queries, análise de planos de execução, catálogo interno e recursos de replicação. Discute os mais comuns problemas de desempenho enfrentados pelos DBAs, o monitoramento do banco e sua forte integração com o sistema operacional. Ainda, ilustra as boas práticas para se alcançar escalabilidade e alta disponibilidade.

A quem se destina

Pessoas com conhecimento básico em bancos de dados e infraestrutura e que precisarão trabalhar na administração/manutenção de ambientes PostgreSQL, se envolvendo com a rotina diária de monitoramento de seus processos e ajuste fino de suas configurações e procedimentos, buscando a melhor performance sem comprometer a segurança e consistência das informações armazenadas. Ainda que profissionais envolvidos com o desenvolvimento de sistemas acessando informações armazenadas em banco de dados possam se beneficiar deste curso, o mesmo não tem como foco a apresentação de conceitos de programação ou da linguagem SQ.

Convenções utilizadas neste livro

As seguintes convenções tipográficas são usadas neste livro:

Itálico
Indica nomes de arquivos e referências bibliográficas relacionadas ao longo do texto.

```
Largura constante
```

Indica comandos e suas opções, variáveis e atributos, conteúdo de arquivos e resultado da saída de comandos. Comandos que serão digitados pelo usuário são grifados em negrito e possuem o prefixo do ambiente em uso (no Linux é normalmente # ou $, enquanto no Windows é C:\).

Conteúdo de slide
Indica o conteúdo dos slides referentes ao curso apresentados em sala de aula.

Símbolo
Indica referência complementar disponível em site ou página na internet.

Símbolo
Indica um documento como referência complementar.

Símbolo
Indica um vídeo como referência complementar.

Símbolo
Indica um arquivo de áudio como referência complementar.

Símbolo
Indica um aviso ou precaução a ser considerada.

Símbolo
Indica questionamentos que estimulam a reflexão ou apresenta conteúdo de apoio ao entendimento do tema em questão.

Símbolo
Indica notas e informações complementares como dicas, sugestões de leitura adicional ou mesmo uma observação.

Símbolo
Indica atividade a ser executada no Ambiente Virtual de Aprendizagem – AVA.

Permissões de uso

Comentários e perguntas

Para enviar comentários e perguntas sobre esta publicação:
Escola Superior de Redes RNP
Endereço: Av. Lauro Müller 116 sala 1103 – Botafogo
Rio de Janeiro – RJ – 22290-906
E-mail: info@esr.rnp.br

Sobre o autor

Fábio Caiut É graduado em Ciência da Computação na Universidade Federal do Paraná (2002) e Especialista em Banco de Dados pela Pontifícia Universidade Católica do Paraná (2010). Trabalha com TI desde 2000, atuando principalmente em Infraestrutura e Suporte ao Desenvolvimento. É Administrador de Banco de Dados PostgreSQL há 5 anos no Tribunal de Justiça do Paraná, focado em desempenho e alta disponibilidade com bancos de dados de alta concorrência. Tem experiência como desenvolvedor certificado Lotus Notes e Microsoft .NET nas softwarehouses Productique e Sofhar, DBA SQL Server e suporte à infraestrutura de desenvolvimento na Companhia de Saneamento do Paraná.

John Lemos Forman é Mestre em Informática (ênfase em Engenharia de Software) e Engenheiro de Computação pela PUC-Rio, com pós-graduação em Gestão de Empresas pela COPPEAD/UFRJ. É vice-presidente do Sindicato das Empresas de Informática do Rio de Janeiro – TIRIO, membro do Conselho Consultivo e de normas Éticas da Assespro-RJ e Diretor da Riosoft. É sócio e Diretor da J.Forman Consultoria e coordenador acadêmico da área de desenvolvimento de sistemas da Escola Superior de Redes da RNP. Acumula mais de 29 anos de experiência na gestão de empresas e projetos inovadores de base tecnológica, com destaque para o uso das TIC na Educação, mídias digitais e Saúde.

1

Arquitetura e instalação do Banco de Dados

objetivos

Conhecer o PostgreSQL e a descrição de sua arquitetura em alto nível, através dos seus processos componentes e suas estruturas de memória; Entender o funcionamento de Banco de Dados e sua importância; Aprender sobre a arquitetura geral dos SGBDs.

conceitos

Banco de Dados; SGBD; Log de Transação; Write-Ahead Log; Checkpoint; Transação; ACID; Shared Memory; Shared Buffer; Arquitetura Multiprocessos; Backend e Page Cache.

Conceitos: Banco de Dados e SGBD

- Banco de Dados: é um conjunto de dados relacionados, representando um pedaço ou interpretação do mundo real, que possui uma estrutura lógica com significado inerente e comumente possui uma finalidade e usuários específicos.
- Sistema Gerenciador de Bancos de Dados (SGBD): um pacote de software, um sistema de finalidade genérica para gerenciar os Bancos de Dados. Facilita o processo de definição, construção e manipulação de bancos de dados.

Características de um SGBD

- **Independência de dados**: não depende da aplicação para entender os dados;
- **Acesso eficiente**: através de índices, caches de dados e consultas, visões materializadas etc.;
- **Segurança mais especializada**: através de visões, ou mesmo por colunas, ou por máquina de origem etc.;
- **Acesso concorrente e compartilhamento dos dados**: permite inúmeros usuários simultâneos operarem sobre os dados;
- **Restrições de Integridade**: impedir um identificador duplicado ou um valor fora da lista etc.;
- **Recuperação de Falhas**: retorna para um estado íntegro depois de um crash;
- **Manipular grande quantidade de dados**: bilhões ou trilhões de registros e petabytes de dados;
- **Diminui o tempo de desenvolvimento de aplicações**: que não precisam escrever código para acessar e estruturar os dados.

Exercício de fixação
Arquitetura genérica de um Banco de Dados

Você é responsável pelo desenvolvimento de um sistema de vendas. Como resolveria a seguinte situação?

Uma grande venda é feita, com mais de 200 itens. O registro representando o pedido é gravado, mas após gravar 50 itens ocorre uma queda de energia. O que deve ocorrer depois que o sistema voltar ao ar?

Apesar de alguns autores diferenciarem o conceito de Base de Dados e Banco de Dados, comumente eles são usados como sinônimos. Às vezes uma instância inteira, um servidor de Bancos de Dados, é chamado simplesmente de Banco de Dados. Por isso aproveitaremos o termo Base de Dados para sempre identificar cada uma das bases ou bancos dentro de uma instância, mas jamais a instância.

Por definição, deseja-se que uma transação em um Banco de Dados seja Atômica: ou é feito tudo ou não é feito nada. Para garantir a propriedade chamada Atomicidade, o Banco de Dados utiliza um recurso chamado genericamente de Log de Transações.

Esse Log pode ser visto como um histórico, ou diário, de todas as operações que alteram os dados de uma base.

Quando uma transação efetua alterações – updates, inserts ou deletes – essas alterações são feitas nos arquivos de log de transação, e posteriormente apenas as transações que terminaram corretamente, ou seja, que efetuaram um COMMIT, são efetivadas nos arquivos de dados.

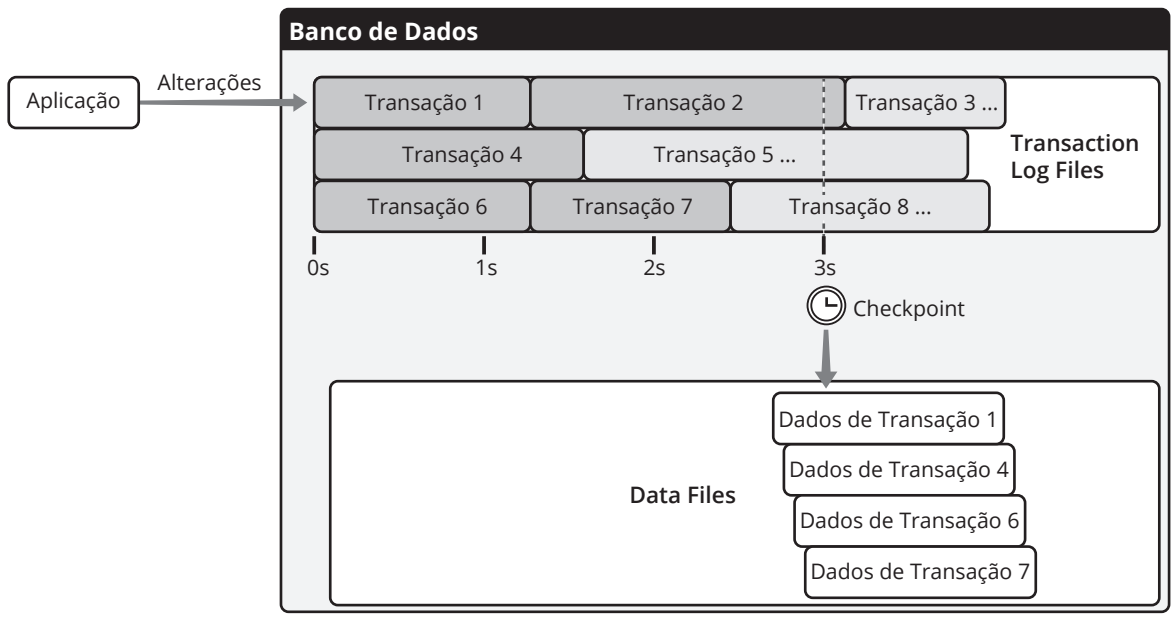


Figura 1.1
Transações são gravadas no log e apenas as comitadas são efetivadas nos arquivos de dados posteriormente.

Essa arquitetura dos SGBDs, baseada nos Logs de Transações, garante transações:

- Atômicas;
- Duráveis;
- Consistentes.

ACID é a abreviatura para Atomicidade, Consistência, Isolamento e Durabilidade, que são as propriedades que garantem que transações em um Banco de Dados são processadas de forma correta; portanto, garantem confiabilidade.

Introdução ao PostgreSQL

PostgreSQL é um Sistema Gerenciador de Banco de Dados Objeto-Relacional (SGBD-OR) poderoso e open source, em desenvolvimento há quase 20 anos, que conquistou forte reputação pela sua robustez, confiabilidade e integridade de dados. Totalmente compatível com as propriedades ACID e suporte completo a recursos como consultas complexas, chaves estrangeiras, joins, visões e triggers and store procedures em diversas linguagens.

- Implementa a maioria dos tipos de dados e definições do padrão SQL:2008, além de outros recursos próprios;
- Está disponível na maioria dos Sistemas Operacionais, incluindo Linux, AIX, BSD, HP-UX, Mac OS X, Solaris e Windows;
- É altamente extensível através de criação de tipos de dados e operações e funções sobre estes, suporte a dezenas de linguagens e Extension Libraries;
- Possui funcionalidades como Multi-Version Concurrency Control (MVCC), Point in Time Recovery (PITR), tablespaces, Replicação Síncrona e Assíncrona, Transações Aninhadas (Savepoint), Backup Online, um sofisticado Otimizador de Queries e Log de Transações (WAL) para tolerância a falhas.

O PostgreSQL foi baseado no Postgres 4.2, da Universidade da Califórnia/Berkeley, projeto encerrado em 1994. Ele é dito Objeto-Relacional por ter suporte a funcionalidades compatíveis com o conceito de orientação de objetos, como herança de tabelas, sobrecarga de operadores e funções e uso de métodos e construtores através de funções.

Seu site é http://www.postgresql.org, os desenvolvedores são o PostgreSQL Global Development Group (voluntários e empresas de diversas partes do mundo) e a licença é a PostgreSQL License, uma licença open source semelhante à BSD.

Versão do PostgreSQL

Todo o conteúdo aqui apresentado é baseado nas versões do PostgreSQL da família 9, particularmente a partir da versão 9.1, e versões posteriores. Para ilustrar detalhes da sua instalação ou funcionamento, usamos a versão mais recente disponível, que é a versão 9.3.4, mas possivelmente o lançamento de novas versões depois da confecção deste material poderão determinar diferenças que não estarão aqui refletidas.

De qualquer modo, a política geral de versões do PostgreSQL é que em minor releases, por exemplo, de 9.3.3 para 9.3.4 são aplicadas apenas correções que não mudam o comportamento do banco. Já no caso de major releases, por exemplo, da versão 9.3 para a 9.4 novas funcionalidades e mudanças em recursos existentes podem ocorrer.

É importante atentar para eventuais lançamentos de novos releases do PostgreSQL. Uma fonte importante de consulta é o Ambiente Virtual de Aprendizagem disponibilizado pela ESR/RNP. Nele serão sempre oferecidos materiais complementares de modo a manter o material do curso o mais atualizado possível.

Arquitetura do PostgreSQL

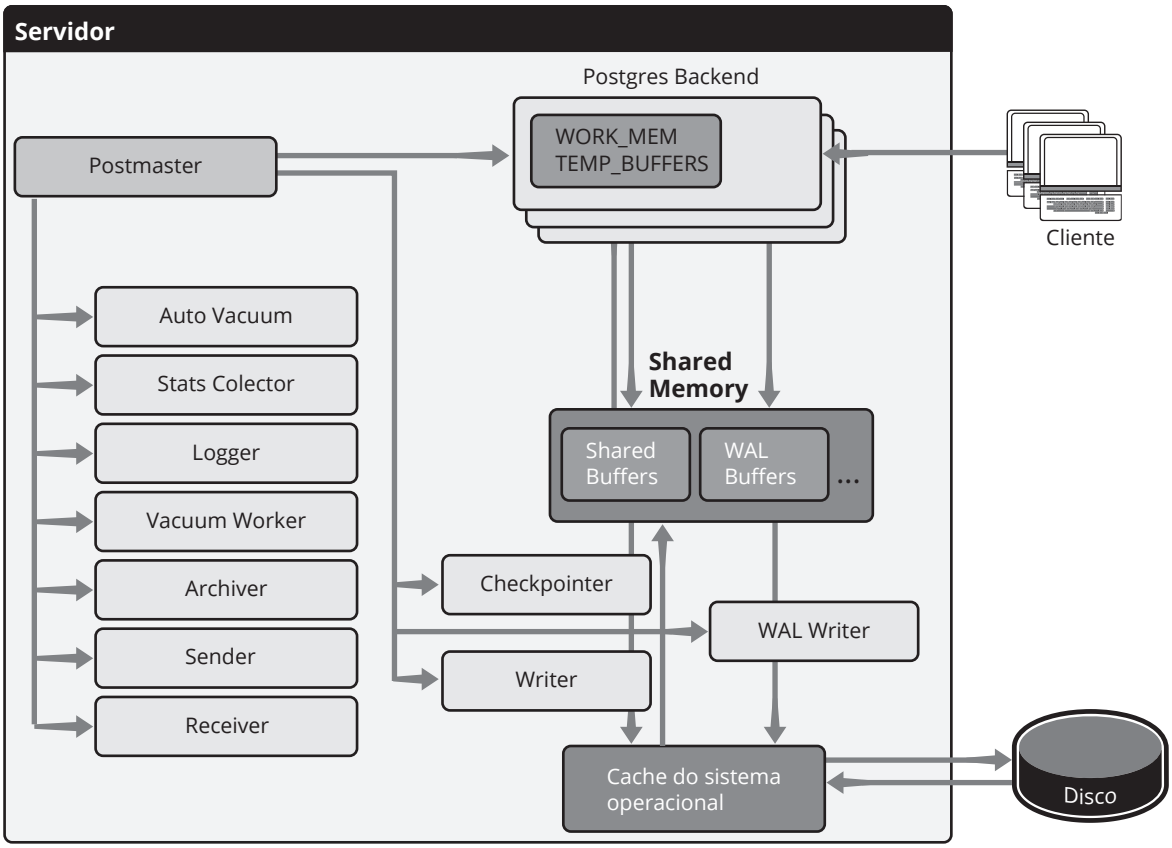


Figura 1.2
Arquitetura Geral do PostgreSQL.

O PostgreSQL é um SGBD multiprocessos, em contraposição aos multithreading. Isso significa que para atender cada conexão de usuário existe um processo, no Sistema Operacional, servindo esse cliente. Esses processos que atendem às conexões são chamados de backend.

Além dos processos para atender requisições de usuários, o PostgreSQL possui outros processos para atender diversas tarefas que o SGBD necessita. A figura 1.3 mostra o menor número de processos possíveis para que o PostgreSQL funcione.

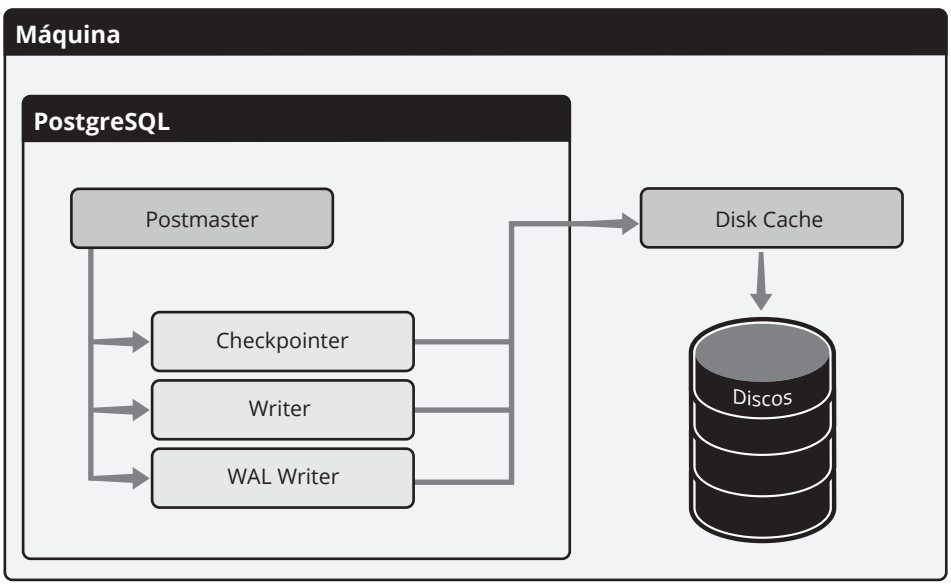


Figura 1.3
Processos básicos do PostgreSQL.

A figura mostra o Postmaster, que é o processo principal e pai de todos os outros processos. Quando o PostgreSQL é iniciado, ele é executado e carrega os demais. O postmaster é um processo do programa executável chamado postgres, porém por compatibilidade com versões anteriores – antes da versão 8, o executável se chamava postmaster de fato – e por identificação do processo principal ele ainda aparece com esse nome.

O Checkpointer é o processo responsável por disparar a operação de Checkpoint, que é a aplicação das alterações do WAL para os arquivos de dados através da descarga de todas as páginas de dados "sujas" da memória (buffers alterados) para disco, na frequência ou quantidade definidos no arquivo de configuração do PostgreSQL, por padrão a cada 5 minutos ou 3 arquivos de log.

O Writer, também conhecido como background writer ou bgwriter, procura por páginas de dados modificadas na memória (shared_buffer) e as escreve para o disco em lotes pequenos. A frequência e o número de páginas que são gravadas por vez estão definidas nos parâmetros de configuração do PostgreSQL, e são por padrão 200ms e 100 páginas.

Em versões anteriores não existia o processo Checkpointer, e o processo Background Writer executava as funções dos dois.

No PostgreSQL, o log de transação é chamado de Write Ahead Log ou simplesmente WAL. O processo WAL writer é responsável por gravar em disco as alterações dos buffers do WAL em intervalos definidos no arquivo de configuração do PostgreSQL.

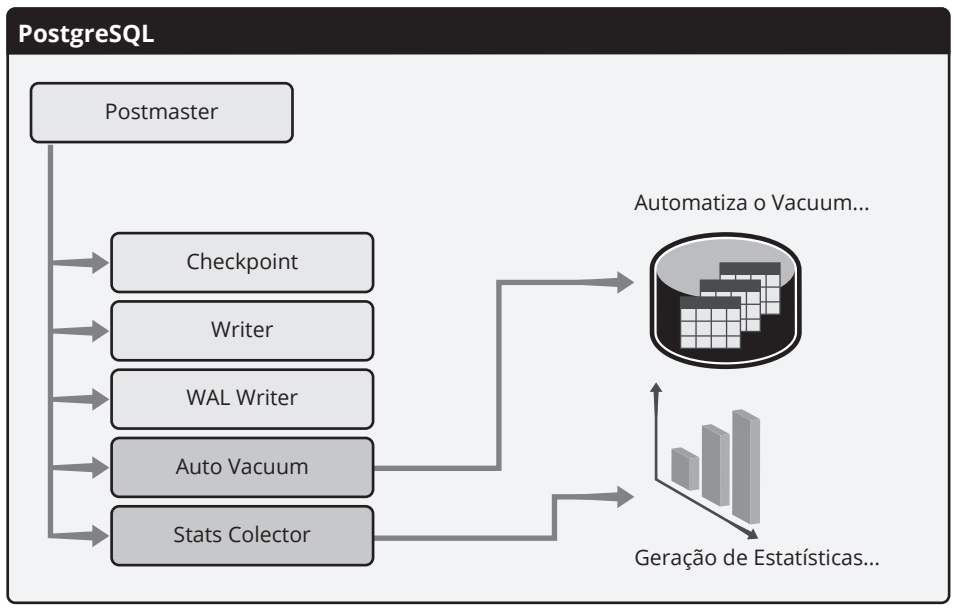


Figura 1.4
Processos em uma configuração padrão.

O AutoVacuum é um daemon que automatiza a execução da operação de Vacuum. Essa é a operação de manutenção mais importante do PostgreSQL, que estudaremos em detalhes adiante, mas basicamente a ideia é análoga a uma desfragmentação.

O Stats Collector é um serviço essencial em qualquer SGBD, responsável pela coleta de estatísticas de uso do servidor, contando acessos a tabelas, índices, blocos em disco ou registros entre outras funções.

Em uma instalação de produção poderão ser vistos mais processos. Vejamos todos os processos que compõem a arquitetura do PostgreSQL e suas funções.

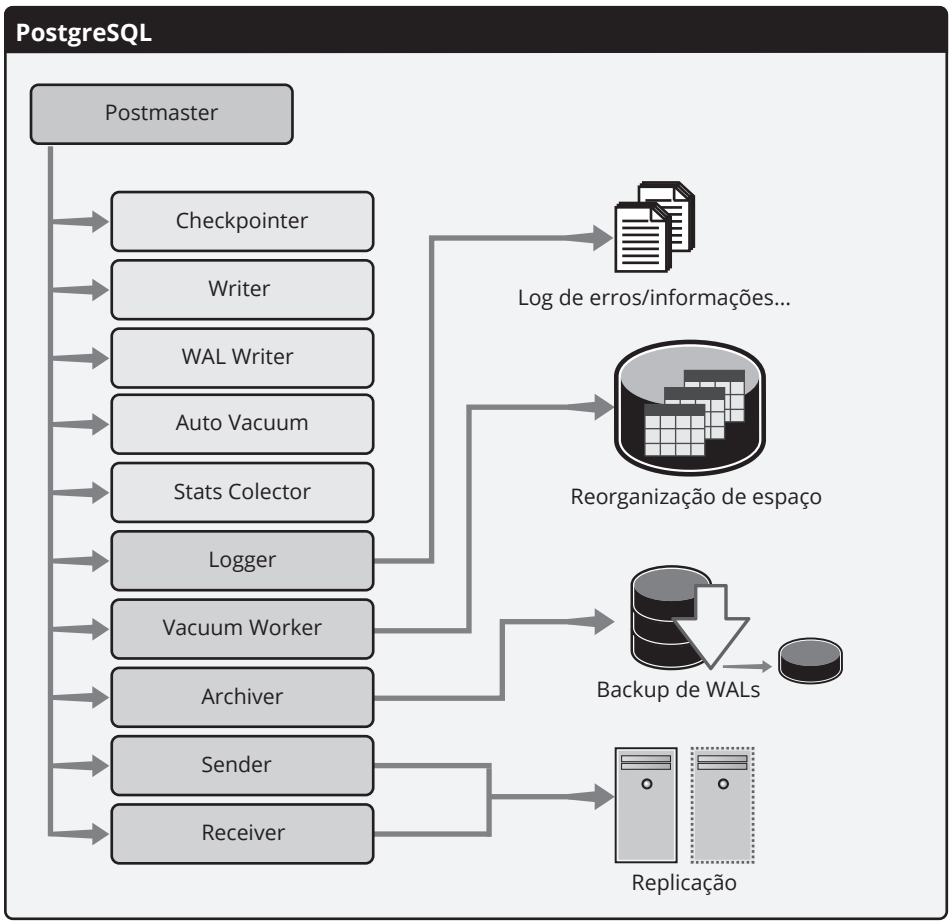


Figura 1.5
Todos os processos utilitários do PostgreSQL.

O Logger é o responsável por registrar o que acontece no PostgreSQL, como tentativas de acesso, erros de queries ou problemas com locks. As informações que serão registradas dependem de diversos parâmetros de configuração.

Para evitar confusão, quando estivermos falando do Log de Transações, vamos nos referir a "WAL (write-ahead log)", e, quando for sobre o Log de Erros/Informações, iremos nos referir apenas a "Log".

O Vacuum Worker é o processo que de fato executa o procedimento de Vacuum, podendo existir diversos dele em execução (três por padrão). Eles são orquestrados pelo processo AutoVacuum.

O processo Archiver tem a função de fazer o backup, ou arquivar, os segmentos de WAL que já foram totalmente preenchidos. Isso permite um Point-In-Time Recovery que estudaremos mais tarde.

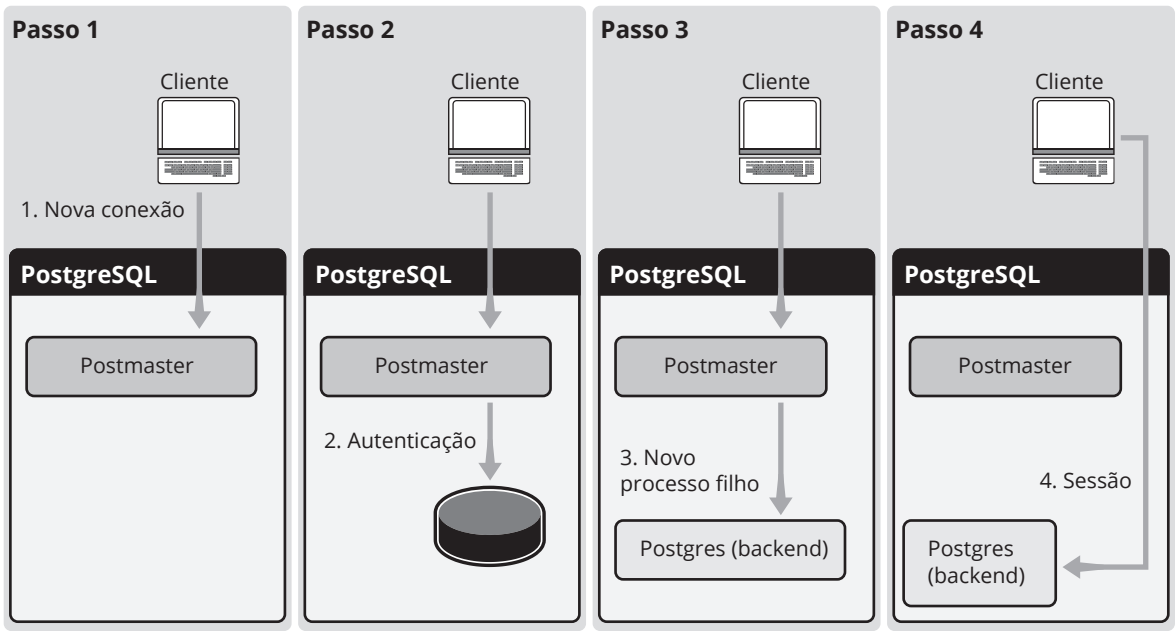
Os processos Sender e Receiver permitem o recurso de Replicação Binária do PostgreSQL, e são responsáveis pelo envio e recebimento, respectivamente, das alterações entre servidores. Esta é uma funcionalidade extremamente útil do PostgreSQL para conseguir Alta Disponibilidade ou Balanceamento de Carga.

Para controlar o comportamento de cada um desses processos, bem como sua execução ou não, e para o comportamento do servidor como um todo, existem diversos parâmetros nos arquivos de configurações que podem ser ajustados. Analisaremos os principais deles mais adiante.

Conexões e processos backends

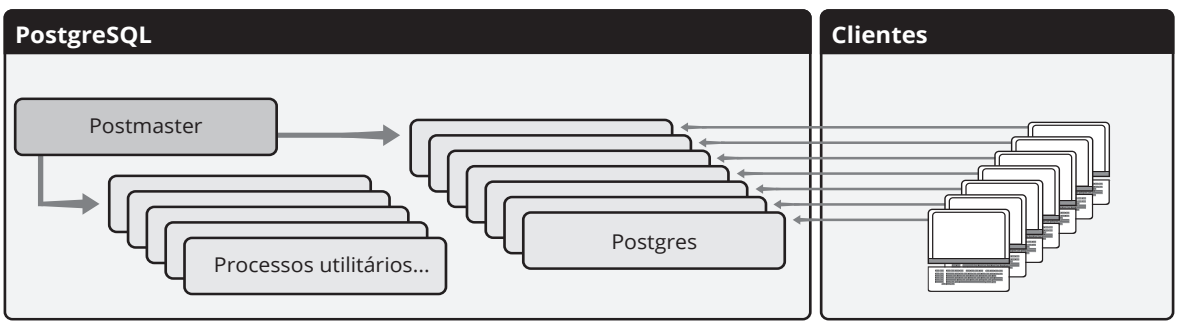
Até agora vimos apenas os processos ditos utilitários, nenhum relacionado às conexões de clientes. Quando um cliente solicita uma conexão ao PostgreSQL, essa requisição é inicialmente recebida pelo processo postmaster, que faz a autenticação e cria um processo filho (backend process) que atenderá as futuras requisições do cliente e processamento das queries sem intervenção do postmaster. O diagrama a seguir ilustra esse mecanismo:

Figura 1.6 Processo de conexão e criação dos backends.



A partir da conexão estabelecida, o cliente envia comandos e recebe os resultados diretamente do processo Postgres que lhe foi associado e que lhe atende exclusivamente. Assim, no lado PostgreSQL, sempre veremos um backend para cada conexão com um cliente.

Figura 1.7 Processos backends.



Clientes podem ser Sistemas, IDEs de acesso a Bancos de Dados, editores de linha de comando como o psql ou Servidores Web, que se conectam ao PostgreSQL via TCP/IP, comumente através da biblioteca LIBPQ em C ou de um driver JDBC em Java.

Em um servidor em funcionamento e com conexões de clientes já estabelecidas, você verá algo semelhante à figura a seguir, mostrando três conexões. No exemplo da imagem, todas as conexões são originadas da mesma máquina, mas cada uma com uma base diferente.

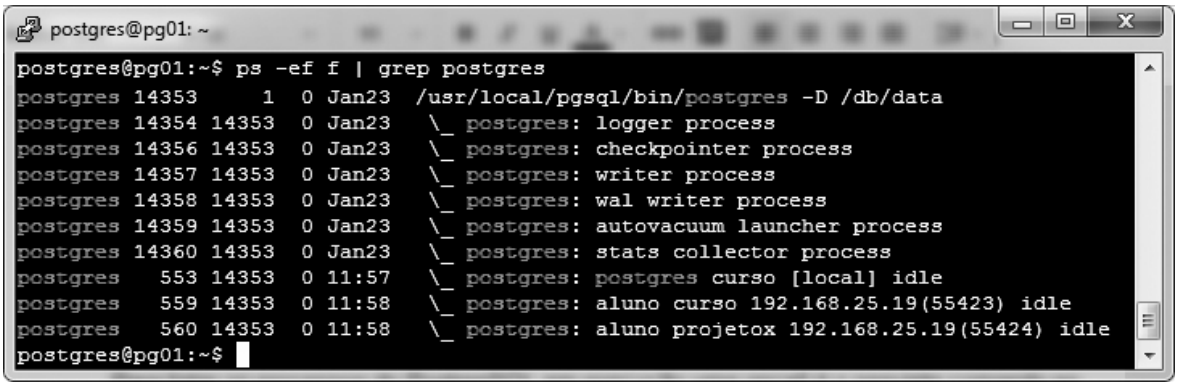
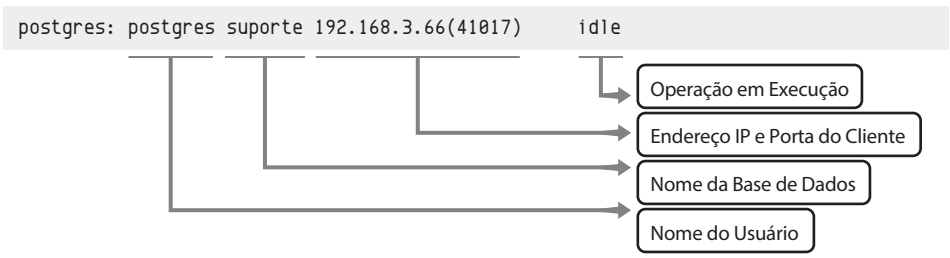


Figura 1.8
Processos do PostgreSQL com conexões estabelecidas.

Para listar os processos do PostgreSQL em execução, uma opção é o seguinte comando no console do Sistema Operacional:

```
$   ps -ef f | grep postgres
```

Ele listará todos os processos do usuário postgres identificados em hierarquia. No caso dos processos backends, são exibidas informações úteis para o Administrador sobre a conexão estabelecida.



Arquitetura de Memória no PostgreSQL

Um componente importante para entender como o PostgreSQL funciona são as áreas de memória utilizadas, principalmente a shared memory.

Normalmente um processo no Sistema Operacional tem sua área de memória exclusiva, chamada de address space. Um segmento de shared memory é uma área de memória que pode ser compartilhada entre processos.

O PostgreSQL utiliza está área para manter os dados mais acessados em uma estrutura chamada shared buffer e permitir que todos os processos tenham acesso a esses dados. Além do shared_buffers, há outras estruturas que o PostgreSQL mantém na shared memory, entre elas o wal buffer, utilizado pelo mecanismo de WAL.

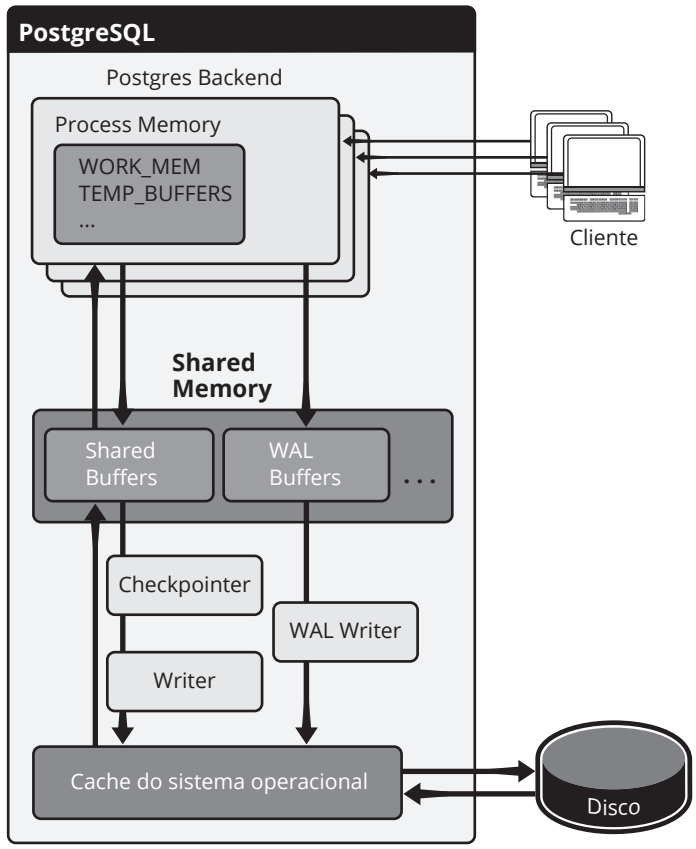


Figura 1.9 Visão geral da arquitetura de memória no PostgreSQL.

Outra área de memória importante é o cache do Sistema Operacional – chamado também de kernel buffer cache, disk cache ou page cache. Basicamente, toda operação de I/O passa pelo cache do SO, mesmo na escrita o dado é carregado para memória e alterado antes de ser gravado em disco. Para todo pedido de leitura de um bloco em disco, o kernel procura pelo dado primeiro no cache e, se não encontrá-lo, buscará em disco e o colocará no cache antes de retornar.

Assim, quando um registro em uma tabela é acessado no PostgreSQL, ele primeiro tenta localizar no shared buffer. Caso não encontre, ele solicita ao Sistema Operacional uma operação de leitura de dados. Porém, isso não significa que ele de fato vá até o disco, já que é possível que o SO encontre o dado no cache. Esse duplo cache é o motivo pelo qual não se configura o tamanho do shared buffer com um valor muito grande, uma vez que o banco confia no cache do SO.

Além das áreas de memória compartilhada, os backends têm sua porção de memória privada que está dividida entre várias áreas para diversas finalidades, como temp_buffers, maintenance_work_mem e, a mais importante delas, a work_mem. Todas podem ter seu tamanho configurável, conforme será visto adiante.

Exercício de fixação: Instalação do PostgreSQL

Na sua opinião, qual a melhor forma de instalar um software como um servidor de Banco de Dados: compilando os fontes ou a partir de um pacote pronto?

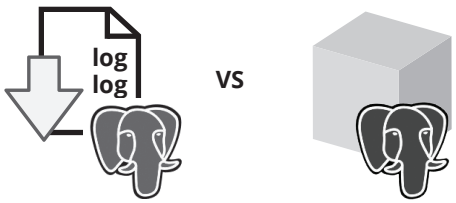


Figura 1.10
Instalação via fontes ou pacotes?

É possível instalar o PostgreSQL de três formas distintas:

- A partir da compilação dos códigos-fonte;
- Por meio de um repositório de uma distribuição Linux;
- Através de um pacote pré-compilado, obtido separadamente.

Apesar de existir uma versão do PostgreSQL para a plataforma Windows, ela está baseada em um mecanismo de emulação. Até porque o PostgreSQL foi concebido para funcionar na arquitetura **Unix**, tendo como pilar o modelo de multiprocessos (diferente do modelo multi-threading do Windows) e funcionalidades específicas do sistema de arquivos que não são encontradas no NTFS para Windows. Assim, em instalações de produção do PostgreSQL, o mais comum (e eficiente) é optar pela plataforma Unix, sendo uma distribuição Linux o mais usual nos dias atuais.

Unix
Sistema Operacional portátil, multitarefa e multiusuário, criado por Ken Thompson, Dennis Ritchie, Douglas McIlroy e Peter Weiner, que trabalhavam nos Laboratórios Bell (Bell Labs), da AT&T.

Instalação a partir dos fontes

A instalação através do códigos-fonte é a forma mais recomendada pelos seguintes motivos:

- O servidor será compilado exatamente para sua plataforma de hardware e software, o que pode trazer benefícios de desempenho. O processo de configuração pode fazer uso de um recurso existente para determinada arquitetura de processador ou SO, permitindo ainda o uso de uma biblioteca diferente, por exemplo;
- Tem-se controle total sobre a versão da sua instalação. Em caso de lançamento de um release para corrigir um bug ou falha de segurança, os respectivos arquivos-fonte podem ser obtidos e compilados sem depender da disponibilização, pelos responsáveis do SO em uso, de um novo pacote refletindo tais correções.

Pré-Requisitos

Para instalar o PostgreSQL a partir dos códigos-fontes, alguns softwares e bibliotecas do Linux são necessários:

- make
- Gcc
- Tar
- Readline
- Zlib

make

GNU make, gmake ou no Linux apenas make, que é um utilitário de compilação. Normalmente já está instalado na maioria das distribuições (versão 3.80 ou superior).

gcc

Compilador da linguagem C. O ideal é usar uma versão recente do gcc, que vem instalado por padrão em muitas distribuições Linux. Outros compiladores C podem também ser utilizados.

tar com gzip ou bzip2

Necessário para descompactar os fontes, normalmente também já instalados na maioria das distribuições Linux. O descompactador a ser utilizado (gzip ou bzip2) dependerá do formato usado (.gz ou .bz2) na compactação dos arquivos-fonte disponíveis.

readline

Biblioteca utilizada pela ferramenta de linha de comando psql para permitir histórico de comandos e outras funções.

É possível não usá-la informando a opção --without-readline no momento da execução do configure, porém é recomendado mantê-la.

zlib

Biblioteca padrão de compressão, usada pelos utilitários de backup pg_dump e pg_restore.

É possível não utilizá-la informando a opção --without-zlib no momento da execução do configure, porém é altamente recomendado mantê-la.

Podem ser necessários outros softwares ou bibliotecas dependendo de suas necessidades e de como será sua instalação. Por exemplo, se você for usar a linguagem pl/perl, então é necessário ter o Perl instalado na máquina. O mesmo vale para pl/python ou pl/tcl ou, se for usar conexões seguras através de ssl, será necessário o openssl instalado previamente.

Aqui assumiremos uma instalação padrão, sem a necessidade desses recursos.

A instalação dos softwares ou bibliotecas é feito da seguinte maneira:

Tabela 1.1
Instalação de softwares ou bibliotecas.

Na distribuição Red Hat/CentOS	Na distribuição Debian/Ubuntu
$ sudo yum install make	$ sudo apt-get install make
$ sudo yum install gcc	$ sudo apt-get install gcc
$ sudo yum install tar	$ sudo apt-get install tar
$ sudo yum install readline-devel	$ sudo apt-get install libreadline6-dev
$ sudo yum install zlib-devel	$ sudo apt-get install zlib1g-dev

O comando *sudo* permite que um usuário comum execute ações que exigem privilégios de superusuário. Quando executado, ele solicitará a senha do usuário para confirmação. Será utilizado diversas vezes durante o curso.

Deve haver um repositório configurado para o seu ambiente para que seja possível usar o yum ou o apt-get.

```
================================================================================
Package             Arch           Version                Repository      Size
================================================================================
Installing:
 gcc                x86_64         4.4.7-4.el6            base            10 M
 readline-devel     x86_64         6.0-4.el6              base            134 k
 zlib-devel         x86_64         1.2.3-29.el6           base            44 k
Installing for dependencies:
 cloog-ppl          x86_64         0.15.7-1.2.el6         base            93 k
 cpp                x86_64         4.4.7-4.el6            base            3.7 M
 glibc-devel        x86_64         2.12-1.132.el6         base            978 k
 glibc-headers      x86_64         2.12-1.132.el6         base            608 k
 kernel-headers     x86_64         2.6.32-431.11.2.el6    updates         2.8 M
 libgomp            x86_64         4.4.7-4.el6            base            118 k
 mpfr               x86_64         2.4.1-6.el6            base            157 k
 ncurses-devel      x86_64         5.7-3.20090208.el6     base            642 k
 ppl                x86_64         0.10.2-11.el6          base            1.3 M
Transaction Summary
================================================================================
Install      12 Package(s)
Total download size: 21 M
Installed size: 41 M
Is this ok [y/N]:
```

Figura 1.11
Instalação dos pré-requisitos com yum no CentOS.

Obtendo o código fonte

No site oficial do PostgreSQL estão disponíveis os arquivos-fonte para diversas plataformas, juntamente com instruções para instalação em cada uma delas a partir dos fontes, de repositórios ou de pacotes: http://www.postgresql.org/download/

O código fonte do PostgreSQLpode ser obtido diretamente na internet no link indicado no AVA.

Neste exemplo utilizaremos como base a versão 9.3.4. Verifique no Ambiente Virtual de Aprendizagem disponibilizado pela ESR/RNP se existem instruções para instalação de versões mais recentes.

De modo a ganhar tempo, o download do pacote foi feito previamente e já estará disponível nos computadores do laboratório. Siga as instruções a seguir para encontrá-lo:

```
$   cd /usr/local/src/
$   sudo tar -xvf postgresql-9.3.4.tar.gz
$   cd postgresql-9.3.4/
```

Compilação e Instalação a partir dos fontes

Configuração

Após descompactar os fontes e entrar no diretório criado, o primeiro passo é executar o configure, para avaliar as configurações do seu ambiente e verificar dependências:

```
$   ./configure
```

Para listar os parâmetros que podem ser definidos execute o configure com -help ou acesse a página com informações mais detalhadas através do link indicado no AVA.

```
  --without-PACKAGE       do not use PACKAGE (same as --with-PACKAGE=no)
  --with-template=NAME    override operating system template
  --with-includes=DIRS    look for additional header files in DIRS
  --with-libraries=DIRS   look for additional libraries in DIRS
  --with-libs=DIRS        alternative spelling of --with-libraries
  --with-pgport=PORTNUM   set default port number [5432]
  --with-blocksize=BLOCKSIZE
                          set table block size in kB [8]
  --with-segsize=SEGSIZE  set table segment size in GB [1]
  --with-wal-blocksize=BLOCKSIZE
                          set WAL block size in kB [8]
  --with-wal-segsize=SEGSIZE
                          set WAL segment size in MB [16]
  --with-CC=CMD           set compiler (deprecated)
  --with-tcl              build Tcl modules (PL/Tcl)
  --with-tclconfig=DIR    tclConfig.sh is in DIR
  --with-perl             build Perl modules (PL/Perl)
  --with-python           build Python modules (PL/Python)
```

Figura 1.12
configure': diversos parâmetros para adaptar a instalação do PostgreSQL.

A execução sem parâmetros configurará o diretório de instalação do PostgreSQL em /usr/local/pgsql. Para alterar o diretório alvo deve-se informar o parâmetro --prefix=<diretório>. Além do diretório, há diversos parâmetros que podem ser informados caso seja necessário alterar o comportamento da sua instalação.

Usar diretório com a versão identificada e usar link simbólico
/usr/local/pgsql9.3.4
/usr/local/pgsql -> pgsql9.3.4

Compilação

Definida a configuração do ambiente o passo seguinte é a compilação, através do comando make:

```
$   make
```

Isso pode levar alguns minutos e, não havendo problemas, será exibida uma mensagem com o final "Ready to Install", conforme pode ser visto na figura 1.13.

```
gcc -O2 -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-statement
 -Wendif-labels -Wmissing-format-attribute -Wformat-security -fno-strict-aliasin
g -fwrapv -fpic -L../../src/port -L../../src/common -Wl,--as-needed -Wl,-rpath,'
/usr/local/pgsql/lib',--enable-new-dtags  -shared -o dummy_seclabel.so dummy_sec
label.o
make[3]: Saindo do diretório `/home/curso/softwares/postgresql-9.3.4/contrib/dum
my_seclabel'
cp ../../../contrib/dummy_seclabel/dummy_seclabel.so dummy_seclabel.so
make[2]: Saindo do diretório `/home/curso/softwares/postgresql-9.3.4/src/test/re
gress'
make[1]: Saindo do diretório `/home/curso/softwares/postgresql-9.3.4/src'
make -C config all
make[1]: Entrando no diretório `/home/curso/softwares/postgresql-9.3.4/config'
make[1]: Nada a ser feito para `all'.
make[1]: Saindo do diretório `/home/curso/softwares/postgresql-9.3.4/config'
All of PostgreSQL successfully made. Ready to install.
[curso@pg02 postgresql-9.3.4]$
```

Figura 1.13
Compilação dos fontes com make.

Teste

É possível testar a instalação em seu ambiente, bastando executar:

```
$   make check
```

O make check é bastante interessante, fazendo a instalação e execução de um servidor PostgreSQL, além de uma série de testes nesta instância temporária. O resultado deve ser algo como:

```
=======================
 All 131 tests passed.
=======================
```

Para mais informações, sobre testes de instalação consulte o link disponibilizado no AVA

```
     alter_table              ... ok
     sequence                 ... ok
     polymorphism             ... ok
     rowtypes                 ... ok
     returning                ... ok
     largeobject              ... ok
     with                     ... ok
     xml                      ... ok
test stats                    ... ok
============== shutting down postmaster                    ==============
=======================
 All 136 tests passed.
=======================
make[1]: Saindo do diretório `/home/curso/softwares/postgresql-9.3.4/src/test/re
gress'
[curso@pg02 postgresql-9.3.4]$
```

Figura 1.14
Teste de regressão com make check.

O teste pode falhar, e neste caso, o resultado deve ser analisado para avaliar a gravidade do problema e formas de resolvê-lo.

Instalação

```
$   sudo make install
```

Esse comando copiará os arquivos para o diretório de instalação (se foi deixado o caminho padrão "/usr/local/pgsql", então é necessário o sudo). O resultado deve ser a exibição da mensagem "PostgreSQL installation complete", conforme a figura 1.15.

```
/usr/bin/install -c  pg_regress '/usr/local/pgsql/lib/pgxs/src/test/regress/pg_r
egress'
make[2]: Saindo do diretório `/home/curso/softwares/postgresql-9.3.4/src/test/re
gress'
/bin/mkdir -p '/usr/local/pgsql/lib/pgxs/src'
/usr/bin/install -c -m 644 Makefile.global '/usr/local/pgsql/lib/pgxs/src/Makefi
le.global'
/usr/bin/install -c -m 644 Makefile.port '/usr/local/pgsql/lib/pgxs/src/Makefile
.port'
/usr/bin/install -c -m 644 ./Makefile.shlib '/usr/local/pgsql/lib/pgxs/src/Makef
ile.shlib'
/usr/bin/install -c -m 644 ./nls-global.mk '/usr/local/pgsql/lib/pgxs/src/nls-gl
obal.mk'
make[1]: Saindo do diretório `/home/curso/softwares/postgresql-9.3.4/src'
make -C config install
make[1]: Entrando no diretório `/home/curso/softwares/postgresql-9.3.4/config'
/bin/mkdir -p '/usr/local/pgsql/lib/pgxs/config'
/usr/bin/install -c -m 755 ./install-sh '/usr/local/pgsql/lib/pgxs/config/instal
l-sh'
make[1]: Saindo do diretório `/home/curso/softwares/postgresql-9.3.4/config'
PostgreSQL installation complete.
[curso@pg02 postgresql-9.3.4]$
```

Figura 1.15 PostgreSQL instalado com make install.

Instalação de extensões

Entre as mais interessantes, do ponto de vista de administração, estão:

- dblink: biblioteca de funções que permite conectar uma base PostgreSQL com outra, estando estas ou não no mesmo servidor.
- pg_buffercache: cria uma view que permite analisar os dados no shared buffer, possibilitando verificar o nível de acerto em cache por base de dados.
- pg_stat_statements: permite consultar online as principais queries executadas no banco, por número de execuções, total de registros, tempo de execução etc.

No diretório contrib podem ser encontrados diversos módulos opcionais, chamados de extensões, que podem ser instalados junto ao PostgreSQL. Existem utilitários, tipos de dados e funções para finalidades específicas, como tipos geográficos.

Para instalar uma extensão, vá para o diretório correspondente em contrib/<extensão> e utilize o make. Por exemplo, para instalar o pg_buffercache:

```
$   cd contrib/pg_buffercache/
$   make
$   sudo make install
```

É necessário também registrar a extensão no PostgreSQL, conectando na base na qual esta será instalada de modo a executar o comando:

```
CREATE EXTENSION pg_buffercache;
```

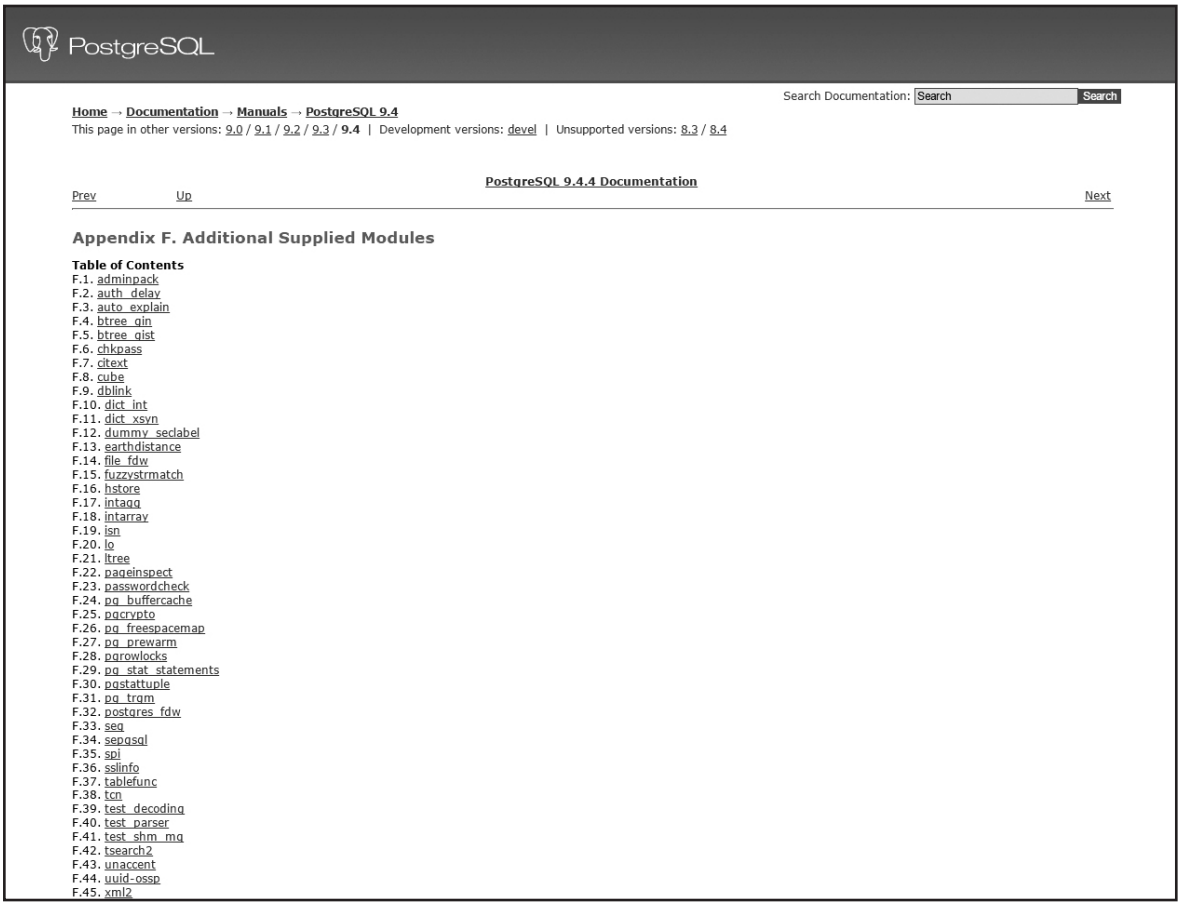


Figura 1.16
Lista oficial de extensões e PGXN, diretório de extensões diversas.

Além das extensões fornecidas juntamente com o PostgreSQL, a PostgreSQL Extension Network (PGXN) é um diretório de extensões open source para as mais variadas finalidades. Acesse: http://pgxn.org/

Removendo o PostgreSQL

Use o make para remover a instalação do PostgreSQL, incluindo todos os arquivos criados durante o processo de configuração e compilação. É necessário estar no diretório raiz dos fontes onde foi feita a compilação:

```
$   cd /usr/local/src/postgresql-9.3.4
$   sudo make uninstall
$   make distclean
```

O make não remove os diretórios criados. Para removê-los, é necessário executar o comando:

```
$   sudo rm -Rf /usr/local/pgsql
```

Instalação a partir de repositórios

A maneira mais simples de instalar o PostgreSQL é através do repositório da sua distribuição Linux. Talvez não seja o mais apropriado para um ambiente de produção, já que o seu servidor dependerá sempre da versão que foi empacotada pela distribuição Linux. Pode ser útil em ambientes de teste ou desenvolvimento, ou para alguma aplicação menos crítica.

Instalação

O processo pelo repositório do Ubuntu instala o PostgreSQL, cria o usuário postgres, inicializa o diretório de dados e executa o banco. O diretório de instalação é o /var/lib/postgresql/<versao>/main/ e a área de dados (diretório data) é criada abaixo deste.

```
Adding user postgres to group ssl-cert
Building PostgreSQL dictionaries from installed myspell/hunspell package ...
update-rc.d: warning  /etc/init.d/postgresql missing LSD information
update-rc.d: see <http //wiki.debian.org/LSBlnitScript >
Starting PostgreSQL: ok
Setting up postgresql-9.1 (9.I.II-OubuntuO.12.04) ...
Creating new cluster (configuration: /etc/pa tgresql/9.1/main, data: /var/lib/po
tgre ql/9.I/main)...
Moving configuration file /var/lib/postgresql/9.1/main/postgresql.conf to /etc/p
ostgresql/9.I/main...
Moving configuration file /var/lib/postgresql/9.1/main/pg_hba.conf to /etc/postg
resql/9.I/main...
Moving configuration file /var/lib/postgresql/9.1/main/pg_ident.conf to /etc/pos
tgresql/9.I/main...
Configuring postgresql.conf tO use port 5433. . .
update-alternatives: using /usr/share/postgresql/9.1/man/man1/postmaster.1.gz tO
provide /usr/share/man/man1/postmaster.1.gz (postmaster.1.gz) in auto mode.
Starting PostgreSQL: ok
Setting up postgresql (9.I+l29ubuntul) . . .
Processing triggers for libc-bin ...
Idconfig deferred processing now taking place
curso@pg01:-$
```

Figura 1.17 Instalação por repositório no Ubuntu.

No caso do CentOS o PostgreSQL é apenas instalado. As demais tarefas devem ser executadas manualmente. A instalação é feita usando o padrão de diretórios da distribuição, por exemplo, programas no /usr/bin e bibliotecas em /usr/lib.

Red Hat/CentOS

```
$   sudo yum install postgresql-server
```

> Red Hat e CentOS possuem versões do PostgreSQL muito defasadas em seus repositórios

Debian/Ubuntu

```
$   sudo apt-get install postgresql
```

Para obter detalhes da versão disponível no repositório use o seguinte comando:

```
$   yum info postgresql-server                (Red Hat/CentOS)
$   apt-cache show postgresql
(Debian/Ubuntu)
```

```
[curso@pg02 postgresql-9.3.4]$ yum info postgresql-server
Loaded plugins: fastestmirror
Determining fastest mirrors
 * base: centos.xfree.com.ar
 * extras: centos.xfree.com.ar
 * updates: centos.xfree.com.ar
base                                     | 3.7 kB     00:00
extras                                   | 3.4 kB     00:00
updates                                  | 3.4 kB     00:00
updates/primary_db                       | 2.3 MB     00:04
Available Packages
Name        : postgresql-server
Arch        : i686
Version     : 8.4.20
Release     : 1.el6_5
Size        : 3.4 M
Repo        : updates
Summary     : The programs needed to create and run a PostgreSQL server
URL         : http://www.postgresql.org/
License     : PostgreSQL
Description : The postgresql-server package includes the programs needed to create
            : and run a PostgreSQL server, which will in turn allow you to create
            : and maintain PostgreSQL databases.  PostgreSQL is an advanced
            : Object-Relational database management system (DBMS) that supports
            : almost all SQL constructs (including transactions, subselects and
            : user-defined types and functions). You should install
            : postgresql-server if you want to create and maintain your own
            : PostgreSQL databases and/or your own PostgreSQL server. You also need
            : to install the postgresql package.
[curso@pg02 postgresql-9.3.4]$
```

Figura 1.18 PostgreSQL com versão defasada no repositório do CentOS.

É possível instalar versões atualizadas do PostgreSQL alterando o repositório utilizado. O PostgreSQL mantém repositórios compatíveis com RedHat/CentOS e Debian/Ubuntu tanto com versões mais recentes quanto com versões mais antigas.

Remoção

Red Hat/CentOS	Debian/Ubuntu
`$ sudo yum erase postgresql-server`	`$ sudo apt-get --purge remove postgresql-9.3`

Instalação de extensões

Os módulos opcionais, quando instalados pelo repositório, contêm em um único pacote todas as extensões.

Red Hat/CentOS	Debian/Ubuntu
`$ sudo yum install postgresql-contrib`	`$ sudo apt-get install postgresql-contrib-9.3`

Também pode ser necessário criar a extensão na base com o comando *CREATE EXTENSION*.

Instalação a partir de pacotes

Se por alguma necessidade específica for necessário instalar um pacote em particular, é possível instalar o PostgreSQL a partir de pacotes .RPM para Red Hat/CentOS ou .DEB para Debian/Ubuntu.

Exemplificaremos utilizando pacotes indicados no site oficial do PostgreSQL do PostgreSQL Global Development Group (PGDG) e do OpenSCG.

Instalação

A localização da instalação do PostgreSQL dependerá do pacote sendo utilizado, mas é possível informar para o rpm o parâmetro --prefix para definir o diretório.

Red Hat/CentOS

```
$   sudo rpm -ivH postgresql93-server-9.3.4-1PGDG.rhel6.x86_64.rpm
```

Debian/Ubuntu

```
$   sudo dpkg -i postgres_9.3.4-1.amd64.openscg.deb
```

Os procedimentos pós-instalação variam de pacote para pacote. Alguns, como o OpenSCG, apresentam uma espécie de wizard na primeira execução que cria scripts de inicialização com nomes personalizados, cria o usuário postgres e pode criar também o diretório de dados automaticamente. Recomendarmos ler as instruções da documentação do pacote para conhecer melhor seu processo de instalação.

Remoção

Se houver processos em execução, eles serão parados. O diretório de dados não é excluído.

Red Hat/CentOS

```
$   sudo rpm -e postgres93
```

Debian/Ubuntu

```
$   sudo dpkg --remove postgres93
```

Instalação de extensões

Como na instalação pelo repositório, os módulos opcionais contêm em um único pacote todas as extensões.

Red Hat/CentOS

```
$   sudo rpm -ivH postgresql93-contrib-9.3.4-1PGDG.rhel6.x86_64.rpm
```

Debian/Ubuntu

```
$   sudo dpkg -i postgresql-contrib-9.3-9.3.4-0ppa1.pgdg+1~precise
```

Também pode ser necessário criar a extensão na base com o comando CREATE EXTENSION.

Resumo da instalação a partir dos fontes:

```
$   sudo yum install make gcc tar readline-devel zlib-devel
$   cd /usr/local/src/
$   sudo wget ftp://ftp.postgresql.org/pub/source/v9.3.4/postgresql-9.3.4.tar.gz
$   sudo tar -xvf postgresql-9.3.4.tar.gz
$   cd postgresql-9.3.4/
$   ./configure
$   make
$   sudo make install
```

2

Operação e configuração

objetivos

Conhecer a operação e controles básicos do PostgreSQL; Aprender a inicialização da área de dados, como iniciar e parar o banco, recarregar configurações, verificar status e outros procedimentos; Ver os parâmetros de configuração do PostgreSQL e os escopos em que estes podem ser aplicados.

conceitos

Superusuário; Área de Dados; Variáveis de Ambiente; Utilitários pg_ctl e initdb, PID e Sinais de Interrupção de Processos.

Operação do Banco de Dados

Concluído o processo de instalação do PostgreSQL, é necessário colocá-lo em operação. Para tanto, os seguintes passos devem ser tomados:

- Criar conta do superusuário.
- Configurar variáveis de ambiente.
- Inicializar área de dados.
- Operações básicas do banco.
- Configuração.

Criação da conta do Superusuário

Antes de executar o PostgreSQL, devemos criar a conta sob qual o serviço será executado e que será utilizada para administrá-lo. Isso exige privilégios de superusuário, demandando novamente o uso de sudo.

O comando a seguir cria a conta do usuário, indicando que:

- A criação do diretório do usuário deverá ser feita em home;
- O interpretador shell será o bash e;
- A criação de um grupo com o mesmo nome do usuário.

```
aluno$      sudo useradd --create-home --user-group --shell /bin/bash postgres
```

Em seguida, temos o comando para definir uma senha para o novo usuário postgres:

```
aluno$      sudo passwd postgres
aluno$      sudo passwd postgres
```

Atenção: será primeiro solicitada a senha do usuário aluno para fazer o sudo, depois a nova senha para o usuário postgres duas vezes:

```
[sudo] password for curso:
Enter new UNIX password:
Retype new UNIX password:
```

O nome da conta pode ser outro, mas por padrão é assumido como “postgres”.

Em um servidor de produção, pode haver regras específicas para a criação de usuários. Verifique com o administrador do seu ambiente.

Na sessão que trata da administração de usuários e segurança, apresentaremos em detalhes os comandos relacionados a essas atividades. Por hora, para facilitar os procedimentos de operação e configuração do Banco de Dados, utilizaremos o comando a seguir para fornecer algumas permissões para o usuário postgres no filesystem /db, que será utilizado pelo Banco de Dados:

```
aluno$      sudo chown -R postgres /db
```

Definindo variáveis de ambiente

Para facilitar a administração, é útil definir algumas variáveis de ambiente para que não seja necessário informar o caminho completo dos programas ou da área de dados em todos os comandos que forem ser executados.

Primeiro, o diretório com os binários do PostgreSQL deve ser adicionado ao path do usuário postgres. Também vamos definir a variável PGDATA, que indica o diretório de dados do PostgreSQL.

De agora em diante, devemos utilizar sempre o usuário postgres.

Conecte-se com o usuário postgres e edite o arquivo *.bashrc* que está no home do usuário. Para tanto, utilize os seguintes comandos:

```
aluno$              su - postgres
postgres$   vi ~/.bashrc
```

O último comando acima aciona o editor de textos vi para que o arquivo .bashrc seja editado. As seguintes linhas devem ser adicionadas a este arquivo:

```
PATH=$PATH:/usr/local/pgsql/bin:$HOME/bin
PGDATA=/db/data/
export PATH PGDATA
```

As alterações passarão a valer para as próximas vezes em que o computador for inicializado. Para que passem a ter efeito na sessão atual, é necessário executar o comando:

```
postgres$   source .bashrc
```

Políticas específicas para definições de variáveis e informações de perfis de usuários podem ser estabelecidas através de diretivas registradas nos arquivos *.bash_profile* ou *.profile*, variando de instituição para instituição. Devemos considerar também a possibilidade de uso de outro interpretador shell que não seja o bash. Verifique esses pontos com o administrador do seu ambiente.

Inicializando a área de dados

Para que o PostgreSQL funcione, é necessário inicializar o diretório de dados, ou área de dados, chamada também de cluster de bancos de dados. Essa área é o diretório que conterá, a príncipio, todos os dados do banco e toda a estrutura de diretórios e arquivos de configuração do PostgreSQL.

Se o servidor foi instalado a partir de repositórios ou pacotes, é possível que a área de dados já tenha sido criada e estará provavelmente abaixo do diretório de instalação do próprio PostgreSQL.

Assumindo que nossa instalação foi feita a partir dos fontes, devemos criar esta área.

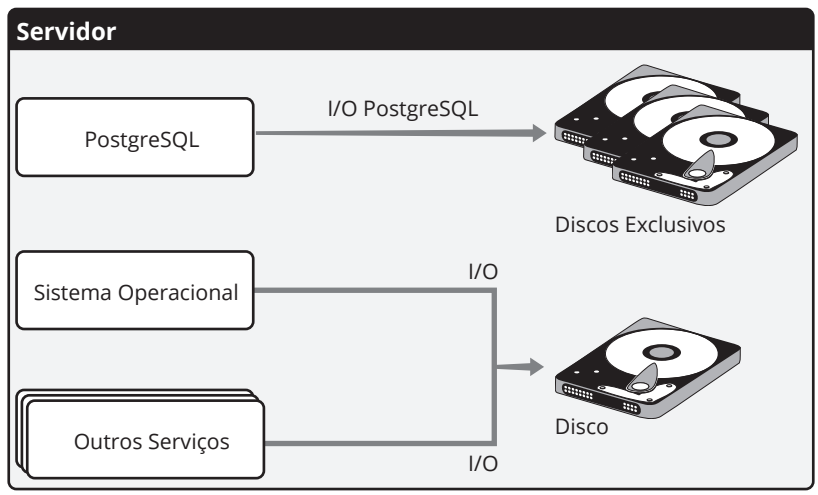


Figura 2.1
Discos exclusivos para o Banco de Dados.

Em um ambiente real com dados de produção, não se deve deixar as bases de dados na mesma partição, nem no mesmo disco, da instalação do PostgreSQL ou de outros softwares e do Sistema Operacional. Isso se deve por questões de desempenho e manutenção.

Do ponto de vista de desempenho, o acesso ao disco é um dos maiores desafios dos administradores de Banco de Dados. Manter bases de dados em um disco concorrendo com outros softwares torna esse trabalho ainda mais difícil e por vezes cria problemas difíceis de detectar.

Do ponto de vista de manutenção, a área de dados de um servidor de Bancos de Dados precisa frequentemente ser expandida, podendo também ter o filesystem alterado, ou ser transferida para um disco mais rápido etc. Em sistemas de alto desempenho é necessário ter diversos discos para o Banco de Dados, um ou mais para os dados, para o WAL, para log, para índices e até mesmo um disco específico para uma determinada tabela.

Essas características dinâmicas exigidas pelo Banco de Dados poderiam gerar conflitos com instalações ou dados de outros softwares.

Usaremos o termo genérico “disco”, mas podem se tratar de discos locais da máquina ou áreas de storage em uma rede SAN.

Feitas as considerações sobre desempenho e manutenção, a criação da área de dados na partição /db pode ser feita através do comando:

```
postgres$            initdb
```

Outra alternativa, caso não tivéssemos definido a variável PGDATA ou quiséssemos inicializar outro diretório, seria usar esse mesmo comando indicando o local para a criação de área de dados:

```
postgres$            initdb -D /db/data
```

Pronto, o PostgreSQL está preparado para executar.

Tome um momento analisando a saída do initdb para entender o que é o processo de inicialização da área de dados e explore um pouco o diretório "/db/data" para começar a se familiarizar. Estudaremos a estrutura completa adiante.

```
[postgres@p
g02 ~]$ initdb
The files belonging to this database system will be owned by user "postgres".
This user must also own the server process.
The database cluster will be initialized with locale "pt_BR.UTF-8".
The default database encoding has accordingly been set to "UTF8".
The default text search configuration will be set to "portuguese".
Data page checksums are disabled.
fixing permissions on existing directory /db/data ... ok
creating subdirectories ... ok
selecting default max_connections ... 100
selecting default shared_buffers ... 128MB
creating configuration files ... ok
creating template1 database in /db/data/base/1 ... ok
initializing pg_authid ... ok
initializing dependencies ... ok
creating system views ... ok
loading system objects' descriptions ... ok
creating collations ... ok
creating conversions ... ok
creating dictionaries ... ok
setting privileges on built-in objects ... ok
creating information schema ... ok
loading PL/pgSQL server-side language ... ok
vacuuming database template1 ... ok
copying template1 to template0 ... ok
copying template1 to postgres ... ok
syncing data to disk ... ok
WARNING: enabling "trust" authentication for local connections
You can change this by editing pg_hba.conf or using the option -A, or
--auth-local and --auth-host, the next time you run initdb.
Success. You can now start the database server using:
    postgres -D /db/data/
or
    pg_ctl -D /db/data/ -l logfile start
[postgres@pg02 ~]$
```

Figura 2.2
Saída do initdb: tarefas executadas na inicialização da área de dados.

Iniciando o PostgreSQL

Há diversas maneiras de iniciar o PostgreSQL. O nome do executável principal é postgres, assim podemos iniciar o banco apenas chamando-o:

```
$   postgres -D /db/data
```

Ou, se a variável PGDATA tiver sido definida, simplesmente:

```
$   postgres
```

Esse comando executará o PostgreSQL em primeiro plano, em sua console. Se for executado um "Ctrl+C" o servidor vai parar. Para executar o Banco de Dados em background, utilize o comando:

```
$   postgres &
```

Para capturar a saída padrão (stdout) e a saída de erros (stderr), e enviá-los para um arquivo de log:

```
$   postgres > /db/data/pg_log/postgresql.log 2>&1 &
```

Porém, a forma mais simples de iniciar o banco em background e com log é usando o utilitário pg_ctl. Para isso é preciso indicar no arquivo de configuração do PostgreSQL (*postgresql.conf*) onde deverão ser armazenados os arquivos de log. Assim, a execução do servidor pode ser rotineiramente comandada por:

```
$   pg_ctl start
```

Importante: o PostgreSQL só executará se for acionado pelo usuário postgres. Nem mesmo como root será possível acioná-lo.

Configurar execução automática

Provavelmente você deseja que o serviço de Banco de Dados suba automaticamente quando o servidor for ligado ou reiniciado.

Para configurar serviços no Linux que devem iniciar automaticamente, o mais comum é adicionar um script de controle no diretório "/etc/init.d/".

Junto com os fontes do PostgreSQL vem um modelo pronto desse script. Precisamos apenas copiá-lo para o diretório correto e editá-lo para ajustar alguns caminhos.

É importante lembrar que em geral o usuário postgres não tem permissão no diretório /etc/init.d/, sendo necessário usar o usuário root ou algum usuário que possa executar sudo. Os seguintes passos devem ser seguidos:

```
$   cd /usr/local/src/postgresql-9.3.4/contrib/start-scripts/
$   sudo cp linux /etc/init.d/postgresql
$   sudo vi /etc/init.d/postgresql
```

Verifique os parâmetros prefix e PGDATA para que estejam de acordo com sua instalação. Se as instruções do capítulo anterior tiverem sido corretamente seguidas estes valores deverão ser:

```
prefix = /usr/local/pgsql
PGDATA = /db/data/
```

Reserve um minuto analisando o restante do script para entender sua função e salve-o. Em seguida forneça permissão de execução.

```
$   sudo chmod +x /etc/init.d/postgresql
```

Por fim, devemos instalar o script de inicialização conforme indicado a seguir:

Red Hat/CentOS	Debian/Ubuntu
`$  sudo chkconfig --add postgresql`	`$  sudo update-rc.d postgresql defaults`

Reinicie seu sistema para testar se o PostgreSQL iniciará junto com o Sistema Operacional.

A família Debian/Ubuntu pode utilizar um sistema de inicialização de serviços diferente, chamado Upstart. Verifique com o administrador do seu ambiente.

Parando o PostgreSQL

O PostgreSQL pode ser parado pelo tradicional comando *kill* do Linux. Primeiro você deve obter o PID (ID do processo) do processo principal do PostgreSQL. Você pode obtê-lo com o comando *ps*:

```
$   ps -ef f | grep postgres
```

```
postgres@pg01:~$ ps -ef f | grep postgres
postgres   815     1  0 22:59 ?  S    0:00 /usr/local/pgsql/bin/postmaster -D /d
b/data
postgres   927   815  0 22:59 ?   Ss   0:00  \_ postgres: logger process
postgres   948   815  0 22:59 ?   Ss   0:00  \_ postgres: checkpointer process
postgres   950   815  0 22:59 ?   Ss   0:00  \_ postgres: writer process
postgres   951   815  0 22:59 ?   Ss   0:00  \_ postgres: wal writer process
postgres   952   815  0 22:59 ?   Ss   0:00  \_ postgres: archiver process
postgres   953   815  0 22:59 ?   Ss   0:00  \_ postgres: stats collector
process
postgres@pg01:~$
```

Figura 2.3
Obtendo o PID do processo principal do PostgreSQL através do 'ps'.

Outra alternativa é consultar o arquivo *postmaster.pid*:

```
$   cat /db/data/postmaster.pid
```

```
postgres@pg01:~$ cat /db/data/postmaster.pid
815
/db/data
1396317578
5432
/tmp
*
  5432001         0
postgres@pg01:~$
```

Figura 2.4
Obtendo o PID através do arquivo 'postmaster.pid'.

O *kill* funciona enviando notificações para o processo. Essas notificações são chamadas sinais.

Há três sinais possíveis para parar o serviço do PostgreSQL através do *kill*:

- TERM;
- INT;
- QUIT.

TERM: modo smart shutdown — o banco não aceitará mais conexões, mas aguardará todas as conexões existentes terminarem para parar.

```
$ kill -TERM 1440
```

INT: modo fast shutdown — o banco não aceitará conexões e enviará um sinal TERM para todas as conexões existentes abortarem suas transações e fecharem. Também aguardará essas conexões terminarem para parar o banco.

```
$ kill -INT 1440
```

QUIT: modo immediate shutdown — o processo principal envia um sinal QUIT para todas as conexões terminarem imediatamente e também sai abruptamente. Quando o banco for iniciado, entrará em recovery para desfazer as transações incompletas.

```
    $ kill -QUIT 1440
```

Nunca use o sinal SIGKILL, mais conhecido como kill -9, já que isso impede o postmaster de liberar segmentos da shared memory e semáforos, além de impedi-lo de enviar sinais para os processos filhos, que terão de ser eliminados manualmente.

A forma mais "elegante" de parar o Banco de Dados é também através do comando *pg_ctl*. Não é necessário nem saber o pid do postmaster. Basta informar o parâmetro m e modo de parada desejado passando como parâmetro a primeira letra de um dos modos: smart, fast e immediate.

```
$   pg_ctl stop -mf
```

Reiniciar o PostgreSQL

Quando alteradas determinadas configurações do PostgreSQL, ou do SO, como as relacionadas a reserva de memória, pode ser necessário reiniciar o PostgreSQL.

O comando restart é de fato um stop seguido de um start, sendo chamado da seguinte forma:

```
$   pg_ctl restart
```

Recarregar os parâmetros de configuração

É possível enviar um sinal para o PostgreSQL indicando que ele deve reler os arquivos de configuração. Muito útil para alterar alguns parâmetros, principalmente de segurança, sem precisar reiniciar o banco. Para tanto faça a seguinte chamada:

```
$   pg_ctl reload
```

Nem todos os parâmetros de configuração podem ser alterados em um reload, existindo alguns parâmetros que demandam um restart do banco.

Verificar se o PostgreSQL está executando

Em alguns momentos é necessário confirmar se o PostgreSQL está rodando. Uma alternativa para isto, bastante comum, é utilizarmos o comando ps para ver se há processos postgres em execução:

```
$   ps -ef f | grep postgres
```

O comando pg_ctl é outra opção, bastando utilizar o parâmetro status conforme demonstrado a seguir:

```
$   pg_ctl status
```

```
postgres@pg01:~$ pg_ctl status
pg_ctl: server is running (PID: 2873)
/usr/local/pgsql9.3.4/bin/postgres "-D" "/db/data"
postgres@pg01:~$
```

Figura 2.5
Status do PostgreSQL.

A vantagem dessa alternativa é que, além de informar se o PostgreSQL está em execução, esse comando mostrará também o PID e a linha de comando completa utilizada para colocar o banco em execução.

Interromper um processo do PostgreSQL

Uma tarefa comum de um administrador de Banco de Dados é matar um processo no banco. Existem diferentes motivos para tanto, como, por exemplo, o processo estar fazendo alguma operação muito onerosa para o horário ou estar demorando para terminar enquanto está bloqueando recursos de que outros processo precisam.

Interromper um backend pode ser feito com *kill*, indicando o processo especificamente em vez de parar o banco inteiro (ao matar o postmaster). O comando *pg_ctl* consegue o mesmo efeito da seguinte forma:

```
$   pg_ctl kill TERM 1520
```

Onde 1520 é, nesse exemplo, o PID do processo que se deseja interromper.

Conexões no PostgreSQL

Para estabelecer uma conexão com o Banco de Dados, no exemplo a seguir vamos utilizar a ferramenta *psql*, um cliente de linha de comando ao mesmo tempo simples e poderoso. Assim, utilizando o usuário postgres, apenas execute:

```
$   psql
```

Acionado dessa forma, sem parâmetros, o psql tentará se conectar ao PostgreSQL da máquina local, na porta padrão 5432 e na base "postgres".

Você pode informar todos esses parâmetros para estabelecer uma conexão com uma máquina remota, em uma base específica e com um usuário determinado. A linha de comando a seguir abre uma conexão:

```
$   psql -h pg02 -p 5432 -d curso -U aluno
```

- **-**h é o servidor (host)
- **-**p a porta

- -d a base (database)
- -U o usuário

Além dos comandos do PostgreSQL, o psql possui uma série de comandos próprios que facilitam tarefas rotineiras. Por exemplo, para listar todas as bases do servidor, basta executar \l:

```
curso=#     \l
```

```
postgres@pg01:~$ psql
psql (9.3.4)
Type “help” for help.
postgres=# \l
                                  List of databases
Name     | Owner    | Encoding | Collate     | Ctype       |   Access privileges
---------+----------+----------+-------------+-------------+-----------------------
 bench   | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 curso   | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =Tc/postgres
 +
         |          |          |             |             | postgres=CTc/postgr
es +
         |          |          |             |             | aluno=Tc/postgres +
         |          |          |             |             | professor=CTc/postg
res
postgres | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
projetox | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =Tc/postgres      +
         |          |          |             |             | postgres=CTc/postgr
es
template0| postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres       +
         |          |          |             |             | postgres=CTc/postgr
es
template1| postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres       +
         |          |          |             |             | postgres=CTc/postgr
es
(6 rows)
postgres=#
```

Figura 2.6
Listando as bases de dados com o psql.

Sempre informe os comandos SQL com ; no final para executá-los:

```
curso=#     SELECT * FROM pg_database;
```

Para executar arquivos de script pelo psql é comum também usar a forma não interativa:

```
$   psql -h pg02 -d curso < /tmp/arquivo.sql
```

Para mudar a base na qual se está conectado, use \c:

```
curso=#     \c projetox;
```

Para sair do psql, use \q:

```
projetox=#   \q
```

Para ver a ajuda sobre os comandos do psql, use \h para comandos SQL do PostgreSQL e \? para os comandos do psql.

O psql será uma ferramenta sempre presente na vida de um administrador PostgreSQL, mesmo que esteja disponível alguma ferramenta gráfica como o pgAdmin. Conhecer o psql pode ser bastante útil.

No Debian/Ubuntu, dependendo de sua instalação, pode ocorrer um erro ao executar o psql indicando que o servidor não foi encontrado. Isto ocorre porque o socket é criado em /tmp mas o cliente está procurando em /var/run/postgresql. Uma possível solução é executar os seguintes passos:

```
$   sudo chown postgres /var/run/postgresql/
```

Edite o *postgresql.conf* e altere o parâmetro unix_socket_directory:

```
unix_socket_directory = '/var/run/postgresql'
```

Reinicie o PostgreSQL e teste novamente o psql.

Quando uma tabela ou visão possui muitas colunas, para que o psql não quebre a linha, é possível habilitar o scroll horizontal. Para isSo, edite o arquivo *~/.bashrc* e adicione o seguinte comando ao final:

```
export "PAGER=less -S"
```

Resumo dos comandos de operação do PostgreSQL

Finalidade	Comando
Iniciar o banco	$ pg_ctl start
Parar o banco	$ pg_ctl stop -mf
Parar o banco	$ pg_ctl restart
Reconfigurar o banco	$ pg_ctl reload
Verificar o status do banco	$ pg_ctl status
Matar um processo	$ pg_ctl kill TERM <pid>
Conectar no banco	$ psql -h pg01 -d curso

Tabela 2.1
Comandos do PostgreSQL.

Atividade de Prática

Configuração do Banco de Dados

- Exemplos de Parâmetros de Configuração:
  - Controle de Recursos de Memória.
  - Controle de Conexões.
  - O quê, quando e como registrar.
  - Custos de Queries.
  - Replicação.
  - Vacuum, Estatísticas.

O PostgreSQL possui diversos parâmetros que podem ser alterados para definir seu comportamento, desde o uso de recursos como memória e controle de conexões até custos de processamento de queries, além de muitos outros aspectos.

Esses parâmetros de configuração podem ser alterados de forma global e permanente no arquivo *postgresql.conf*, refletindo as mudanças em todas as sessões dali para a frente. É possível também fazer ajustes que serão válidos por uma sessão, valendo apenas no escopo desta e até que a respectiva conexão seja encerrada. Outros ajustes podem ser definidos apenas para um usuário ou base específica. Também pode-se definir parâmetros passando-os pela linha de comando que inicia o servidor.

Configuração por sessão

Para definir um parâmetro na sessão, use o comando *SET*. O exemplo a seguir altera o timezone apenas na sessão para Nova York.

```
curso=>      SET timezone = 'America/New_York';
```

```
postgres@pg01:~$ psql -d curso -U aluno
curso=> show timezone;
  TimeZone
-------------
 Brazil/East
(1 row)
curso=> select now();
              now
-------------------------------
 2014-03-31 23:46:31.787769-03
(1 row)
curso=> SET timezone = 'America/New_York';
SET
curso=> select now();
             now
-------------------------------
 2014-03-31 22:46:58.37592-04
(1 row)
curso=> \q
postgres@pg01:~$ psql -d curso -U aluno
curso=> select now();
              now
-------------------------------
 2014-03-31 23:47:16.475588-03
(1 row)
curso=>
```

Figura 2.7 Alteração de parâmetro de sessão.

Veja a figura 2.7, que mostra a alteração do timezone na sessão corrente. Após a desconexão uma nova sessão é iniciada e o valor volta ao original.

Configuração por usuário

Para alterar um parâmetro apenas para um usuário, use o comando *ALTER ROLE ... SET* como no exemplo, que altera o work_mem do usuário jsilva:

```
postgres=#   ALTER ROLE jsilva SET work_mem = ‘16MB’;
```

Configuração por Base de Dados

Para alterar uma configuração para uma base, use *ALTER DATABASE ... SET*. No exemplo a seguir é alterado o mesmo parâmetro work_mem, porém, no escopo de uma base:

```
postgres=#   ALTER DATABASE curso SET work_mem = ’10MB’;
```

Para desfazer uma configuração específica para uma role ou base, use o atributo RESET:

```
postgres=#   ALTER ROLE jsilva RESET work_mem;
postgres=#   ALTER DATABASE curso RESET work_mem;
```

Configurações Globais – postgresql.conf

Na maioria das vezes, desejamos alterar os parâmetros uma única vez valendo para toda a instância. Nesse caso devemos editar o arquivo *postgresql.conf*, que fica na raiz do PGDATA.

```
    $       vi /db/data/postgresql.conf
```

```
#------------------------------------------------------------------------------
# RESOURCE USAGE (except WAL)
#------------------------------------------------------------------------------
# - Memory -
shared_buffers = 96MB                           # min 128kB
                                                                    # (change
requires restart)
#temp_buffers = 8MB                              # min 800kB
#max_prepared_transactions = 0          # zero disables the feature
                                                                    # (change
requires restart)
# Note:  Increasing max_prepared_transactions costs ~600 bytes of shared memory
# per transaction slot, plus lock space (see max_locks_per_transaction).
# It is not advisable to set max_prepared_transactions nonzero unless you
# actively intend to use prepared transactions.
#work_mem = 1MB                                 # min 64kB
#maintenance_work_mem = 16MB     # min 1MB
#max_stack_depth = 2MB                      # min 100kB
# - Disk -
#temp_file_limit = -1                   # limits per-session temp file space
                                                        # in kB, or -1 for no l
imit
```

Figura 2.8
Arquivo de configuração ‘postgresql.conf’.

A maioria dos parâmetros possui comentários com a faixa de valores aceitos e indicação se a alteração exige um restart ou se apenas um reload é suficiente.

Na tabela a seguir, listamos os principais parâmetros que devem ser analisados/ajustados antes de começar a utilizar o PostgreSQL.

Parâmetro	Descrição	Valor
listen_addresses	Por qual interface de rede o servidor aceitará conexões.	O valor default “localhost” permite apenas conexões locais. Alterar para “*” aceitará acessos remotos e em qualquer dos IPs do servidor, mas geralmente só há um.
max_connections	Máximo de conexões aceitas pelo banco.	Depende da finalidade, do número de aplicações e tamanho dos pools de conexões. O valor padrão é 100. Algumas centenas já podem exaurir o ambiente.
statement_timeout	Tempo máximo de execução para um comando. Passado esse valor, o comando será cancelado.	Por padrão é desligado. Esse parâmetro pode ser útil se não se tem controle na aplicação. Porém, alterá-lo para todo o servidor não é recomendado.
shared_buffers	Área da shared memory reservada ao PostgreSQL, é o cache de dados do banco.	Talvez o mais importante parâmetro, não é simples sua atribuição e ao contrário da intuição, aumentá-lo demais pode ser ruim. O valor padrão de 32MB é extremamente baixo. Em um servidor dedicado, inicie com 20% - 25% da RAM.
work_mem	Memória máxima por conexão para operações de ordenação.	O valor padrão de 1MB é muito baixo. Pode-se definir de 4MB - 16MB inicialmente, dependendo da RAM, e analisar se está ocorrendo ordenação em disco. Nesse caso pode ser necessário aumentar. Considerar número de conexões possíveis usando este valor simultaneamente.
Ssl	Habilita conexões SSL.	O padrão é desligado. Se necessário utilizar, é preciso compilar o PostgreSQL com suporte e instalar o openssl.
superuser_reserved_connections	Número de conexões, dentro de max_connections, reservada para o superusuário postgres.	Se houver diversos administradores ou ferramentas de monitoramento que executam com o usuário postgres, pode ser útil aumentar esse parâmetro.
effective_cache_size	Estimativa do Otimizador sobre o tamanho do cache, é usada para decisões de escolha de planos de execução de queries.	Definir como o tamanho do shared_ buffer mais o tamanho do cache do SO. Inicialmente pode-se usar algo entre 50% e 75% da RAM.
logging_collector	Habilita a coleta de logs, interceptando a saída para stderr e enviando para arquivos.	Por padrão é desligado, enviando as mensagens para a saída de erro (stderr). Altere para ON para gerar os arquivos de log no diretório padrão pg_log.
bytea_output	Formato de retorno de dados de campos do tipo bytea.	Importante se houve migração de versão. Na versão 9 o padrão passou a ser HEX, porém antes era Escape. Se estiver migrando bases da versão 8 é necessário manter como escape.
Datestyle	Formato de exibição de datas.	Definir como ‘iso, dmy’ para interpretar datas no formato dia/mes/ano.
lc_messages	Idioma das mensagens de erro.	Traduções de mensagens de erro no lado servidor podem causar ambiguidades ou erros de interpretação e dificultam a busca de soluções, pois a grande maioria das informações reportadas estão em inglês. Sugestão: en_US.UTF-8

Parâmetro	Descrição	Valor
lc_monetary	Formato de moeda.	pt_BR.UTF-8
lc_numeric	Formato numérico.	pt_BR.UTF-8
lc_time	Formato de hora.	pt_BR.UTF-8

Tabela 2.2
Parâmetros básicos de configuração do PostgreSQL.

A lista apresenta parâmetros gerais. Vamos voltar a falar de parâmetros nas próximas sessões, por exemplo, quando analisarmos os processos de Monitoramento e Manutenção do Banco de Dados.

Consultando as configurações atuais

Para consultar os valores atuais de parâmetros de configuração, podemos usar o comando *SHOW*.

```
curso=#     SHOW ALL;
```

O comando acima mostra todos os parâmetros. É possível consultar um parâmetro específico:

```
curso=#     SHOW max_connections;
```

Considerações sobre configurações do Sistema Operacional

Além das configurações do PostgreSQL, podem ser necessários ajustes nas configurações do Sistema Operacional, principalmente relacionados a shared memory e semáforos. Dependendo do volume de uso do banco, precisaremos aumentar esses parâmetros do kernel.

Shared Memory

O mais importante é o SHMMAX, que define o tamanho máximo de um segmento da shared memory. Para consultar o valor atual execute:

```
$   sysctl kernel.shmmax
```

Dependendo do valor de shared_buffer e outros parâmetros de memória do PostgreSQL, o valor padrão provavelmente é baixo e deve ser aumentado. Sempre que alterar o tamanho do shared buffer, verifique o shmmax, que deve sempre ser maior que shared_buffers.

Para isso, edite o *arquivo /etc/sysctl.conf* e adicione, ou altere, a entrada para shmmax.
Por exemplo, para definir o máximo como 8GB:

```
kernel.shmmax = 8589934592
```

```
$ sysctl kernel.shmmax
kernel.shmmax = 183554432
$
$ /sbin/sysctl kernel.sem
kernel.sem = 250        32000   32      128
$
$ ulimit -n -u
open files                      (-n) 1024
max user processes              (-u) 3850
$
```

Figura 2.9
Configurações do Sistema Operacional.

Semáforos

Outro recurso do Sistema Operacional que talvez precise ser ajustado em função da quantidade de processos diz respeito aos semáforos do sistema.

Para consultar os valores atuais dos parâmetros de semáforos, faça uma consulta a kernel.sem:

```
$ sysctl kernel.sem
kernel.sem = 250        32000   32      128
```

Esses quatro valores são, em ordem: SEMMSL, SEMMNS, SEMOPM e SEMMNI.

Os relevantes para as configurações do PostgreSQL são:

- **SEMMNI**: número de conjuntos de semáforos;
- **SEMMNS**: número total de semáforos.

Esses parâmetros podem precisar ser ajustados para valores grandes, levando em consideração principalmente o max_connections.

Pode-se alterá-los no mesmo arquivo */etc/sysctl.conf*. Exemplo aumentando SEMMNI para 256:

```
kernel.sem = 250        32000   32      256
```

Será necessário reiniciar a máquina para que os valores alterados no arquivo */etc/sysctl.conf* tenham efeito.

Para saber qual os valores mínimos exigidos pelo PostgreSQL, pode-se fazer o seguinte cálculo:

```
SEMMNI      (max_connections + autovacuum_max_workers + 4) / 16
SEMMSN      ((max_connections + autovacuum_max_workers + 4) / 16) * 17
```

Limites

Outras configurações importantes estão relacionadas aos limites de recursos por usuário. Dependendo do número de conexões é necessário definir os parâmetros *max user processes* e *open files*. Para consultar o valor atual, conectado com o usuário postgres, execute:

```
$   ulimit -n -u
```

O número máximo de processos deve ser maior que max_connections.
Para alterar estes valores edite o arquivo */etc/security/limits.conf*:

```
$   sudo vi /etc/security/limits.conf
```

Por exemplo, para aumentar o número máximo de arquivos abertos e de processos pelo usuário postgres, insira as linhas no arquivo como no exemplo abaixo:

```
postgres soft nofile 8000
postgres hard nofile 32000
postgres soft nproc 5000
postgres hard nproc 10000
```

soft indica um limite no qual o próprio processo do usuário pode alterar o valor até no máximo o definido pelo limite hard.

Para mais informações sobre configurações do Sistema Operacional, acesse: http://www.postgresql.org/docs/9.3/static/kernel-resources.html

3

Organização lógica e física dos dados

objetivos

Conhecer a organização do PostgreSQL do ponto de vista lógico (bases de dados e schemas) e físico (área de dados e tablespaces) Aprender a estrutura de diretórios e arquivos, além da função de cada item; Entender a organização da instância em bases, schemas e objetos, e ainda os metadados do banco no catálogo do sistema.

conceitos

PGDATA; Catálogo; Instância; Schemas; Tablespaces; TOAST e metadados.

Estrutura de diretórios e arquivos do PostgreSQL

O PostgreSQL armazena e organiza os dados e informações de controle por padrão sob o pgdata, a área de dados.

Tanto as bases de dados podem ser armazenadas em outros locais através do uso de tablespaces quanto o WAL pode ser armazenado fora do PGDATA com o uso de links simbólicos; porém, as referências a eles continuarão sob essa área.

Os arquivos de log de erros e os arquivos de configuração de segurança podem ser armazenados fora do PGDATA por configurações no *postgresql.conf*.

Mesmo que se tenha uma instalação com os dados, logs de transação, logs de erros, distribuídos em locais diferentes, o PGDATA é o coração do PostgreSQL e entender sua estrutura ajuda muito, sendo essencial para a sua administração.

Em nosso exemplo o PGDATA está em "/db/data" e sua estrutura geral pode ser vista na figura 3.1.

```
postgres@pg01:~$ tree -L 2 -I lost+found /db/
/db/
├── data
│   ├── base
│   ├── global
│   ├── pg_clog
│   ├── pg_log
│   ├── pg_multixact
│   ├── pg_notify
│   ├── pg_serial
│   ├── pg_snapshots
│   ├── pg_stat_tmp
│   ├── pg_subtrans
│   ├── pg_tblspc
│   ├── pg_twophase
│   ├── pg_xlog
│   ├── PG_VERSION
│   ├── postgresql.conf
│   ├── pg_hba.conf
│   ├── pg_ident.conf
│   ├── postmaster.opts
│   ├── postmaster.pid
│   └── serverlog
└── ...
15 directories, 7 files
postgres@pg01:~$
```

Figura 3.1
Layout de arquivos do PostgreSQL.

Arquivos

Há três arquivos de configuração:

- *postgresql.conf*: já vimos na sessão anterior. É o arquivo principal de configuração do banco.
- *pg_hba.conf*: usado para controle de autenticação, e que será abordado com mais detalhes adiante, no tópico sobre segurança.
- *pg_ident.conf*: usado para mapear usuários do SO para usuários do banco em determinados métodos de autenticação.

Além dos arquivos de configuração, existem outros arquivos com informações de controle utilizadas por utilitários como o pg_ctl.

São eles:

- **postmaster.pid:** é um arquivo lock para impedir a execução do PostgreSQL duplicado, contendo o PID do processo principal em execução e outras informações, tais como a hora em que o serviço foi iniciado;
- **postmaster.opts:** contém a linha de comando, com todos os parâmetros, usada para iniciar o PostgreSQL e que é usada pelo pg_ctl para fazer o restart;
- **PG_VERSION**: contém apenas a versão do PostgreSQL.

Pode existir também algum arquivo de log de erros, como serverlog, criado antes de se alterar o parâmetro logging_collector para ON e direcionar as logs para outro diretório.

Diretórios

Veremos em detalhes cada um dos seguintes diretórios:

- base.
- global.
- pg_xlog.
- pg_log.
- pg_tblspc.
- diretórios de controle de transação.

O diretório base é onde, por padrão, estão os arquivos de dados. Dentro dele existe um subdiretório para cada base.

```
postgres@pg01:~$ tree -d /db/data/base/
/db/data/base/
├── 1
├── 12035
├── 16384
├── 16385
├── 49503
└── 58970
6 directories
postgres@pg01:~$
```

Figura 3.2
Estrutura do diretório PGDATA/base.

O nome dos subdiretórios das bases de dados é o OID da base, que pode ser obtido consultando a tabela do catálogo pg_database com o seguinte comando:

```
postgres=#   SELECT oid, datname FROM pg_database;
```

```
postgres=# SELECT oid, datname FROM pg_database;
  oid  |  datname
-------+-----------
     1 | template1
 12035 | template0
 16384 | curso
 16385 | projetox
 49503 | postgres
 58970 | bench
(6 rows)

postgres=#
```

Figura 3.3
OIDs das bases de dados.

Dentro do diretório de cada base estão os arquivos das tabelas e índices. Cada tabela ou índice possui um ou mais arquivos. Uma tabela terá inicialmente um arquivo, cujo nome é o atributo filenode que pode ser obtido nas tabelas de catálogo. O tamanho máximo do arquivo é 1GB. Ao alcançar este limite serão criados mais arquivos, cada um com o nome filenode.N, onde N é um incremental.

Para descobrir o nome dos arquivos das tabelas consulte o filenode com o seguinte comando:

```
curso=>     SELECT relfilenode FROM pg_class WHERE relname='grupos';
```

Esse comando exibe o filenode para uma tabela específica. A consulta exibida na figura a seguir lista o filenode para todas as tabelas da base.

```
curso=# SELECT relname, relfilenode FROM pg_class WHERE relkind='r' AND relname
not like 'pg%' and relname not like 'sql%';
   relname    | relfilenode
--------------+-------------
 jogos        |       24593
 times        |       41077
 grupos_times |       24588
 grupos       |       24584
 cidades      |       24599
 contas       |       41230
(6 rows)
curso=#
```

Figura 3.4
Tabelas e Filenodes.

Outra forma de obter o nome do arquivo de dados das tabelas é a função pg_relation_filepath(), que retorna o caminho do primeiro arquivo da tabela:

```
curso=>     SELECT pg_relation_filepath(oid)
FROM pg_class WHERE relname='grupos';
```

Além dos arquivos de dados, existem arquivos com os seguintes sufixos:

- **_fsm**: para o Free Space Map, indicando onde há espaço livre nas páginas das tabelas;
- **_vm**: para o Visibility Map, que indica as páginas que não precisam passar por vacuum;
- **_init:** para unlogged tables;
- **arquivos temporários**: cujo nome tem o formato tNNN_filenode, onde NNN é o PID do processo backend que está usando o arquivo.

O diretório global contém os dados das tabelas que valem para toda a instância e são visíveis de qualquer base. São tabelas do catálogo de dados como, por exemplo, pg_databases.

O pg_xlog contém as logs de transação do banco e os arquivos de WAL, que são arquivos contendo os registros das transações efetuadas. Eles possuem as seguintes características:

- Cada arquivo tem 16MB;
- O nome é uma sequência numérica hexadecimal;
- Após checkpoints e arquivamento, os arquivos são reciclados.

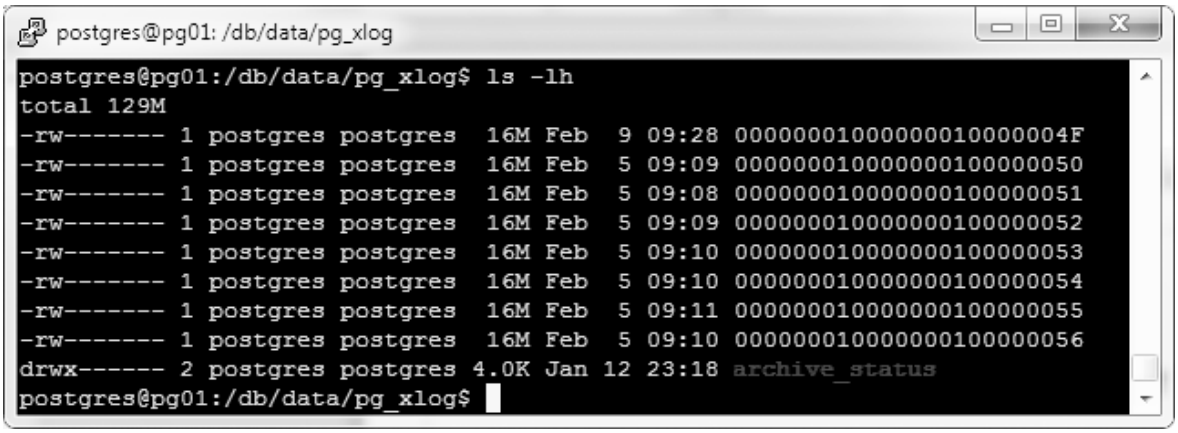


Figura 3.5
O diretório 'archive_ status' contém informações de controle sobre quais arquivos já foram arquivados.

Pode existir ainda o diretório "pg_log", dependendo de suas configurações, que contém os logs de erro e atividade. Se foi habilitada a coleta de log com o parâmetro logging_collector, o diretório padrão será esse. Os nomes dos arquivos dependem também das configurações escolhidas. Na sessão sobre monitoramento, os aquivos de log voltarão a ser tratados.

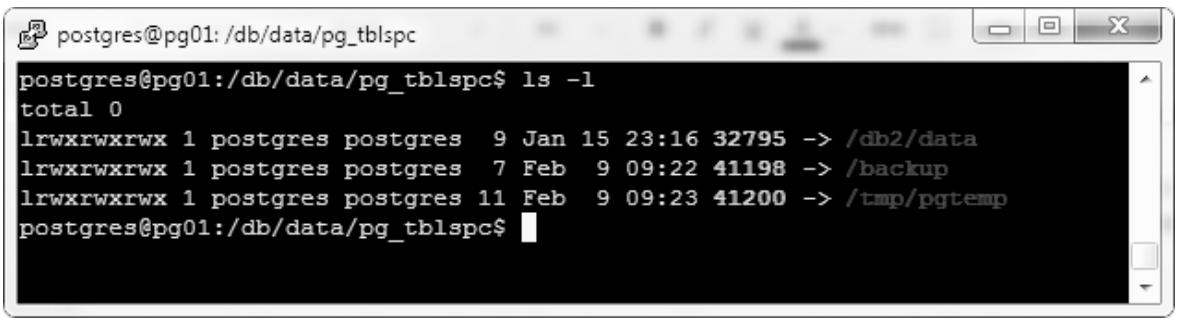


Figura 3.6
O pg_tblspc contém links simbólicos para as áreas reais dos tablespaces.

Por fim, temos um conjunto de diretórios que contém arquivos de controle de status de transações diversas:

- pgdata/pg_clog;
- pgdata/pg_serial;
- pgdata/pg_multixact;
- pgdata/pg_subtrans;
- pgdata/pg_twophase.

Organização geral

Um servidor ou instância do PostgreSQL pode conter diversas bases de dados. Essas bases, por sua vez, podem conter Schemas que conterão objetos como tabelas e funções. Pode-se dizer que esta é, de certa forma, uma divisão lógica.

Por outro lado, tanto bases inteiras, ou uma tabela ou índice, podem estar armazenados em um tablespace. Logo, não podemos colocá-lo na mesma hierarquia. O esquema a seguir demonstra essa organização básica:

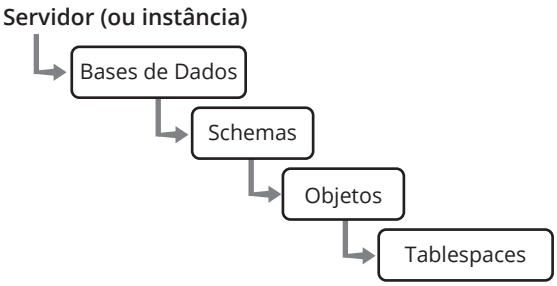


Figura 3.7
Estrutura do PostgreSQL.

Bases de dados

Uma base de dados é uma coleção de objetos como tabelas, visões e funções.
No PostgreSQL, assim como na maioria dos SGBDs, observa-se que:

- Um servidor pode ter diversas bases.
- Toda conexão é feita em uma base.
- Não se pode acessar objetos de outra base.
- Bases são fisicamente separadas.
- Toda base possui um owner com controle total nela.

Quando uma instância é iniciada, com o initdb, três bases são criadas:

- **template0**: usado para recuperação pelo próprio Postgres, não é alterado;
- **template1**: por padrão, serve de modelo para novos bancos criados. Se for necessário ter um modelo corporativo de base, conterá as extensões e funções pré-definidas;
- **postgres**: base criada para conectar-se por padrão, não sendo necessária para o funcionamento do PostgreSQL (pode ser necessária para algumas ferramentas).

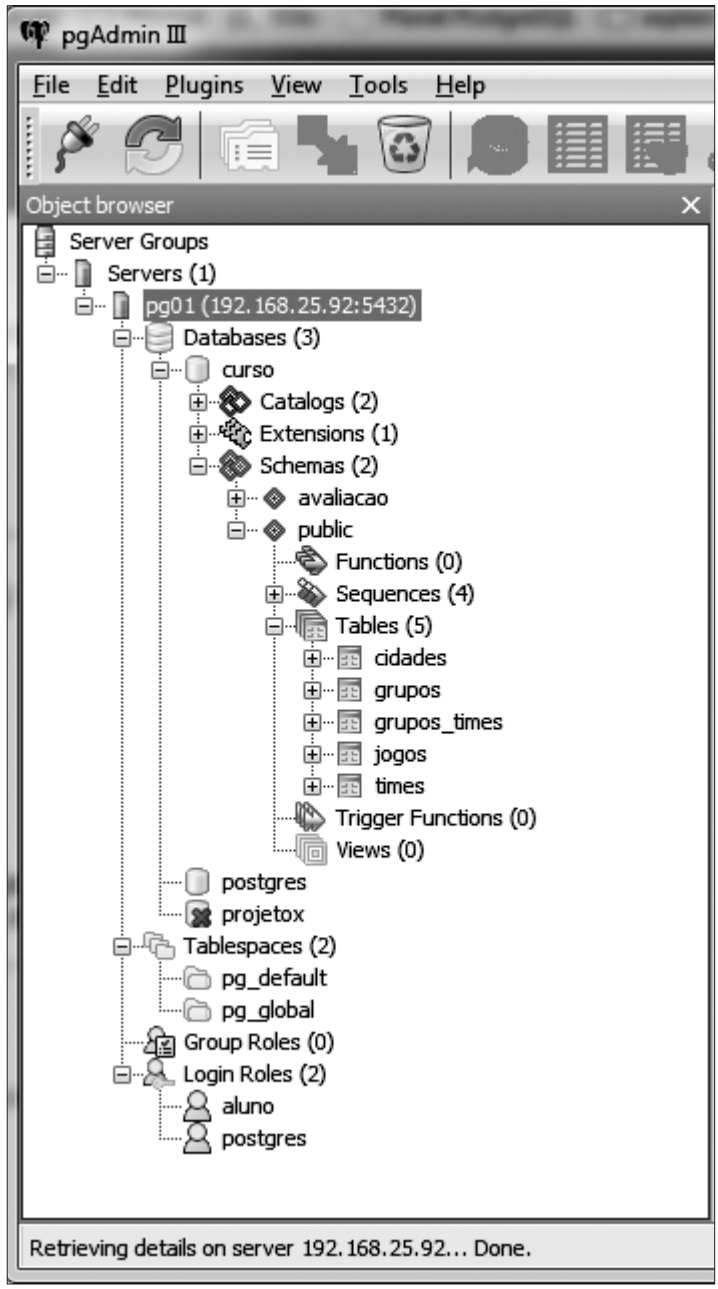


Figura 3.8
Ferramenta gráfica pgAdminIII, ilustrando a árvore de hierárquia no PostgreSQL.

Para listar as bases de dados existentes no servidor, podemos consultar a tabela pg_database do catálogo do sistema, ou usar o comando \l do psql. Muito útil, \l+ exibe também o tamanho da base e o tablespace padrão.

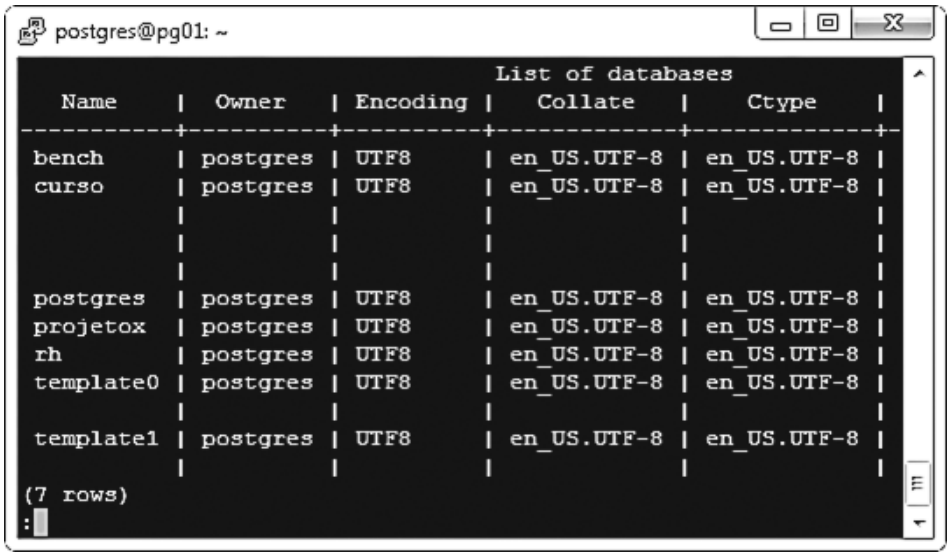


Figura 3.9
No psql, use \l para listar as bases existentes.

Toda base possui um dono que, caso não seja informado no momento da criação será o usuário que está executando o comando. Apenas superusuários ou quem possui a role CREATEDB podem criar novas bases.

Criação de bases de dados

Para criar uma nova base, conecte-se no PostgreSQL (pode ser na base postgres ou outra qualquer) e execute o comando:

```
postgres=#   CREATE DATABASE curso;
```

As principais opções no momento da criação de uma base são:

- **OWNER**: o usuário dono da base pode criar schemas e objetos e dar permissões;
- **TEMPLATE**: a base modelo a partir da qual a nova base será copiada;
- **ENCODING**: que define o conjunto de caracteres a ser utilizado;
- **TABLESPACE**: que define o tablespace padrão onde serão criados os objetos na base.

```
postgres=#   CREATE DATABASE rh
                                    OWNER postgres
                                    ENCODING 'UTF-8'
                                    TABLESPACE = tbs_disco2;
```

Esse exemplo criará a base rh com as opções definidas e tendo como modelo default o template1.

Há um utilitário para linha de comando que pode ser usado para a criação de bases de dados chamado createdb. Ele tem as mesmas opções do comando SQL:

```
$   createdb curso -O aluno -T template_novo
```

Exclusão de bases de dados

Além da role superusuário, apenas o dono da base poderá excluí-la. A exclusão de uma base não pode ser desfeita e não é possível remover uma base com conexões.

Assim como na criação da base, é possível executar a exclusão por comando *SQL*:

```
postgres=#   DROP DATABASE curso;
```

Ou pelo utilitário:

```
$   dropdb curso;
```

createdb e dropdb, como a maioria dos utilitários do PostgreSQL, podem ser usados remotamente. Assim, suportam as opções de conexão -h host e -U usuário.

Schemas

Como já foi dito anteriormente, bases podem ser organizadas em schemas. Schemas são apenas uma divisão lógica para os objetos do banco, sendo permitido acesso cruzado entre objetos de diferentes schemas. Por exemplo, supondo um schema chamado vendas e um schema chamado estoque, a seguinte query pode ser realizada:

```
curso=>      SELECT *
             FROM vendas.pedido as pe
             INNER JOIN estoque.produtos as pr ON pe.idProduto = pr.idProduto;
```

Também é importante frisar que podem existir objetos com o mesmo nome em schemas diferentes. É completamente possível existir uma tabela produto no schema vendas e uma no schema estoque. Para referenciá-las, sem ambiguidade, deve-se usar sempre o nome completo do objeto (vendas.produto e estoque.produto).

Quando referenciamos ou criamos um objeto sem informar o nome completo, costuma-se entender que ele está ou será criado no schema public. O funcionamento correto é um pouco mais complexo e depende do parâmetro search_path.

O public é um schema que por padrão existe no modelo padrão template1 e geralmente existe em todas as bases criadas posteriormente (seja porque não se alterou a template1, seja por não usar um outro modelo). Neste schema public, com as permissões originais, qualquer usuário pode acessar e criar objetos. Não é obrigatória a existência do public; ele pode ser removido.

O search_path é um parâmetro que define justamente a ordem e quais schemas serão varridos quando um objeto for referenciado sem o nome completo. Por padrão, o search_path é definido:

```
curso=#       SHOW search_path;
              search_path
              ------------------
              "$user",public
```

"$user" significa o próprio usuário conectado. Assim, supondo que estejamos conectados com o usuário aluno e executarmos o comando:

```
curso=#      SELECT * FROM grupos;
```

O PostgreSQL procurará uma tabela, ou visão, chamada grupos em um esquema chamado aluno. Se não encontrar, procurará no public e, se também não encontrar nada ali, um erro será gerado.

A mesma ideia vale para a criação de objetos. Se executarmos o seguinte comando:

```
curso=#      CREATE TABLE pais (id int, nome varchar(50), codInternet char(2));
```

Se existir um esquema chamado aluno, essa tabela será criada lá. Caso contrário, será criada no public.

Dito isso, é fácil notar como podem acontecer confusões. Portanto, sempre referencie e exija dos desenvolvedores que usem o nome completo dos objetos.

Uma prática comum é criar um schema para todos os objetos da aplicação (por exemplo, vendas) e definir a variável search_path para esse schema. Desse modo, nunca seria necessário informar o nome completo, apenas o nome do objeto:

```
curso=#     SET search_path = vendas;
curso=#     CREATE TABLE pedidos (idcliente int, idproduto int, data timestamp);
curso=#     SELECT * FROM pedidos;
```

O parâmetro search_path pode ser definido para uma sessão apenas, para um usuário ou para uma base.

Para listar todos os schemas existentes na base atual, no psql pode-se usar o comando *\dn+*. Uma alternativa é consultar a tabela pg_namespace do catálogo de sistema.

Ao modelar o Banco de Dados, uma dúvida comum é sobre como fazer a distribuição das bases de dados, utilizando ou não diferentes schemas. Do ponto de vista de arquitetura de bancos de dados, essa decisão deve ser tomada se há relação entre os dados e se será necessário acessar objetos de diferentes aplicações no nível de SQL. Se sim, então deve-se usar schemas em uma mesma base, caso contrário, mais de uma base de dados.

Criação de schema

Para criar um schema, basta estar conectado na base e usar o comando:

```
curso=#     CREATE SCHEMA auditoria;
```

Somente quem possui a permissão CREATE na base de dados e superusuários podem criar schemas. O dono da base recebe a permissão CREATE.

Para definir o dono de um schema, atribui-se a propriedade AUTHORIZATION:

```
curso=#     CREATE SCHEMA auditoria AUTHORIZATION aluno;
```

Diferente do padrão SQL, no PostgreSQL pode-se atribuir donos de objetos diferentes do dono do schema ao qual eles pertencem.

Exclusão de Schema

Só é possível remover um schema se este estiver vazio. Alternativamente, pode-se forçar a remoção com a cláusula CASCADE:

```
curso=#     DROP SCHEMA auditoria CASCADE;
```

Schemas pg_toast e pg_temp

Schemas criados pelo próprio PostgreSQL:

- pg_toast_N
- pg_temp_N
- pg_toast_temp_N

Eventualmente poderão ser vistos schemas na base de dados que não foram criados explicitamente. Esses schemas poderão ter nomes como pg_toast, pg_temp_N e pg_toast_temp_N, onde N é um número inteiro.

pg_temp identifica schemas utilizados para armazenar tabelas temporárias e seu conteúdo pode ser ignorado.

pg_toast e pg_toast_temp são utilizados para armazenar tabelas que fazem uso do TOAST, explicado a seguir.

TOAST

Por padrão, o PostgreSQL trabalha com páginas de dados de 8kB e não permite que um registro seja maior do que uma página. Quando, por exemplo, um registro possui um campo de algum tipo texto que recebe um valor maior do que 8kB, internamente é criada uma tabela chamada **TOAST,** que criará registros "auxiliares" de tal forma que o conteúdo do campo original possa ser armazenado.

TOAST
abreviatura de The Oversized-Attribute Storage Technique, um recurso do PostgreSQL para tratar campos grandes.

Esse processo é completamente transparente para os usuários do Banco de Dados. Quando inserimos ou consultamos um campo desse tipo, basta referenciar a tabela principal, sem sequer tomar conhecimento das tabelas TOAST.

Vale mencionar que o PostgreSQL busca sempre compactar os dados armazenados em tabelas TOAST.

Tablespaces

Tablespaces:

- locais no sistema de arquivos para armazenar dados
- é um diretório
- permite utilizar outros discos para:
  - expandir o espaço atual (disco cheio)
  - questões de desempenho

Tablespaces são locais no sistema de arquivos onde o PostgreSQL pode armazenar os arquivos de dados. Basicamente, tablespace é um diretório.

A ideia de um tablespace é fornecer a possibilidade de utilizar outros discos para armazenar os dados do banco, seja por necessidade de expandir o espaço atual, caso um disco em uso esteja cheio, seja por questões de desempenho.

Do ponto de vista de desempenho, usar tablespaces permite dividir a carga entre mais discos, caso estejamos enfrentando problemas de I/O, movendo uma base para outro disco, ou mesmo um conjunto de tabelas e índices. Pode-se criar um tablespace em um disco mais rápido e utilizá-lo para um índice que seja extremamente utilizado, enquanto outro disco mais lento e mais barato pode ser usado para armazenar dados históricos pouco acessados. Pode-se também usar tablespaces diferentes para apontar para discos com RAIDs diferentes.

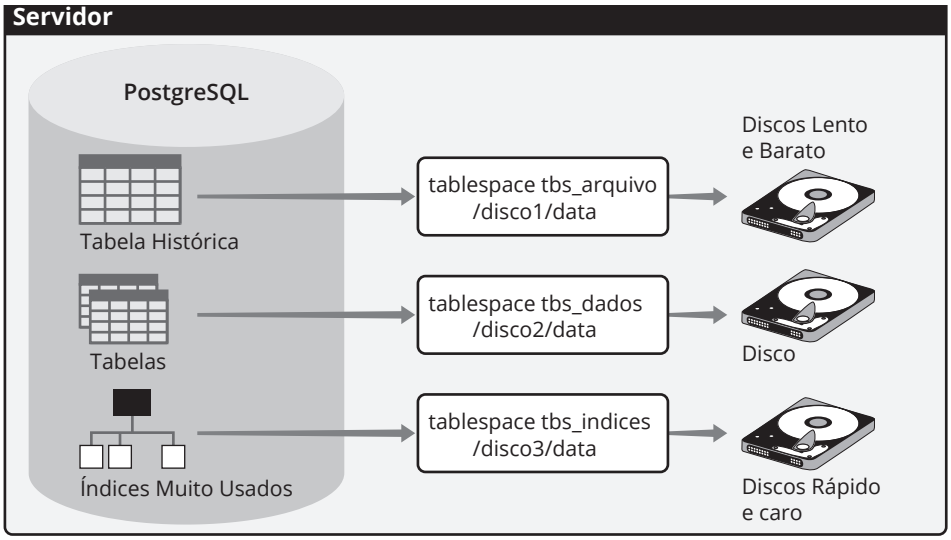


Figura 3.10
Exemplo de uso de tablespaces em discos de diferentes tipos.

Para listar os tablespaces existentes no servidor, no psql use o comando *\db* ou consulte a tabela do catálogo pg_tablespace.

Por padrão, existem dois tablespaces pré-definidos:

- **pg_default:** aponta para o diretório PGDATA/base que já estudamos, e, caso não se altere a base template1, será o template default de todas as demais bases;
- **pg_global:** aponta para o diretório PGDATA/global, que contém os objetos que são compartilhados entre todas as bases, como a tabela pg_database.

Criação e uso de tablespaces

Para criar um tablespace, o diretório deve existir previamente, estar vazio e ter como dono o usuário postgres. Somente superusuários podem criar tablespaces. Para criar um novo tablespace, use o comando *SQL*:

```
postgres=#   CREATE TABLESPACE tbs_arquivo LOCATION '/disco1/data';
```

Depois de criado um tablespace, pode-se colocar objetos nele. Veja os exemplos:

```
curso=#      CREATE TABLE registro (login varchar(20), datahora timestamp)
             TABLESPACE tbs_arquivo;
curso=#      CREATE INDEX idx_data ON compras(data) TABLESPACE tbs_indices;
```

Pode-se criar uma base de dados e definir seu tablespace, onde serão criados os objetos do catálogo de sistema da base e também será o tablespace padrão dos objetos a serem criados caso não seja especificado diferente. Por exemplo:

```
postgres=#   CREATE DATABASE vendas TABLESPACE tbs_dados;
```

No exemplo, o catálogo da base vendas foi criado no tablespace tbs_dados e, agora, se for criada uma tabela sem informar um tablespace, ela será também criada no tbs_dados e não mais no pg_default.

É possível alterar o tablespace de uma tabela ou índice, resultando em uma operação em que os dados serão movidos para o novo local. Também podemos alterar o tablespace padrão de uma base, o catálogo e todos os objetos que foram criados sem informar o tablespace diferente serão movidos também. Durante a alteração, os objetos ficam bloqueados mesmo para consultas.

O seguinte exemplo move o índice para uma nova localização:

```
curso=#      ALTER INDEX idx_data SET TABLESPACE tbs_dados;
```

É possível também definir os parâmetros de custo, seq_page_cost e random_page_cost, usados pelo Otimizador de queries, em um tablespace específico. Esses parâmetros serão explicados na sessão que trata as questões de desempenho.

Exclusão de tablespaces

Para excluir um tablespace, ele deve estar vazio. Todos os objetos contidos nele devem ser removidos antes. Para remover:

```
postgres=#   DROP TABLESPACE tbs_arquivo;
```

Tablespace para tabelas temporárias

Quando são criadas tabelas temporárias ou uma query precisa ordenar grande quantidade de dados, essas operações armazenam os dados de acordo com a configuração temp_tablespaces. Esse parâmetro é uma lista de tablespaces que serão usados para manipulação de dados temporários. Se houver mais de um tablespace temporário definido, o PostgreSQL seleciona aleatoriamente um deles a cada vez.

Esse parâmetro, por padrão, é vazio e, nese caso, é usado o tablespace padrão da base para criação dos dados temporários.

Catálogo de Sistema do PostgreSQL

Algumas das principais tabelas e views do catálogo:

- pg_database.
- pg_namespace.
- pg_class.
- pg_proc.
- pg_roles.
- pg_view.
- pg_indexes.
- pg_stats.
- Views estatísticas.

O Catálogo de Sistema ou Catálogo do PostgreSQL, e às vezes também chamado de dicionário de dados, contém as informações das bases e objetos criados no banco. O PostgreSQL armazena informações sobre todos os seus objetos, como tabelas, em outras tabelas, por isso mesmo chamados de objetos “internos” e constituindo um meta modelo.

Existem dezenas de tabelas e views no catálogo, como por exemplo a pg_role e pg_attribute, contendo respectivamente as informações de roles do servidor e as informações de todas as colunas de todas as tabelas da base de dados. O catálogo é uma rica fonte de informação e controle do SGBD.

Não altere dados direto no catálogo: use os comandos SQL para criar e alterar objetos.

pg_database

Contém informações das bases de dados do servidor. Ela é global, existe uma para toda instância.

datname	Nome da base de dados.
dattablespace	Tablespace default da base de dados.
datacl	Privilégios da base de dados.

pg_namespace

Contém informações dos schemas da base atual.

nspname	Nome do schema.
nspowner	ID do dono do schema.
nspacl	Privilégios do schema.

pg_class

A pg_class talvez seja a mais importante tabela do catálogo. Ela contém informações de tabelas, views, sequences, índices e toast tables. Esses objetos são genericamente chamados de relações.

relname	Nome da tabela ou visão ou índice.
relnamespace	ID do schema da tabela.
relowner	ID do dono da tabela.
reltablespace	ID do tablespace que a tabela está armazenada. 0 se default da base.
reltuples	Estimativa do número de registros.
relhasindex	Indica se a tabela possui índices.
relkind	Tipo do objeto: r = tabela, i = índice, S = sequência, v = visão, c = tipo composto, t = tabela TOAST, f = foreign table.
relacl	Privilégios da tabela.

pg_proc

Contém informações das funções da base atual.

proname	Nome da função.
prolang	ID da linguagem da função.
pronargs	Número de argumentos.
prorettype	Tipo de retorno da função.
proargnames	Array com nome dos argumentos.
prosrc	Código da função, ou arquivo de biblioteca do SO etc.
proacl	Privilégios da função.

pg_roles

Esta é uma visão que usa a tabela pg_authid. Contém informações das roles de usuários e grupos. Os dados de roles são globais à instância.

rolname	Nome do usuário ou grupo.
rolsuper	Se é um superusuário.
rolcreaterole	Se pode criar outras roles.
rolcreatedb	Se pode criar bases de dados.
rolcanlogin	Se pode conectar-se.
rolvaliduntil	Data de expiração.

pg_view

Contém informações das visões da base atual.

viewname	Nome da visão.
schemaname	Nome do schema da visão.
definition	Código da visão.

pg_indexes

Contém informações dos índices da base atual.

schemaname	Schema da tabela e do índice.
tablename	Nome da tabela cujo índice pertence.
indexname	Nome do índice.
definition	Código do índice.

pg_stats

Contém informações mais legíveis das estatísticas dos dados da base atual.

schemaname	Nome do schema da tabela cuja coluna pertence.
tablename	Nome da tabela cuja coluna pertence.
Attname	Nome da coluna.
null_frac	Percentual de valores nulos.
avg_width	Tamanho médio dos dados da coluna, em bytes.
n_distinct	Estimativa de valores distintos na coluna.
most_common_vals	Valores mais comuns na coluna.
Correlation	Indica um percentual de ordenação física dos dados em relação a ordem lógica.

Visões estatísticas

O catálogo do PostgreSQL contém diversas visões estatísticas que fornecem informações que nos ajudam no monitoramento do que está acontecendo no banco. Essas visões contêm dados acumulados sobre acessos a bases e objetos.

Dentre essas visões, destacamos:

- pg_stat_activity;
- pg_locks;
- pg_stat_database;
- pg_stat_user_tables.

pg_stat_database

Essa visão mantém informações estatísticas das bases de dados. Os números são acumulados desde o último reset de estatísticas.

As colunas xact_commit e xact_rollback somadas fornecem o número de transações ocorridas no banco. Com o número de blocos lidos e o número de blocos encontrados, podemos calcular o percentual de acerto no cache do PostgreSQL.

As colunas temp_files e deadlock devem ser acompanhadas, já que números altos podem indicar problemas.

Datname	Nome da base de dados.
xact_commit	Número de transações confirmadas.
xact_rollback	Número de transações desfeitas.
blks_read	Número de blocos lidos do disco.
blks_hit	Número de blocos encontrados no Shared Buffer cache.
temp_files	Número de arquivos temporários.
Deadlock	Número de deadlocks.

pg_stat_user_tables

Essa visão contém estatísticas de acesso para as tabelas da base atual, exceto do catálogo.

Schemaname	Nome do schema da tabela.
Relname	Nome da tabela.
seq_scan	Número de seqscans (varredura sequencial) ocorridos na tabela.
idx_scan	Número de idxscans (varredura de índices) ocorridos nos índices da tabela.
n_live_tup	Número estimado de registros.
n_dead_tup	Número estimado de registros mortos (excluídos ou atualizados mas ainda não removidos fisicamente).
last_vacuum / last_autovacuum	Hora da última execução de um vacuum / auto vacum.
last_analyze / last_autoanalyze	Hora da última execução de um analyze / auto analyze.

Existem diversas outras visões com informações estatísticas sobre índices, funções, sequences, dados de I/O e mais. Para conhecer todas, acesse o link indicado no AVA.

Na sessão sobre Monitoramento, serão vistos exemplos de uso das tabelas e visões do Catálogo.

4

Administrando usuários e segurança

objetivos

Aprender sobre os recursos de segurança do PostgreSQL; Entender o conceito e gerenciamento de Roles; Conhecer os principais tipos de Privilégios, como fornecê-los e revogá-los, além do controle de Autenticação.

conceitos

Roles de Usuários e de Grupos; Privilégios e Host Based Authentication.

Gerenciando roles

Controle de permissões por roles:

- "Usuário" e "Grupo".
- Roles são globais ao servidor.
- Primeiro superusuário criado no initdb.
- Não há relação entre usuário do SO e banco.

O PostgreSQL controla permissões de acesso através de roles. Estritamente falando, no PostgreSQL não existem usuários ou grupos, apenas roles. Porém, roles podem ser entendidas como usuários ou grupos de usuários dependendo de como são criadas. Inclusive, o PostgreSQL aceita diversos comandos com sintaxe USER e GROUP para facilitar a manipulação de roles (e para manter a compatibilidade com versões anteriores).

Roles existem no SGBD e a princípio não possuem relação com usuários do Sistema Operacional. Porém, para alguns utilitários clientes, se não informado o usuário na linha de comando, ele assumirá o usuário do Sistema Operacional (ou a variável PGUSER, se existir) – como por exemplo, no psql.

As roles são do escopo do servidor: não existe uma role de uma base de dados específica, ela vale para a instância. Por outro lado, uma role pode ser criada no servidor e não possuir acesso em nenhuma base de dados.

Quando a instância é inicializada com o initdb, é criada a role do superusuário com o mesmo nome do usuário do Sistema Operacional, geralmente chamado postgres.

Criação de roles

Para criar uma role, usa-se o comando *CREATE ROLE*. Esse comando possui diversas opções. As principais serão apresentadas a seguir.

Uma prática comum é a criação de um usuário específico que será utilizado para conectar-se ao banco por um sistema ou aplicação. Exemplo:

```
postgres=#   CREATE ROLE siscontabil LOGIN PASSWORD 'a1b2c3';
```

Nesse comando, a opção LOGIN informa que a role pode conectar-se e define sua senha. Esse formato de opções é basicamente o que entende-se como um usuário.

Outro exemplo de criação de role para usuários:

```
postgres=#   CREATE ROLE jsilva LOGIN
             PASSWORD 'xyz321'
             VALID UNTIL '2014-12-31';
```

Nesse exemplo, criamos uma role com opção VALID UNTIL que informa uma data de expiração para a senha do usuário. Depois dessa data, a role não mais poderá ser utilizada.

Agora vamos criar uma role que possa criar outras roles. Para isso, basta informar a opção CREATEROLE (tudo junto) ao comando:

```
postgres=#   CREATE ROLE moliveira LOGIN
             PASSWORD 'xyz321'
             CREATEROLE;
```

Importante frisar que apenas superusuários ou quem possui o privilégio CREATEROLE pode criar roles.

Agora vamos criar uma role com o comportamento de grupo. Basta criar uma role simples:

```
postgres=#   CREATE ROLE contabilidade;
```

Depois, adicionamos usuários ao grupo, que na sintaxe do PostgreSQL significa fornecer uma role a outra role:

```
postgres=#   GRANT contabilidade TO jsilva;
postgres=#   GRANT contabilidade TO moliveira;
```

O comando *GRANT* fornece um privilégio para uma role e será tratado em mais detalhes logo à frente. Nesse exemplo, fornecemos uma role – contabilidade – para outras roles, jsilva e moliveira. Assim essas duas roles recebem todos os privilégios que a role contabilidade tiver.

Esse conceito fica muito mais simples de entender se assumirmos jsilva e moliveira como usuários e contabilidade como um grupo.

Outros atributos importantes das roles são:

- **SUPERUSER**: fornece a role o privilégio de superusuário. Roles com essa opção não precisam ter nenhum outro privilégio;
- **CREATEDB**: garante a role o privilégio de poder criar bases de dados;
- **REPLICATION**: roles com esse atributo podem ser usadas para replicação.

Pode parecer estranha a existência da opção LOGIN, pois uma role que não pode conectar-se parece inútil, mas isso se aplica para a criação de grupos ou roles para funções administrativas.

Exclusão de roles

Para remover uma role, simplesmente use o comando *DROP ROLE*:

```
postgres=#   DROP ROLE jsilva;
```

Entretanto, para uma role ser removida, ela não pode ter nenhum privilégio ou ser dona de objetos ou bases. Se houver, ou os objetos deverão ser excluídos previamente ou deve-se revogar os privilégios e alterar os donos.

Para remover todos os objetos de uma role, é possível utilizar o comando *DROP OWNED*:

```
postgres=#   DROP OWNED BY jsilva;
```

Serão revogados todos os privilégios fornecidos a role e serão removidos todos os objetos na base atual, caso não tenham dependência de outros objetos. Caso queira remover os objetos que dependem dos objetos da role, é possível informar o atributo CASCADE. Porém, deve-se ter em mente que isto pode remover objetos de outras roles. Este comando não remove bases de dados e tablespaces.

Caso deseje alterar o dono dos objetos da role que será removida, é possível usar o comando *REASSIGN OWNED*:

```
postgres=#   REASSIGN OWNED BY jsilva TO psouza;
```

Modificando roles

Como na maioria dos objetos no PostgreSQL, as roles também podem ser modificadas com um comando *ALTER*. A instrução ALTER ROLE pode modificar todos os atributos definidos pelo CREATE ROLE. No entanto, o ALTER ROLE é muito usado para fazer alterações específicas em parâmetros de configuração definidos globalmente no arquivo *postgresql.conf* ou na linha de comando.

Por exemplo, no arquivo *postgresql.conf* pode estar definido o parâmetro WORK_MEM com 4MB, mas para determinado usuário que executa muitos relatórios com consultas pesadas, podemos definir, com a opção SET, um valor diferente. Por exemplo:

```
postgres=#   ALTER ROLE psouza SET WORK_MEM = '8MB';
```

Para desfazer uma configuração específica para uma role, use o atributo RESET:

```
postgres=#   ALTER ROLE psouza RESET WORK_MEM;
```

Privilégios

Para fornecer e remover permissões em objetos e bases de dados, usamos os comandos *GRANT* e *REVOKE*.

GRANT

Quando se cria uma base de dados ou um objeto em uma base, sempre é atribuído um dono. Caso nada seja informado, será considerado dono a role que está executando o comando. O dono possui direito de fazer qualquer coisa nesse objeto ou base, mas os demais usuários precisam receber explicitamente um privilégio. Esse privilégio é concedido com o comando *GRANT*.

O GRANT possui uma sintaxe rica que varia de acordo com o objeto para o qual se está fornecendo o privilégio.

Para exemplificar, considere a criação de uma base para os sistemas contábeis. Para atribuir como dono a role gerente, que pertence ao Gerente da contabilidade, e que administrará toda a base, o seguinte comando deve ser utilizado:

```
postgres=#  CREATE DATABASE sis_contabil OWNER gerente;
```

O gerente por sua vez conecta em sua nova base, cria schemas e fornece os acessos que considera necessários:

```
sis_contabil=>       CREATE SCHEMA controladoria;
sis_contabil=>       GRANT CREATE ON SCHEMA controladoria TO controller;
```

O gerente criou o schema controladoria e forneceu o privilégio CREATE para a role controller, que agora pode criar tabelas, visões e quaisquer outros objetos dentro do schema.

```
sis_contabil=>       CREATE SCHEMA geral;
sis_contabil=>       GRANT USAGE ON SCHEMA geral TO contabilidade;
```

Foi então criado o schema geral e fornecida permissão USAGE para o grupo contabilidade. As roles jsilva e moliveira podem acessar objetos no schema geral, mas ainda dependem de também receber permissões nesses objetos. Eles não podem criar objetos nesse schema, mas apenas com o privilégio USAGE.

Uma vez criadas as tabelas e demais objetos no schema geral, o gerente pode fornecer os privilégios para a role contabil, que é o usuário usado pela aplicação:

```
sis_contabil=>       GRANT CONNECT ON DATABASE sis_contabil TO contabil;
sis_contabil=>       GRANT USAGE ON SCHEMA geral TO contabil;
sis_contabil=>       GRANT SELECT, INSERT, UPDATE, DELETE
                             ON ALL TABLES IN SCHEMA geral
                             TO contabil;
```

Note que a primeira concessão é o acesso à base de dados com o privilégio CONNECT, depois o acesso ao schema com USAGE e por fim o facilitador ALL TABLES, para permitir que o usuário da aplicação possa ler e gravar dados em todas as tabelas do schema.

É possível fornecer também permissões mais granulares para o grupo contabilidade nas tabelas do schema geral, conforme o exemplo:

```
sis_contabil=>       GRANT SELECT ON geral.balanco TO contabilidade;
sis_contabil=>       GRANT EXECUTE ON FUNCTION geral.lancamento()
                             TO contabilidade;
```

Nesses exemplos, foram fornecidas permissão de leitura de dados na tabela balanço, e permissão de execução da função lancamento() ao grupo contabilidade.

Repasse de privilégios

Quando uma role recebe um privilégio com GRANT, é possível que ela possa repassar esses mesmos privilégios para outras roles.

```
sis_contabil=>       GRANT SELECT, INSERT ON geral.contas
                             TO moliveira
                             WITH GRANT OPTION;
```

Nesse exemplo, a role moliveira ganhou permissão para consultar e adicionar dados na tabela geral.contas. Com a instrução WITH GRANT OPTION, ela tem o direito de repassar essa mesma permissão para outras roles. Por exemplo, a role moliveira pode conectar-se e executar:

```
sis_contabil=>       GRANT SELECT, INSERT ON geral.contas
                     TO jsilva;
```

Porém, a role moliveira não pode fornecer a role UPDATE ou DELETE na tabela, pois ela não possui esse privilégio. Assim, se moliveira executar o comando:

```
sis_contabil=>       GRANT ALL ON geral.contas TO jsilva;
```

A role jsilva receberá apenas INSERT e SELECT, que são os privilégios que o grant moliveira possui e não todos os privilégios existentes para a tabela.

Privilégios de objetos

Os privilégios de objetos são relativamente intuitivos se considerarmos o significado de cada um deles em inglês. Apresentamos alguns dos principais objetos e seus respectivos privilégios.

Base de dados:

- **CONNECT**: permite à role conectar-se à base;
- **CREATE**: permite à role criar schemas na base;
- **TEMP or TEMPORARY**: permite à role criar tabelas temporárias na base.

Exemplos:

```
curso=#      GRANT CONNECT, TEMP ON DATABASE curso TO aluno;
```

Schemas:

- **CREATE**: permite criar objetos no schema;
- **USAGE**: permite acessar objetos do schema, mas ainda depende de permissão no objeto.

Tabelas

SELECT, INSERT, UPDATE e DELETE são triviais para executar as respectivas operações.

Para UPDATE e DELETE com cláusula WHERE, é necessário também que a role possua SELECT na tabela.

Com exceção do DELETE, para os demais privilégios é possível especificar colunas. Os exemplos a seguir fornecem permissão para a role psouza inserir, ler e atualizar dados apenas na coluna descrição.

```
sis_contabil=#               GRANT SELECT (descricao),
                             INSERT (descricao),
                             UPDATE (descricao)
ON geral.balanco
TO psouza;
```

Demais colunas serão preenchidas com o valor default, no caso da inserção.

Outros privilégios para tabela são:

- **TRUNCATE**: permite que a role execute essa operação na tabela;
- **TRIGGER**: permite que a role crie triggers na tabela;
- **REFERENCES**: permite que a role referencie essa tabela quando criando uma foreign key em outra.

Truncate
É uma operação que elimina todos os dados de uma tabela mais rapidamente do que o DELETE.

Visões/views

São tratadas no PostgreSQL praticamente como uma tabela devido à sua implementação. Tabelas, visões e sequências (sequences) são vistas genericamente como relações.

Por isso, visões podem receber GRANTS de escrita como INSERT e UPDATE. Porém, o funcionamento de comandos de inserção e atualização em uma view dependerá da existência de RULES ou TRIGGERS para tratá-las.

Sequências/sequences

Os seguintes privilégios se aplicam às sequências:

- **USAGE**: permite executar as funções currval e nextval sobre a sequência;
- **SELECT**: permite executar a função curval;
- **UPDATE**: permite o uso das funções nextval e setval.

Sequences são a forma que o PostgreSQL fornece para implementar campos autoincrementais. Elas são manipuladas através das funções:

- **curval**: retorna o valor atual da sequence;
- **nextval**: incrementa a sequence e retorna o novo valor;
- **setval**: atribui à sequence um valor informado como argumento.

Funções/procedures

As funções, ou procedures, possuem apenas o privilégio EXECUTE:

```
sis_contabil=#               GRANT EXECUTE
ON FUNCTION validacao(cpf bigint, nome varchar)
TO contabilidade;
```

Por padrão, a role PUBLIC recebe permissão de EXECUTE em todas as funções. Caso queira evitar esse comportamento, use REVOKE para tirar a permissão após a criação da função.

Cláusula ALL

Para sequências, visões e tabelas, existe a cláusula ALL ... IN SCHEMA, que ajuda a fornecer permissão em todos os objetos daquele tipo no schema. Essa é uma tarefa extremamente comum e antes dessa instrução era necessário criar scripts que lessem o catálogo para encontrar todos os objetos e preparar o comando de *GRANT*. Sintaxe:

```
sis_contabil=#       GRANT USAGE
                     ON ALL SEQUENCES IN SCHEMA geral
                     TO contabilidade;
```

Vários outros objetos do PostgreSQL, tais como languages, types, domains e large objects, possuem privilégios que podem ser manipulados. A sintaxe é muito semelhante ao que foi mostrado e pode ser consultada na documentação online do PostgreSQL.

REVOKE

O comando *REVOKE* remove privilégios fornecidos com GRANT.

O REVOKE revogará um privilégio específico concedido em um objeto ou base, porém não significa que vai remover qualquer acesso que a role possua. Por exemplo, uma role pode possuir um GRANT de SELECT direto em uma tabela, e ao mesmo tempo fazer parte de um grupo que também possui acesso a tabela.

```
sis_contabil=#                GRANT SELECT ON geral.balanco TO jsilva;
sis_contabil=#                GRANT SELECT ON geral.balanco TO contabilidade;
```

Fazer o REVOKE da role direta não impedirá a role de acessar a tabela, pois ainda terá o privilégio através do grupo.

```
sis_contabil=#                REVOKE SELECT ON geral.balanco FROM jsilva;
```

É possível revogar apenas o direito de repassar o privilégio que foi dado com WITH GRANT OPTION, sem revogar o privilégio em si.

```
sis_contabil=>                REVOKE GRANT OPTION FOR
                              SELECT, INSERT ON geral.contas
                              FROM moliveira;
```

Nesse exemplo, a instrução GRANT OPTION FOR não remove o acesso de INSERT e SELECT da role moliveira, apenas o direito de repassar essas permissões para outras roles.

Usando GRANT e REVOKE com grupos

Como vimos no tópico sobre ROLES, podemos dar a elas o comportamento de usuários e grupos. A forma de fornecer essa interpretação é por meio do comando *GRANT*, fornecendo uma role a outra role. É um uso diferente do comando *GRANT* que vimos nas últimas sessões, onde fornecemos privilégios em objetos.

Anteriormente incluímos os usuários jsilva e moliveira no grupo contabilidade. A forma de fazer isso é fornecer o privilégio contabilidade, no caso uma role, para as outras roles:

```
postgres=#   GRANT contabilidade TO jsilva;
postgres=#   GRANT contabilidade TO moliveira;
```

Nesse momento, demos a semântica de grupo à role contabilidade e de membros do grupo às roles jsilva e moliveira.

Analogamente, pode-se remover um usuário do grupo, com a instrução REVOKE:

```
postgres=#   REVOKE contabilidade FROM moliveira;
```

Consultando os privilégios

Para consultar os privilégios existentes, no psql você pode usar o comando *\dp* para listar as permissões de tabelas, visões e sequences:

```
curso=>     \dp cidades;
```

```
curso=# GRANT SELECT, INSERT ON cidades TO aluno;
GRANT
curso=# \dp cidades;
                              Access privileges
 Schema |  Name   | Type  |     Access privileges     | Column access privileges
--------+---------+-------+---------------------------+--------------------------
 public | cidades | table | postgres=arwdDxt/postgres+|
        |         |       | aluno=ar/postgres         |
(1 row)

curso=#
```

Figura 4.1
Consultando GRANTs com psql.

A figura 4.1 mostra a saída do comando, a coluna "Access privileges" contém os usuários, permissões e quem forneceu o privilégio na tabela. A primeira linha mostra as permissões do dono da tabela, no caso o postgres:

```
postgres=arwdDxt/postgres
|          |     |_____________ Quem forneceu a permissão
|          |
|          |________ Privilégios, cada letra representa um
|
|_____________ Role que recebeu os privilégios
```

Essa primeira linha indica a permissão padrão que todo dono de objeto recebe automaticamente.

Na segunda linha, temos: **aluno**=**ar**/postgres

Significa que a role aluno recebeu os privilégios "ar" da role postgres.

A tabela a seguir mostra o significado das letras:

a	INSERT (append)
r	SELECT (read)
w	UPDATE (write)
d	DELETE
D	TRUNCATE
x	REFERENCES
t	TRIGGER
X	EXECUTE
U	USAGE
C	CREATE
c	CONNECT
T	TEMPORARY

Tabela 4.1
Privilégios.

Pela tabela, podemos ver que o postgres possui todos os privilégios possíveis para uma tabela, arwdDxt:

INSERT(a), SELECT(r), UPDATE(w), DELETE(d), TRUNCATE(D), REFERENCES(x) e TRIGGER(t).

Já a role aluno possui apenas INSERT(a) e SELECT(r).

```
curso=# GRANT SELECT ON cidades TO professor WITH GRANT OPTION;
GRANT
curso=# GRANT INSERT, UPDATE, DELETE ON cidades TO professor;
GRANT
curso=# \dp cidades;
                              Access privileges
 Schema |  Name   | Type  |     Access privileges     | Column access privileges
--------+---------+-------+---------------------------+--------------------------
 public | cidades | table | postgres=arwdDxt/postgres+|
        |         |       | aluno=ar/postgres        +|
        |         |       | professor=ar*wd/postgres  |
(1 row)

curso=#
```

Figura 4.2
Privilégios com WITH GRANT OPTION.

Além dos privilégios, você pode ver um * entre as letras. Isso significa que a role pode repassar o privilégio, pois recebeu o atributo WITH GRANT OPTION.

No exemplo da figura 4.2, a role professor pode repassar apenas o privilégio SELECT(r).

É possível também verificar os privilégios nas bases de dados, pelo psql, com o comando *\l*. No exemplo abaixo da figura 4.3, além da primeira linha que se refere ao owner postgres, a role aluno possui privilégios para criar tabelas temporárias TEMP(T) e CONNECT(c). Já a role professor possui essas duas e ainda o privilégio de CREATE(C).

```
curso=# \l
                                 List of databases
Name  |  Owner   | Encoding |  Collate    |  Ctype      |   Access privileges
------+----------+----------+-------------+-------------+-----------------------
curso | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =Tc/postgres          +
      |          |          |             |             | postgres=CTc/postgres +
      |          |          |             |             | aluno=Tc/postgres     +
      |          |          |             |             | professor=CTc/postgres
```

Figura 4.3
Privilégios de bases de dados com o comando \l.

Para exibir privilégios de schemas, usamos o comando *\dn+*.

No exemplo mostrado pela figura 4.4, vemos o schema avaliacao, que possui privilégio padrão, pois a coluna "Access privileges" está vazia. Para o schema extra vemos o dono postgres com privilégios USAGE(U) e CREATE(C), e a role aluno com acesso apenas USAGE.

```
curso=# \dn+
                        List of schemas
Name       |  Owner   | Access privileges         | Description
-----------+----------+---------------------------+-----------------------
 avaliacao | postgres |                           |
 extra     | postgres | aluno=U/postgres         +|
           |          | postgres=UC/postgres      |
 public    | postgres | postgres=UC/postgres+     | standard public schema
           |          | =UC/postgres              |
```

Figura 4.4
Privilégios de Schemas com o comando \dn+ do psql.

> Ao executar \dp, se a coluna "access privileges" estiver vazia, significa que está com apenas os privilégios default. Após o objeto receber o primeiro GRANT, esta coluna passará a mostrar todos os privilégios.

Além dos comandos do psql, podemos consultar privilégios existentes em bases, schemas e objetos através de diversas visões do catálogo do sistema e também através de funções de sistema do PostgreSQL.

Por exemplo, a função has_table_privilege(user, table, privilege) retorna se determinado usuário possui determinado privilégio na tabela.

```
curso=# select has_table_privilege('aluno','cidades','UPDATE');
 has_table_privilege
---------------------
 f
(1 row)

curso=# select has_table_privilege('aluno','cidades','SELECT');
 has_table_privilege
---------------------
 t
(1 row)

curso=#
```

Figura 4.5
Consulta privilégios do usuário/tabela com funções de sistema.

Existem dezenas de funções desse tipo para os mais variados objetos.

Por fim, uma maneira muito fácil de consultar os privilégios em objetos é através da ferramenta gráfica pgAdmin III. Apesar de ser um projeto à parte, não fazendo parte do pacote do PostgreSQL, consultar as permissões de objetos, schemas e bases com ela é muito simples.

O pgAdmin interpreta a coluna Access Privileges do catálogo e monta os comandos *GRANTs* que geraram o privilégio. Basta clicar sob o objeto e o código do objeto e seus privilégios são mostrados no quadro SQL PANEL, como mostrado na figura 5.6.

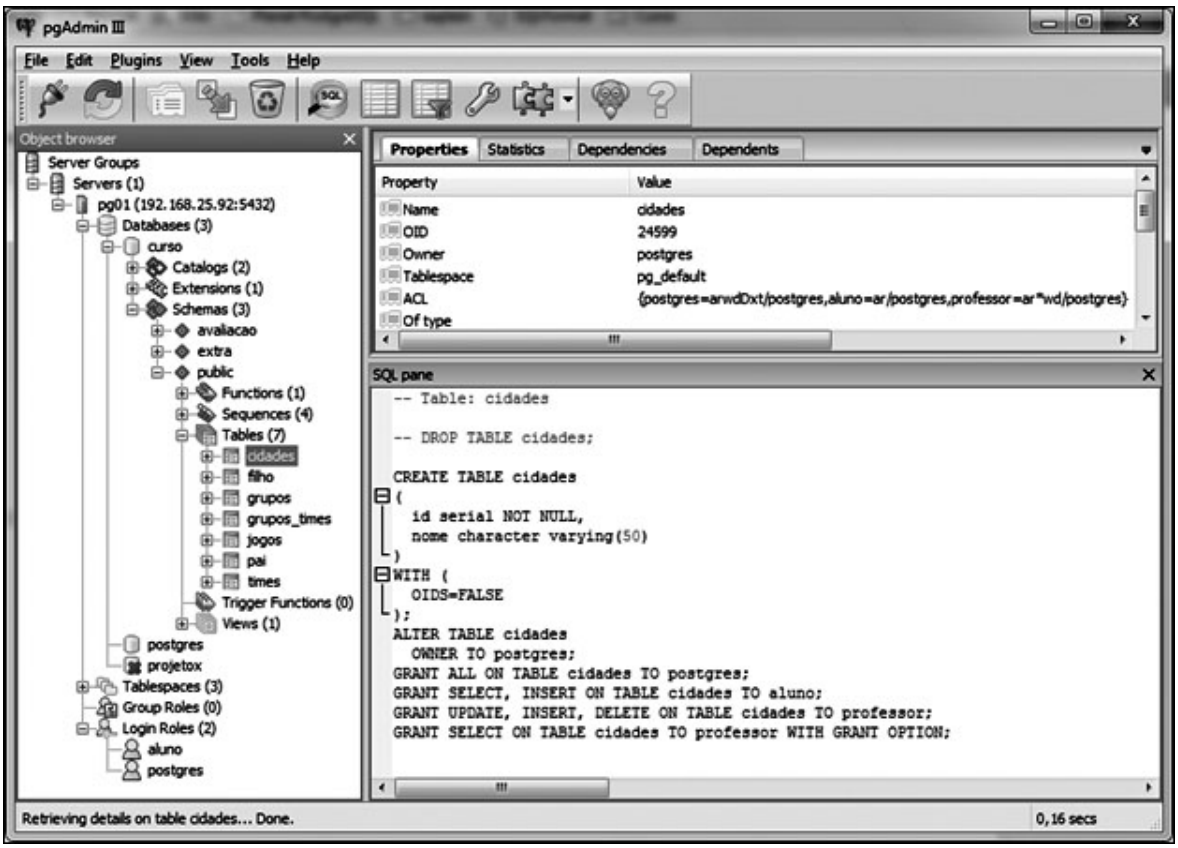


Figura 4.6
Consultando privilégios com pgAdmin III.

Gerenciando autenticação

Uma parte importante do mecanismo de segurança do PostgreSQL é o controle de autenticação de clientes pelo arquivo *pg_hba.conf*, onde HBA significa Host Based Authentication.

Basicamente, nesse arquivo de configuração temos várias linhas, onde cada linha é um registro indicando permissão de acesso de uma role, a partir de um endereço IP a determinada base.

Por padrão a localização do arquivo é "PGDATA/pg_hba.conf", mas nome e diretório podem ser alterados.

O formato de cada registro no arquivo é:

```
tipo conexão     base de dados    usuário          endereço         método
```

Um exemplo de registro que permitiria a role aluno conectar na base de dados curso poderia ser assim:

```
host             curso            aluno            10.5.15.40/32    md5
```

Essa linha host "diz" que uma conexão IP, na base curso, com usuário aluno, vindo do endereço IP 10.5.15.40 autenticando por md5 pode passar.

Um outro exemplo seria permitir um grupo de usuários vindos de determinada rede em vez de uma estação específica.

```
host             contabil         +contabilidade   172.22.3.0/24    md5
```

Nesse exemplo, qualquer usuário do grupo contabilidade, acessando a base contabil, vindo de qualquer máquina da rede 172.22.3.x e autenticando por md5 é permitido.

Destacamos o sinal +, que identifica um grupo e a máscara de rede /24.

Há diversas opções de valores para cada campo. Veja algumas nas tabelas:

Tipo de conexão	
local	Conexões locais do próprio servidor por unix-socket.
host	Conexões por IP, com ou sem SSL.
hostssl	Conexões somente por SSL.

Base de dados	
nome da(s) base(s)	Uma ou mais bases de dados separada por vírgula.
all	Acesso a qualquer base.
replication	Utilizado exclusivamente para permitir a replicação.

Usuário	
role(s)	Um ou mais usuários, separados por vírgula.
+grupo(s)	Um ou mais grupos, separados por vírgula e precedidos de +.
all	Acesso de qualquer usuário.

Endereço	
Um endereço IP v4	Um endereço IP v4 como 172.22.3.10/32.
Um rede IP v4	Uma rede IP v4 como 172.22.0.0/16.
Um endereço IP v6	Um endereço IP v6 como fe80::a00:27ff:fe78:d3be/64.
Um rede IP v6	Uma rede IP v6 como fe80::/60.
0.0.0.0/0	Qualquer endereço IPv4.
::/0	Qualquer endereço IPv6.
all	Qualquer IP.

É possível usar nomes de máquinas em vez de endereços IP, porém isso pode gerar problemas de lentidão no momento da conexão, dependendo da sua infraestrutura DNS.

Método	
trust	Permite conectar sem restrição, sem solicitar senha permitindo que qualquer usuário possa se passar por outro.
md5	Autenticação com senha encriptada com hash MD5.
password	Autenticação com senha em texto pleno.
ldap	Autenticação usando um servidor LDAP.
reject	Rejeita a conexão.

Existem diversas outras opções de métodos de autenticação.

```
# TYPE  DATABASE        USER            ADDRESS             METHOD

# "local" is for Unix domain socket connections only
local   all             all                                 trust
# IPv4 local connections:
host    all             all             127.0.0.1/32        trust
# IPv6 local connections:
host    all             all             ::1/128             trust
```

Figura 4.7
Permissões padrões do arquivo pg_hba.conf.

Quando o banco é inicializado com o initdb, um arquivo *pg_hba.conf* modelo é criado com algumas concessões por padrão. A figura 4.7 mostra os registros que permitem a qualquer conexão do próprio servidor local, seja por unix-socket (local), por IPv4 (127.0.0.1) ou por IPv6 (::1), conectar em qualquer base com qualquer usuário e sem solicitar senha.

Se não houver uma entrada no arquivo *pg_hba.conf* que coincide com a tentativa de conexão, o acesso será negado. Você verá uma mensagem como na figura 4.8:

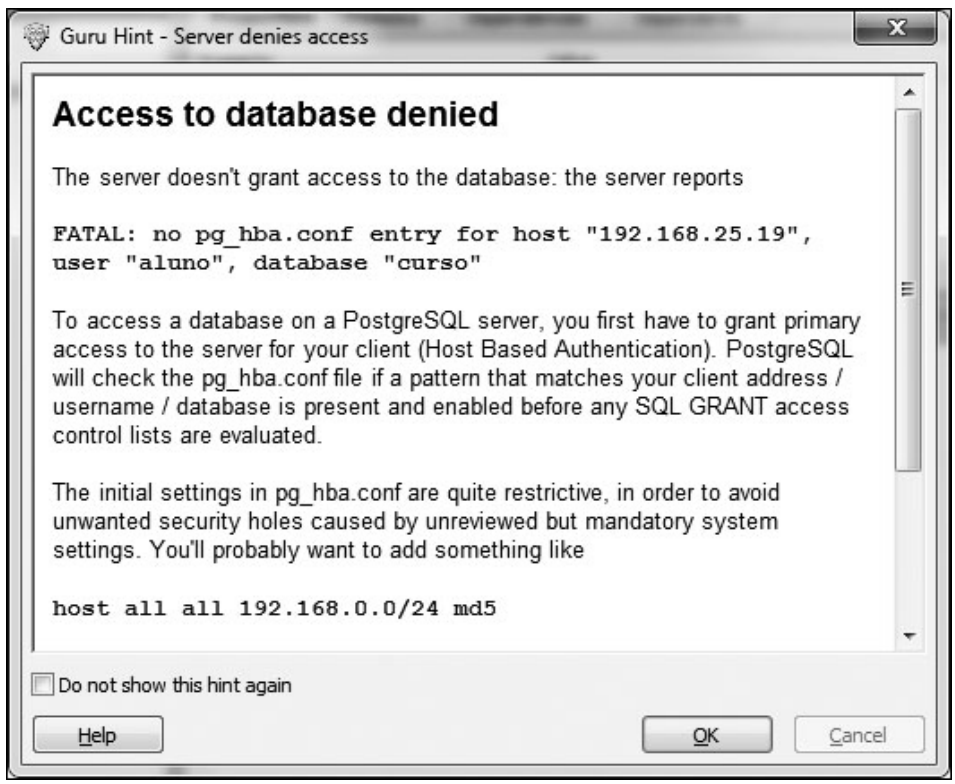


Figura 4.8
Acesso negado por falta de autorização no *pg_hba.conf*.

Quando alterar o pg_hba.conf é necessário fazer a reconfiguração com pg_ctl **reload**.

Boas práticas

Apresentamos a seguir algumas dicas relacionadas à segurança que podem ajudar na administração do PostgreSQL.
São elas:

- Utilize roles de grupos para gerenciar as permissões. Como qualquer outro serviço – não somente Bancos de Dados –, administrar permissões para grupos e apenas gerenciar a inclusão e remoção de usuários do grupo torna a manutenção de acessos mais simples;
- Remova as permissões da role public e o schema public se não for utilizá-lo. Retire a permissão de conexão na base de dados, de uso do schema e qualquer privilégio em objetos. Remova também no template1 para remover das futuras bases;
- Seja meticuloso com a *pg_hba.conf*, em todos os seus ambientes. No início de um novo servidor, ou mesmo na criação de novas bases, pode surgir a ideia de liberar todos os acessos à base e controlar isso mais tarde. Não caia nessa armadilha! Desde o primeiro acesso, somente forneça o acesso exato necessário naquele momento. Sempre crie uma linha para cada usuário em cada base vindo de cada estação ou servidor. É comum sugerir a liberação de acesso a partir de uma rede de servidores de aplicação e não um servidor específico. Evite isso;
- Somente use trust para conexões locais no servidor e assim mesmo se você confia nos usuários que possuem acesso ao servidor ou ninguém além dos administradores de banco possuem acesso para conectar-se no Sistema Operacional;
- Documente suas alterações. É possível associar comentários a roles. Documente o nome real do usuário, utilidade do grupo, quem e quando solicitou a permissão etc. Também comente suas entradas no arquivo *pg_hba.conf*. De quem é o endereço IP, estação de usuário ou nome do servidor, quem e quando foi solicitado;
- Faça backup dos seus arquivos de configuração antes de cada alteração.

5

Monitoramento do ambiente

objetivos

Conhecer ferramentas e recursos para ajudar na tarefa de monitoramento do PostgreSQL e do ambiente operacional; Compreender as principais informações que devem ser acompanhadas e medidas no Sistema Operacional e no banco, e conhecer as opções para monitorá-las.

conceitos

ps; top; iostat; vmstat; Nagios; Cacti; Zabbix; pg_Fouine; pg_Badger; pg_stat_activity; pg_locks e tps.

Monitoramento

Depois que instalamos e configuramos um serviço em produção é que realmente o trabalho se inicia. Acompanhar a saúde de um ambiente envolve monitorar diversos componentes, por vezes nem todos sob a alçada das mesmas pessoas. Essas partes podem ser, por exemplo, o serviço de Banco de Dados em si, a infraestrutura física de rede, serviços de rede como firewall e DNS, hardware do próprio servidor de banco, Sistema Operacional do servidor de banco, infraestrutura de storage e carga proveniente de servidores de aplicação.

Administradores de ambientes Unix/Linux acostumados a monitoramento de serviços talvez já estejam bastante preparados para monitorar um servidor PostgreSQL, pois muitas das ferramentas usadas são as mesmas, como o ps, top, iostat, vmstat e outras. Isso não é por acaso, mas sim porque muitos dos problemas enfrentados na administração de servidores de Banco de Dados estão relacionados aos recursos básicos de um sistema computacional: memória, processador e I/O.

Suponha um ambiente que não sofreu atualização recente. O PostgreSQL não foi atualizado, o SO não foi atualizado, nenhum serviço ou biblioteca foi atualizado e nem a aplicação.
Se nos defrontamos com um problema de lentidão, normalmente temos duas hipóteses iniciais: ou aumentou a carga sobre o banco ou temos um problema em algum desses recursos.

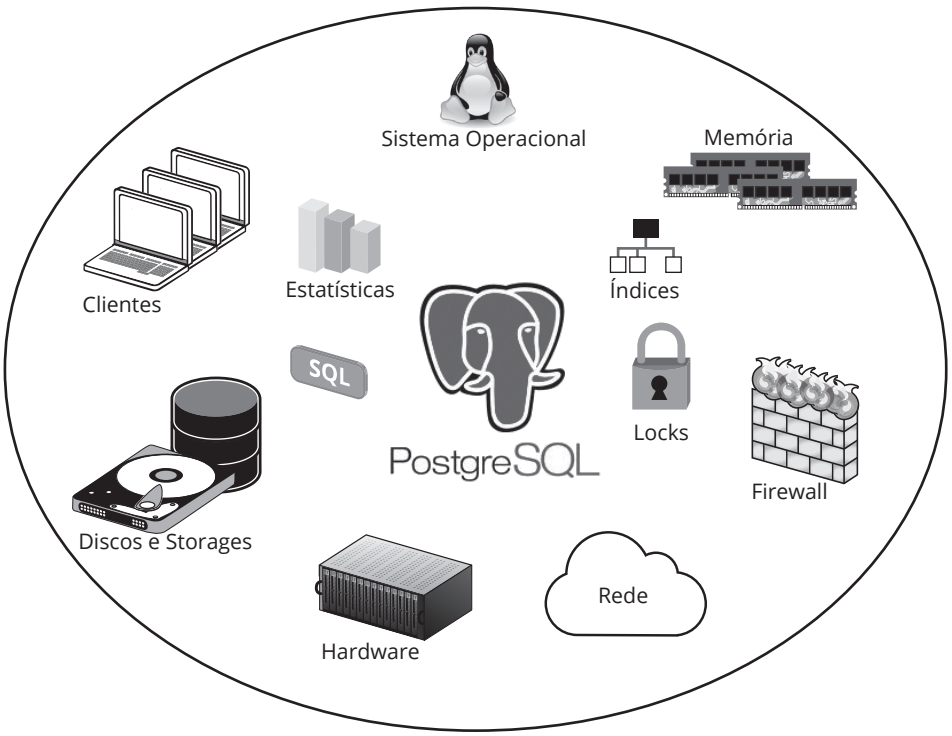


Figura 5.1
Diferentes aspectos do PostgreSQL, que devem ser monitorados.

A primeira coisa em um servidor Linux seria consultar o load, provavelmente com o top. Se a carga está maior do que de costume, investigamos se há algum processo abusando de CPU ou talvez excesso de processos. Em seguida podemos verificar se há memória suficiente – com o free ou o próprio top, por exemplo. Pode estar ocorrendo swap por falta de memória, ou por outros motivos. Nesse caso podemos verificar com o sar ou vmstat.
O terceiro passo talvez seja verificar se há gargalo de I/O. Podemos usar novamente o top para uma informação básica ou o iostat.

Um administrador de Banco de Dados deve também suspeitar se há processos bloqueados. Nesse caso precisaremos usar uma ferramenta específica para o PostgreSQL como o pg_activity e pgAdmin, ou consultar tabelas do catálogo que mostrem o status dos processos e os locks envolvidos. Uma transação pode estar aberta há muito tempo e bloqueando recursos, o que pode ser verificado pela view do catálogo pg_stat_activity ou pelo próprio pg_activity. Um processo pode estar demorando demais e não desocupando CPUs ou gerando muitos arquivos temporários, hipótese que pode ser confirmada pela análise dos logs do PostgreSQL.

Ainda poderíamos considerar algum erro no nível do Sistema Operacional, verificando a syslog ou as mensagens do kernel, onde algum indício relacionado a hardware ou falhas de rede pode também ser encontrado.

Todas essa ações são comuns e necessárias, mas são reativas, sendo executadas depois que um problema já apareceu. A melhor forma de monitoramento é tentar identificar um padrão de funcionamento de seu ambiente, o que é considerado uma situação normal. Levantar o que é considerada uma carga de trabalho (load) normal, determinando valores considerados normais para o número de transações, por segundo, operações de I/O por segundo, número de processos, tempo máximo das queries, tipos de queries etc. Assim, encontrando o cenário normal para a sua infraestrutura, pode-se passar a monitorá-lo com ferramentas que geram gráficos históricos e alertam em caso de um indicador sair da sua faixa de normalidade. Existem diversas ferramentas dessa natureza, algumas muito conhecidas entre os administradores Linux, tais como o Nagios, Cacti e o Zabbix.

Em especial para o PostgreSQL, você pode configurar seus logs para um formato que possa ser processado automaticamente por ferramentas que geram relatórios, destacando as queries mais lentas, as mais executadas, as que mais geraram arquivos temporários, entre muitos outros indicadores que ajudam em muito a controlar o comportamento do banco. Duas ferramentas para essa finalidade são o pg_Fouine e o excelente pg_Badger.

Monitorando pelo Sistema Operacional

- top.
- Vmstat.
- Iostat.
- sar e Ksar.

Em função da arquitetura do PostgreSQL, que trata cada conexão por um processo do SO, podemos monitorar a saúde do banco monitorando os processos do SO pertencentes ao PostgreSQL.

A seguir analisaremos os utilitários e ferramentas que permitem esse monitoramento.

top

Para monitorar processos no Linux talvez a ferramenta mais famosa seja o top. O top é um utilitário básico na administração de servidores e podemos extrair informações valiosas. Basta executar "top" na shell para invocá-lo. Pode ser útil com o PostgreSQL exibir detalhes do comando com *-c* e filtrar apenas os processos do usuário postgres com *-u*.

```
$   top -p postgres -c
```

```
top - 21:11:28 up 3 days, 21:59,  1 user,  load average: 1.96, 0.64, 0.28
Tasks:  79 total,   4 running,  75 sleeping,   0 stopped,   0 zombie
Cpu(s): 33.7%us, 32.0%sy,  0.0%ni,  7.7%id,  0.0%wa,  0.0%hi, 26.5%si,  0.0%st
Mem:    507508k total,   501388k used,     6120k free,      292k buffers
Swap:   522236k total,      728k used,   521508k free,   443888k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
25805 postgres  20   0 40736 4288 3236 R 37.4  0.8   0:06.61 postgres: aluno projetox 192.168.25.19(54108) SELECT
25806 postgres  20   0 40736 4284 3220 R 36.9  0.8   0:04.04 postgres: aluno projetox 192.168.25.19(54109) SELECT
25809 postgres  20   0 40736 4284 3232 S  5.8  0.8   0:04.68 postgres: aluno curso 192.168.25.19(54110) SELECT
  516 syslog    20   0 30040  808  480 S  3.7  0.2   4:12.06 rsyslogd -c5
25799 postgres  20   0  2720  840  616 R  1.1  0.2   0:02.54 top -c
25813 postgres  20   0 40364 2320 1452 R  1.1  0.5   0:00.02 postgres: autovacuum worker process   curso
  927 root      20   0  9112  404  152 S  0.5  0.1   9:54.46 /usr/sbin/VBoxService
  943 postgres  20   0 40436 2132 1200 S  0.5  0.4   2:03.60 postgres: autovacuum launcher process
  944 postgres  20   0 10360 1288  724 S  0.5  0.3   3:50.86 postgres: stats collector process
22726 root      20   0     0    0    0 S  0.5  0.0   0:47.20 [kworker/0:1]
25344 postgres  20   0  9652 1128  512 S  0.5  0.2   0:03.56 sshd: postgres@pts/0
    1 root      20   0  3540 1116  504 S  0.0  0.2   0:03.09 /sbin/init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.40 [kthreadd]
    3 root      20   0     0    0    0 S  0.0  0.0   0:47.51 [ksoftirqd/0]
```

Figura 5.2
Monitoramento de processos com top.

Com o top, podemos verificar facilmente o load médio dos últimos 1min, 5min e 15min, números de processos totais, número de processos em execução, total de memória usada, livre, em cache ou em swap. Percentual de CPU para processos do kernel(sys), demais processos(us) e esperando I/O (wa). Mas, principalmente, podemos verificar os processos que estão consumindo mais CPU.

Existem diversas variações do top: uma opção com uma interface mais amigável é o htop.

Merece destaque a coluna S, que representa o estado do processo. O valor "D" indica que o processo está bloqueado, geralmente aguardando operações de disco. Deve-se acompanhar se está ocorrendo com muita frequência ou por muito tempo.

vmstat

Outra importante ferramenta é a vmstat. Ela mostra diversas informações dos recursos por linha em intervalos de tempo passado como argumento na chamada. Para executar o vmstat atualizando as informações uma vez por segundo, basta o seguinte comando:

```
$   vmstat 1
```

```
postgres@pg01:~$ vmstat 1
procs -----------memory---------- ---swap-- -----io---- -system-- ----cpu----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa
 2  0    728   6036  45820 368780    0    0     1     9   13   17  0  1 99  0
 2  1    728   5628  45812 370684    0    0     0  8232  272  217  4 96  0  0
 2  0    728   5524  37424 380756    0    0     0  4132  276  182  4 96  0  0
 0  1    728   6328  34408 382784    0    0     0 45008  117   72  1 30  0 69
 0  1    728   6328  34408 383076    0    0     0     0   64   21  0  4  0 96
 1  0    728   6020  24980 393200    0    0     0     0  249  154  3 89  0  8
 1  1    728   6240  24988 394816    0    0     0  6892  267  211  3 97  0  0
 0  0    728   6356  25112 396772    0    0   124     8  242  186  3 90  3  4
 0  0    728   6292  25112 396796    0    0     0 41408   74   16  0 11 89  0
```

Figura 5.3
Monitorando seu ambiente com vmstat.

Na primeira parte, procs, o vmstat mostra números de processos. A coluna "r" são processos na fila prontos para executar, e "b" são processos bloqueados aguardando operações de I/O (que estariam com status D no top).

Na seção memory existem as colunas semelhantes como vimos com top: swap, livre e caches (buffer e cache). A diferença na análise com a vmstat é entender as tendências. Podemos ver no top que há, por exemplo, 100MB usados de swap. Mas com a vmstat podemos acompanhar esse número mudando, para mais ou para menos, para nos indicar uma tendência no diagnóstico de um problema.

A seção swap mostra as colunas swap in (si), que são dados saindo de disco para memória, e swap out (so), que são as páginas da memória sendo escritas no disco. Em uma situação considerada normal, o Swap nunca deve acontecer, com ambas as colunas sempre "zeradas". Qualquer anormalidade demanda a verificação do uso de memória pelos processos, podendo ser também originada por parâmetros de configuração mal ajustados ou pela necessidade do aumento de memória física.

Na vmstat, o ponto de vista é sempre da memória principal, então IN significa entrando na memória, e OUT saindo.

A seção io tem a mesma estrutura da seção swap, porém em relação a operações normais de I/O. A coluna blocks in (bi) indica dados lidos do disco para memória, enquanto a blocks out (bo) indica dados sendo escritos no disco.

As informações de memória, swap e I/O estão em blocos, por padrão de 1024 bytes. Use o parâmetro -Sm para ver os dados de memória em MBytes (não altera o formato de swap e io).

Na seção system, são exibidos o número de interrupções e trocas de contexto no processador. Servidores atuais, multiprocessados e multicore podem exibir números bem altos para troca de contexto; e altos para interrupções devido à grade atividade de rede.

Em cpu temos os percentuais de processador para processos de usuários (us), do kernel (si), não ocupado (id) e aguardando operações de I/O (wa). Um I/O wait alto é um alerta, indicando que algum gargalo de disco está ocorrendo. Percentuais de cpu para system também devem ser observados, pois se estiverem fora de um padrão comumente visto podem indicar a necessidade de se monitorar os serviços do kernel.

iostat

Se for necessário analisar situações de tráfego de I/O, a ferramenta iostat é indicada. Ela exibe informações por device em vez de dados gerais de I/O, como a vmstat. Para invocar a iostat, informamos um intervalo de atualização, sendo útil informar também a unidade para exibição dos resultados (o padrão é trabalhar com blocos de 512 bytes). A seguinte chamada atualiza os dados a cada cinco segundos e em MB:

```
$   iostat -m 5
```

```
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           7.41    0.01    1.31    3.70    0.00   87.56

Device:            tps    MB_read/s    MB_wrtn/s    MB_read    MB_wrtn
sde              75.13         1.64         1.01   15368614    9459382
sdf              16.70         1.82         0.01   17046452      92238
dm-0            624.63         2.35         2.32   21974363   21757447

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          14.65    0.00    1.25    4.07    0.00   80.03

Device:            tps    MB_read/s    MB_wrtn/s    MB_read    MB_wrtn
sde              5.40         0.00         0.92          0          4
sdf             57.20         6.17         0.00         30          0
dm-0          1934.20         0.00         7.56          0         37

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          14.70    0.00    1.21    2.89    0.00   81.19

Device:            tps    MB_read/s    MB_wrtn/s    MB_read    MB_wrtn
sde              5.20         0.00         0.92          0          4
sdf             37.40         4.26         0.00         21          0
dm-0          2160.20         0.00         8.44          0         42
```

Figura 5.4 Monitorando devices de I/O com iostat.

> O iostat pode não estar instalado em seu ambiente. Ele faz parte do pacote sysstat e pode ser instalado através do comando *sudo apt-get install sysstat* (Debian/Ubuntu) ou *sudo yum install sysstat* (Red Hat/CentOS).

O iostat exibe um cabeçalho com os dados já conhecidos de CPU e uma linha por device com as estatísticas de I/O. A primeira coluna é a tps, também conhecida como IOPS, que é o número de operações de I/O por segundo. Em seguida exibe duas colunas com MB lidos e escritos por segundo, em média. As últimas duas colunas exibem a quantidade de dados em MB lidos e escritos desde a amostra anterior, no exemplo, a cada 5 segundos.

Repare que a primeira amostra exibe valores altíssimos porque ela conta o total de dados lidos e escritos desde o boot do sistema.

Existe uma forma estendida do iostat que mostra mais informações, acionada através do uso do parâmetro -x:

```
$   iostat -x -m 5
```

Figura 5.5 Versão estendida do iostat.

```
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           3,09    0,00    0,96    0,96    0,00   94,99

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await  svctm  %util
cciss/c0d0p4      0,00     0,00    0,00    0,00     0,00     0,00     0,00     0,00    0,00   0,00   0,00
dm-0              0,00   165,00    8,00   94,00     0,08     1,01    21,88     0,24    2,33   0,23   2,30
dm-1              0,00     0,00  372,00    0,00    46,50     0,00   256,00     0,74    2,00   1,15  42,60
dm-2              0,00    47,00    0,00    9,00     0,00     0,22    49,78     0,01    0,56   0,56   0,50
```

Duas colunas merecem consideração na forma estendida: await e %util. A primeira é o tempo médio, em milissegundos, que as requisições de I/O levam para serem servidas (tempo na fila e de execução). A outra, %util, mostra um percentual de tempo de CPU em que requisições foram solicitadas ao device. Apesar de esse número não ser acurado para arrays de discos, storages e discos SSD, um percentual próximo de 100% é certamente um indicador de saturação.

```
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          15.97    0.00    2.18   19.50    0.00   62.35

Device:         rrqm/s   wrqm/s   r/s   w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await  svctm  %util
sde               0.99     0.00 269.31  0.00     2.08     0.00    15.85     1.18    4.40   3.64  97.92
sdf              52.48     0.00 33.66  0.00     5.89     0.00   358.12     0.14    4.18   4.15  13.96
dm-0              0.00     0.00  0.00 1604.95     0.00     6.27     8.00   397.03  277.63   0.05   7.62
```

A figura 5.6 mostra um exemplo de sistema próximo da saturação, com I/O wait próximo de 20% e %util do device sde próximo de 100%. Interessante notar que a carga de I/O naquela amostra não era alta, 2MB/s, porém estava apresentando fila e tempo de resposta (await) elevando.

Figura 5.6
Sistema com pico de I/O.

sar e Ksar

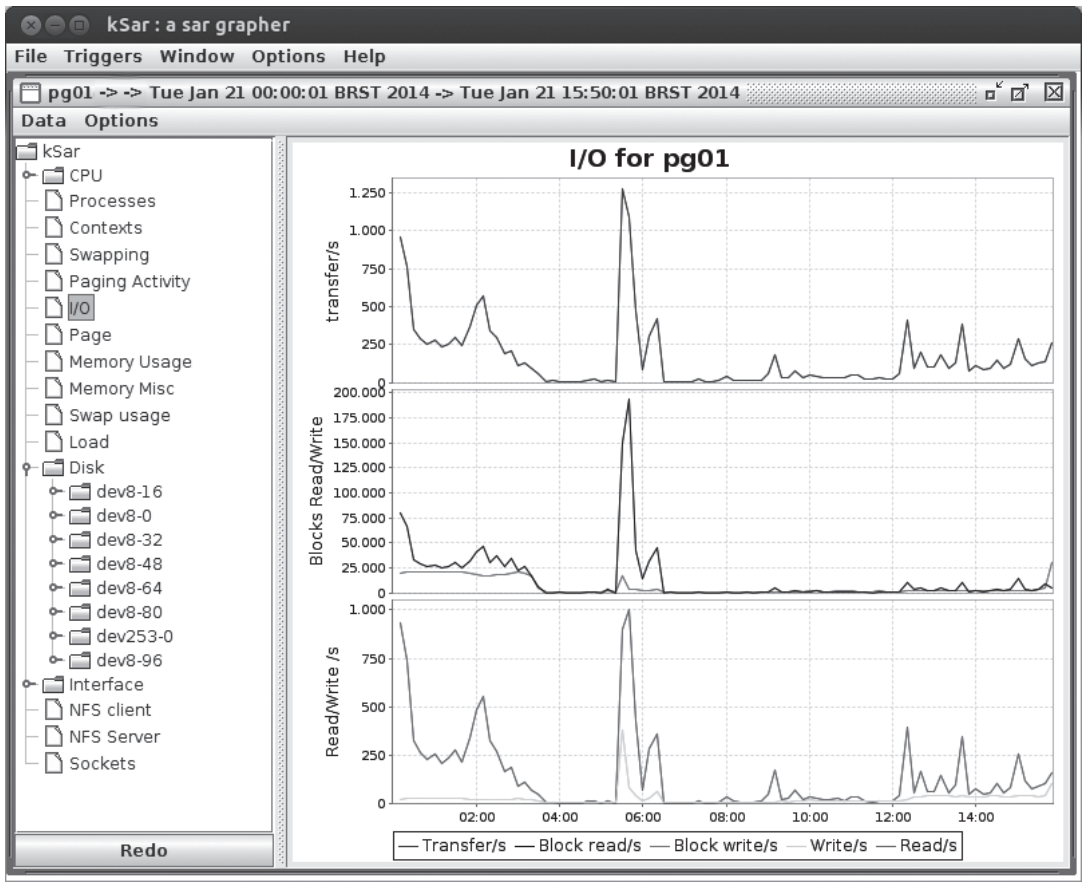
```
12:05:43 AM       DEV       tps  rd_sec/s  wr_sec/s  avgrq-sz  avgqu-sz     await     svctm     %util
12:05:44 AM     dev8-0     78.00      0.00  79832.00   1023.49      2.91     41.33      9.44     73.60
12:05:44 AM    dev11-0      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00

12:05:44 AM       DEV       tps  rd_sec/s  wr_sec/s  avgrq-sz  avgqu-sz     await     svctm     %util
12:05:45 AM     dev8-0    104.26      8.51 103872.34    996.41     45.50    428.73      9.96    103.83
12:05:45 AM    dev11-0      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00

12:05:45 AM       DEV       tps  rd_sec/s  wr_sec/s  avgrq-sz  avgqu-sz     await     svctm     %util
12:05:46 AM     dev8-0     21.65      0.00  17550.52    810.67      3.01    180.57     13.71     29.69
12:05:46 AM    dev11-0      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
```

O sar é outra ferramenta extremamente versátil instalada junto com o pacote sysstat. Ela pode reportar dados de CPU, memória, paginação, swap, I/O, huge pages, rede e mais.

Figura 5.7
Exemplo de saída do sar -d, informações de I/O.

Para utilizá-la, pode ser necessário ativar a coleta, normalmente no arquivo de configuração /etc/default/sysstat nos sistemas Debian/Ubuntu ou pela configuração da Cron nos sistemas Red Hat/CentOS em /etc/cron.d/sysstat.

Com a coleta de estatísticas do sistema funcionando, é possível utilizar o KSar para gerar gráficos sob demanda. O KSar é uma aplicação open source em Java muito simples de usar. Uma vez acionado o KSar, basta fornecer as informações para conexão com o host que deseja analisar. A conexão é estabelecida via SSH e recupera os dados coletados com o comando *sar*, plotando os gráficos de todos os recursos coletados.



Monitorando o PostgreSQL

As principais ferramentas que permitem o monitoramento do PostgreSQL são:

- pg_activity.
- pgAdmin III.
- Nagios.
- Cacti.
- Zabbix.

pg_activity

Uma excelente ferramenta para analisar processos, específica para o PostgreSQL, é a pg_activity. É uma ferramenta open source semelhante ao top como monitoramento de processos, mas cruzando informações de recursos do Sistema Operacional com dados do banco sobre o processo, como qual a query em execução, há quanto tempo, se está bloqueada por outros processos e outras informações.

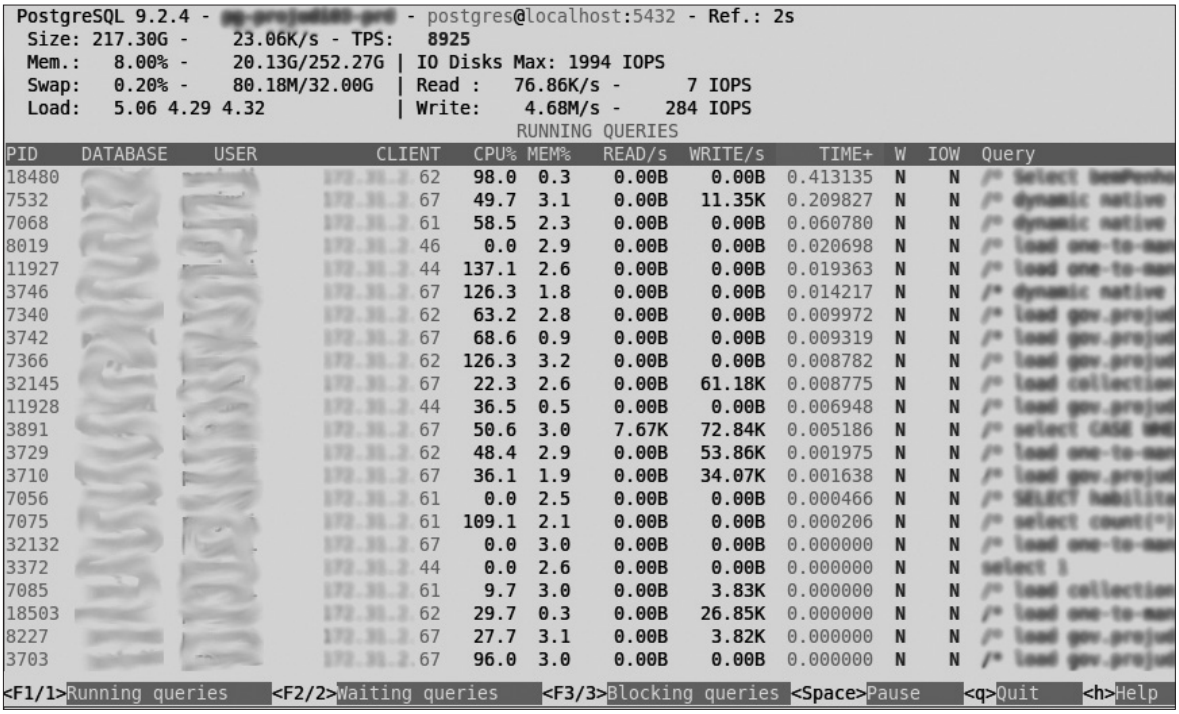
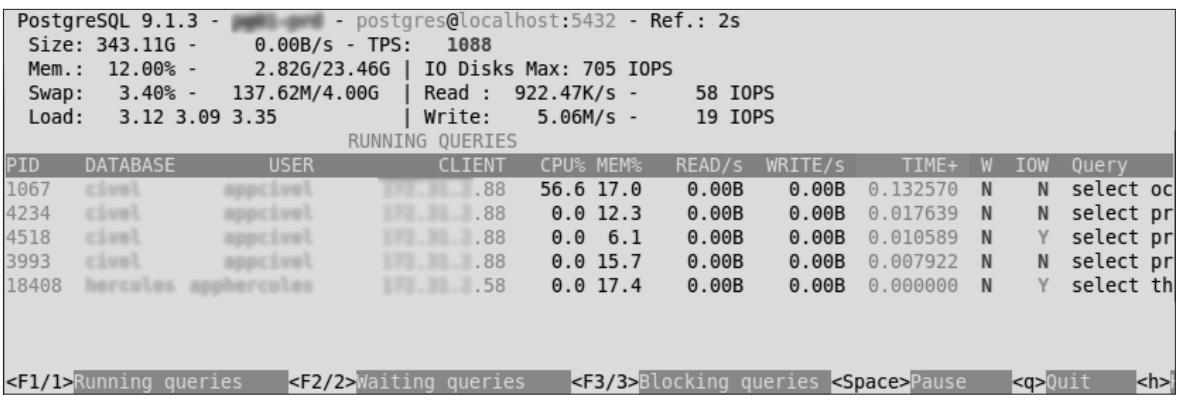


Figura 5.9
pg_activity, umas das melhores ferramentas para PostgreSQL.

A maior vantagem do pg_activity sobre ferramentas genéricas de monitoramento de processos é que ela considera o tempo em execução da query e não do processo em si. Isso é muito mais útil e realista para o administrador de Banco de Dados.

Uma característica interessante dessa ferramenta é o fato de ela apresentar cada processo normalmente na cor verde. Se a query está em execução há mais de 0,5s, ela é exibida em amarelo, e em vermelho se estiver em execução há mais de 1s. Isso torna muito fácil para o administrador enxergar algo fora do normal e possibilita tomar decisões mais rapidamente.

Exibe também qual a base, usuário, carga de leitura(READ/s) e escrita(WRITE/s) em disco. Indica se o processo está bloqueado por outro (W) ou se o processo está bloqueado aguardando operações de disco (IOW).



É possível listar somente os processos que estão bloqueando (Blocking Queries). Para tanto, basta pressionar “F3” ou “3”. O mesmo vale para os processos que estão sendo bloqueados (Waiting Queries), filtrados através da tecla “F2” ou “2”.

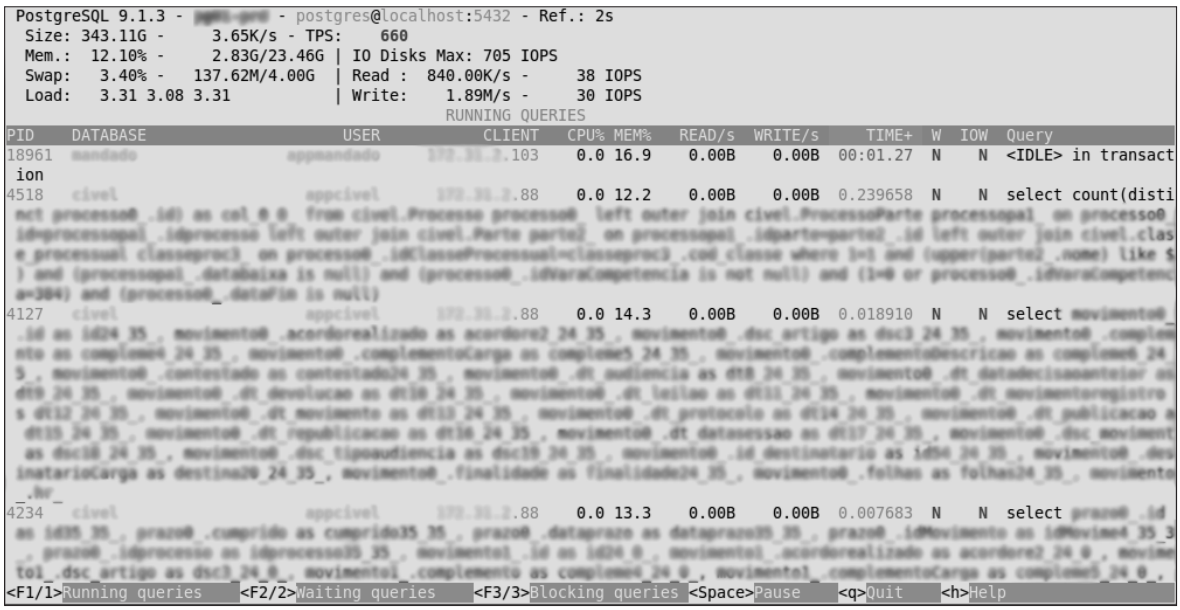
Figura 5.10
Processos no pg_activity aguardando operações em disco, coluna IOW.

Acima dos processos, nos dados gerais no topo, é exibido o Transactions Per Second (TPS), que são operações no banco por segundo. Essa informação é uma boa métrica para acompanharmos o tamanho do ambiente. Ela indica qualquer comando disparado contra o banco, incluindo selects.

Também são exibidos dados gerais de I/O, com dados de leitura e escrita tanto em MB por segundo (throughput) quanto em operações por segundo.

Figura 5.11
pg_activity: pressionando v alterna modo de exibição da query.

Dependendo do que se deseja analisar, pode-se querer suprimir o texto da query para mostrar mais processos na tela. É possível alternar entre a exibição da query completa, identada ou apenas um pedaço inicial pressionando a tecla "v".



pgAdmin III

A ferramenta gráfica mais utilizada com o PostgreSQL, a pgAdmin III, também possui facilidades para o seu monitoramento.

Depois de conectado a um servidor, acesse o menu "Tools" e a opção "Server Status". Será aberta uma nova janela, por padrão com quatro painéis. Um deles exibe a log do PostgreSQL (caso tenham sido instaladas no servidor algumas funções utilitárias). Os outros três são Activity, Locks e Prepared Transactions.

Activity lista todos os processos existentes, com todas as informações relevantes como base, usuários, hora de início de execução da query, a query em si, estado e outras informações.

Quando o processo está bloqueado, ele é mostrado em vermelho. Se está demorando mais de 10s, é exibido em laranja.

É possível cancelar queries e matar processos pelo painel Server Status.

Figura 5.12
Server Status no pgAdmin III.

O painel Locks exibe todos os locks existentes e informações como o processo, o modo de lock, se foi fornecido ou não e a relação (tabela ou índice) cujo lock se refere.

Prepared Transactions são transações utilizadas em transações distribuídas e seu uso é mais restrito.

Um detalhe importante sobre o Server Status do pgAdmin é o tempo de atualização das informações. O tempo é escolhido em uma combo box na parte superior da janela, e por padrão é de 10s. Não deve-se deixar um tempo muito baixo, pois as consultas ao catálogo feitas pelo pgAdmin são complexas, em especial a consulta a view de locks, que cria ela própria novos locks.

Nagios

O Nagios é open source e uma das ferramentas mais usadas para monitoramento de serviços e infraestrutura de TI. É possível monitorar praticamente tudo com ele, desde equipamentos de rede passando por disponibilidade de servidores até número de scans em tabelas do banco, seja pelo protocolo padrão para gerenciamento SNMP seja por scripts específicos. O fato de ser tão flexível e poderoso traz, como consequência, o fato de não ser tão simples configurar o Nagios inicialmente.

O nagios trabalha basicamente alertando quando um indicador passa de determinado limite. É possível ter dois limites:

- **warning**: normalmente é representado em amarelo na interface;
- **critical**: normalmente em vermelho.

Quando alcançados esses limites, o Nagios pode enviar e-mails ou um SMS de alerta. Mais do que isso, o Nagios pode executar ações mais complexas quando limites forem alcançados, como por exemplo, executar um script de backup quando detectado que um erro ocorreu ou que o último backup é muito antigo.

Apesar de muitas dessas ações ainda requererem que se escreva scripts, o Nagios ajuda em muito a centralização do monitoramento. Mas seu uso pode exigir também a instalação de um agente nos servidores monitorados.

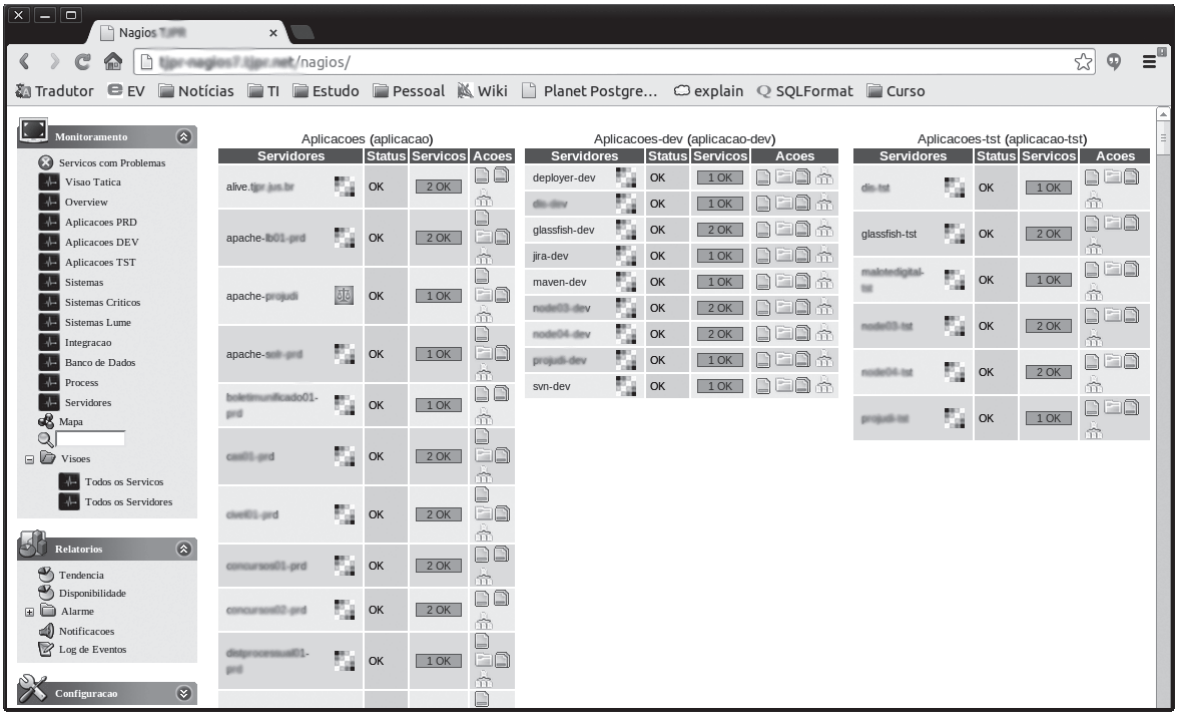


Figura 5.13 Nagios: monitoramento e alertas de infraestrutura.

Podemos utilizar o Nagios para monitorar dados internos do PostgreSQL através de plugins, sendo o mais conhecido deles para o PostgreSQL o check_postgres. Alguns exemplos de acompanhamentos permitidos pelo check_postgres são tabelas ou índices bloated ("inchados"), tempo de execução de queries, número de arquivos de WAL, taxa de acerto no cache ou diferença de atualização das réplicas.

Cacti

O Cacti é uma ferramenta com uma filosofia diferente. É uma ferramenta para geração de gráficos e não de emissão de alertas. Sua utilidade está em auxiliar na análise de dados históricos para diagnóstico de problemas ou para identificação dos padrões normais de uso dos recursos. Outra utilidade é para planejamento de crescimento com base na análise, por exemplo, do histórico de percentual de uso de CPU ou uso de espaço em disco.

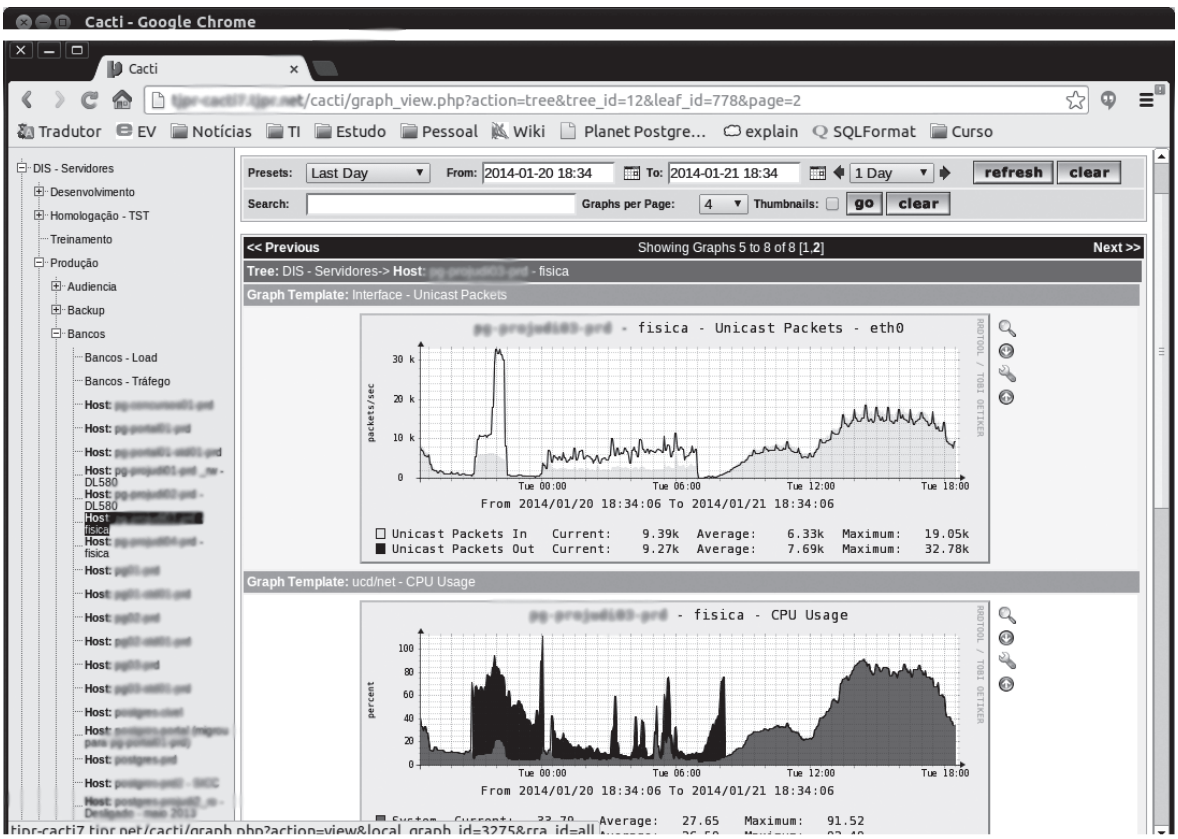


Figura 5.14
Cacti: gráficos e dados históricos.

Zabbix

Outra ferramenta open source de monitoramento é o Zabbix, que une funcionalidades de alertas do Nagios e plotagem de gráficos do Cacti. O Zabbix é mais flexível que o Nagios no gerenciamento de alertas, possibilitando ter mais níveis de estado para um alerta do que somente warning e critical.

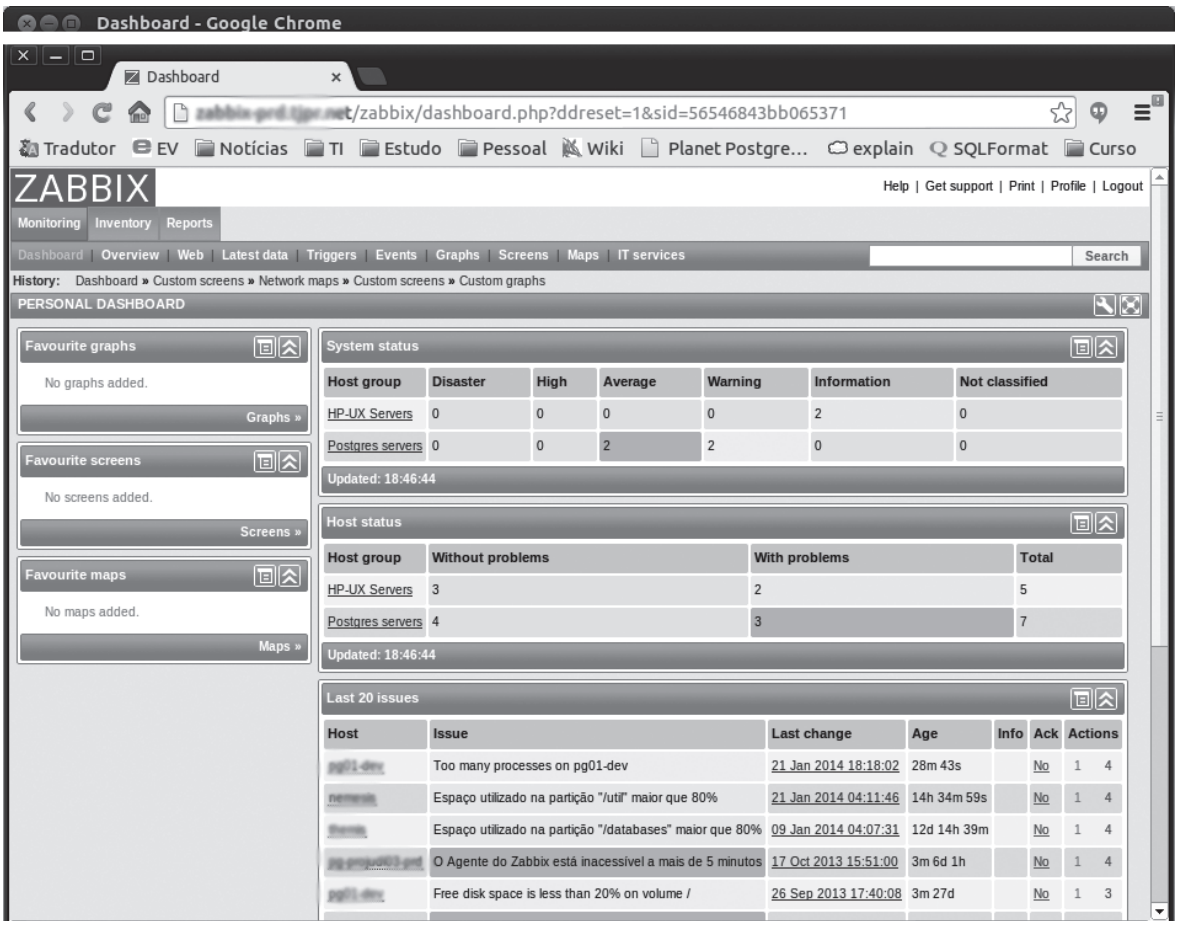


Figura 5.15
Zabbix: uma junção de Nagios e Cacti.

Monitorando o PostgreSQL pelo catálogo

Na sessão de aprendizagem 3 foi visto o Catálogo de Sistema do PostgreSQL e foram apresentadas as principais tabelas e visões contendo os metadados do banco. Além disso, foi comentada a existência das visões estatísticas, que contêm informações de acesso aos objetos.

Dentre estas, destacamos:

- pg_stat_activity;
- pg_locks;
- pg_stat_database;
- pg_stat_user_tables.

Agora vamos ver alguns exemplos de como usá-los para a tarefa de monitoramento do PostgreSQL. Porém, antes veremos mais duas visões dinâmicas muito úteis, que contêm informações sobre os processos em execução no momento: pg_stat_activity e pg_locks.

pg_stat_activity

A pg_stat_activity é considerada extremamente útil, por exibir uma fotografia do que os usuários estão executando em um determinado momento. Essa view fornece algumas vantagens sobre o uso de ferramentas como o pg_activity. Primeiro, ela contém mais informações, como a hora de início da transação. Uma segunda vantagem é que podemos manipulá-la com SELECT para listarmos apenas o que desejamos analisar, como por exemplo, apenas os processos de determinado usuário ou base de dados, ou que a query contenha determinado nome de tabela, ou ainda selecionar apenas aquelas em estado IDLE IN TRANSACTION. O uso dessa view fornece uma alternativa ágil e poderosa para o administrador PostgreSQL monitorar a atividade no banco.

Datid	ID da base de dados.
datname	Nome da base de dados.
Pid	ID do processo.
usesysid	ID do usuário.
usename	Login do usuário.
application_name	Nome da aplicação.
client_addr	IP do cliente. Se for nulo, é local ou processo utilitário como o vacum.
client_hostname	Nome da máquina cliente.
client_port	Porta TCP do cliente.
backend_start	Hora de início do processo. Pouco útil quando usado com pools.
xact_start	Hora de início da transação. Null se não há transação ativa.
query_start	Hora de início de execução da query atual OU início de execução da última query se state for diferente de ACTIVE.
state_change	Hora da última mudança de estado.
waiting	Se a query está bloqueada aguardando um lock.
State	active: a query está em execução no momento. idle: não há query em execução. idle in transaction: há uma transação aberta, mas sem query executando no momento. idle in transaction(aborted): igual a idle in transaction mas alguma query causou um erro.
query	Query atual ou a última, caso state for diferente de active.

Tabela 5.1
Colunas da pg_stat_activity.

Um exemplo de uso da pg_stat_activity buscando listar todos os processos da base curso que estão bloqueados há mais de 1h:

```
postgres=#  SELECT pid, usename, query_start
            FROM pg_stat_activity
            WHERE datname='curso'
            AND waiting
            AND (state_change + interval '1 hour') < now();
```

Outra possibilidade: matar todos os processos que estão rodando há mais de 1h, mas não estão bloqueados, ou seja, simplesmente estão demorando demais:

```
postgres=#  SELECT pg_terminate_backend(pid)
            FROM pg_stat_activity
            WHERE datname='curso'
            AND NOT waiting
            AND (query_start + interval '1 hour') < now();
```

pg_locks

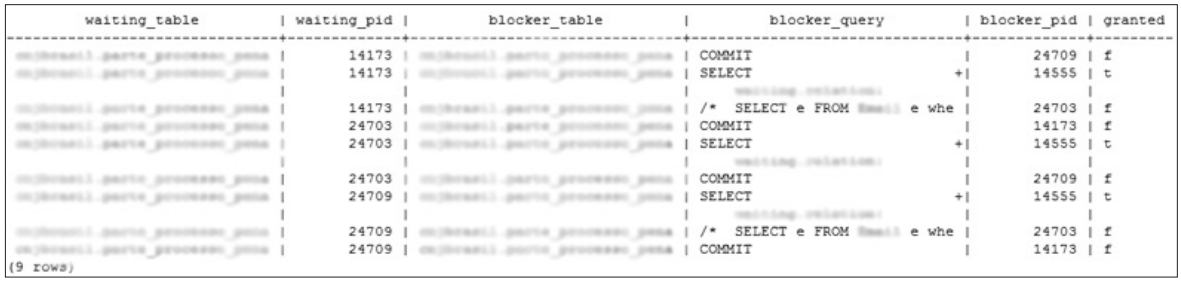
A visão pg_locks contém informações dos locks mantidos por transações, explícitas ou implícitas, abertas no servidor.

Locktype	Tipo de objeto alvo do lock. Por exemplo: relation(tabela), tuple(registro), transactionid (transação)
Database	Base de dados
Relation	Relação (Tabela/Índice/Sequence...) alvo do lock, se aplicável
Transactionid	ID da transação alvo. Caso o lock seja para aguardar uma transação.
Pid	Processo solicitante do lock
Mode	Modo de lock. Por exemplo: AccessShareLock(SELECT), ExclusiveLock, RowExclusiveLock
Granted	Indica se o lock foi adquirido ou está sendo aguardado

Tabela 5.2
Principais colunas da pg_locks.

Em algumas situações mais complexas, pode ser necessário depurar o conteúdo da tabela pg_locks para identificar quem é o processo raiz que gerou uma cascata de locks. A seguinte query usa pg_locks e pg_stat_activity para listar processos aguardando locks de outros processos:

```
postgres =#  SELECT
                waiting.relation::regclass    AS waiting_table,
                waiting_stm.query             AS waiting_query,
                waiting.pid                   AS waiting_pid,
                blocker.relation::regclass    AS blocker_table,
                blocker_stm.query     AS blocker_query,
                blocker.pid                AS blocker_pid,
                blocker.granted            AS blocker_granted
             FROM    pg_locks AS waiting,
                     pg_locks AS blocker,
                     pg_stat_activity AS waiting_stm,
                     pg_stat_activity AS blocker_stm
             WHERE waiting_stm.pid = waiting.pid
               AND ((waiting."database" = blocker."database"
                     AND waiting.relation  = blocker.relation)
                     OR waiting.transactionid = blocker.transactionid)
               AND blocker_stm.pid = blocker.pid
               AND NOT waiting.granted
               AND waiting.pid <> blocker.pid
             ORDER BY waiting_pid;
```



waiting_table	waiting_pid	blocker_table	blocker_query	blocker_pid	granted
[illegible]	14173	[illegible]	COMMIT	24709	f
[illegible]	14173	[illegible]	SELECT +|	14555	t
			[illegible]		
[illegible]	14173	[illegible]	/* SELECT e FROM [illegible] e whe	24703	f
[illegible]	24703	[illegible]	COMMIT	14173	f
[illegible]	24703	[illegible]	SELECT +|	14555	t
			[illegible]		
[illegible]	24703	[illegible]	COMMIT	24709	f
[illegible]	24709	[illegible]	SELECT +|	14555	t
			[illegible]		
[illegible]	24709	[illegible]	/* SELECT e FROM [illegible] e whe	24703	f
[illegible]	24709	[illegible]	COMMIT	14173	f

(9 rows)

Figura 5.16
Dependência entre processos causadas por Locks.

Outras visões (views) úteis

Apresentamos a seguir um conjunto de queries SQL que ajudam no monitoramento do banco. O primeiro exemplo consulta a view pg_stat_database e gera, como resultado, o número de transações total, o transaction per second (TPS) médio e o percentual de acerto no cache para cada base do servidor:

```
postgres=#   SELECT datname,
(xact_rollback + xact_commit) as total_transacoes,
((xact_rollback + xact_commit)
/ EXTRACT(EPOCH FROM (now() - stats_reset))) as tps,
CASE WHEN blks_hit = 0 THEN 0
     ELSE (( blks_hit / (blks_read + blks_hit)::float) * 100)
END as cache_hit
  FROM pg_stat_database
 WHERE datname NOT LIKE 'template_';
```

Nessa consulta a view pg_stat_user_tables o resultado obtido traz todas as tabelas e calcula o percentual de index scan em cada uma:

```
curso=# SELECT
        schemaname,
            relname,
            seq_scan,
            idx_scan,
            ((idx_scan::float / (idx_scan + seq_scan)) * 100) as percentual_idxscan
          FROM pg_stat_user_tables
         WHERE (idx_scan + seq_scan) > 0
         ORDER BY percentual_idxscan;
```

Para listar todas as bases e seus respectivos tamanhos, ordenado da maior para menor:

```
postgres=#   SELECT pg_database.datname,
             pg_size_pretty(pg_database_size(datname)) AS tamanho
             FROM pg_database
             ORDER BY 1;
```

Um exemplo que lista os objetos, tabelas e índices que contém mais dados no Shared Buffer, ou seja, que estão tirando maior proveito do cache, é:

```
postgres=#   SELECT n.nspname || '.' || c.relname as objeto,
             pg_size_pretty(count(*) * 8192) as tamanho
             FROM pg_class c
             INNER JOIN pg_buffercache b ON b.relfilenode = c.relfilenode
             INNER JOIN pg_database d ON b.reldatabase = d.oid
             AND d.datname = current_database()
             INNER JOIN pg_namespace n ON c.relnamespace = n.oid
             GROUP BY n.nspname || '.' || c.relname
             ORDER BY 2 DESC;
```

Finalmente, listar todos os índices por tabela, com quantidade de scans nos índices, ajuda a decidir a relevância e utilidade dos índices existentes. Essa informação pode ser obtida através da seguinte consulta:

```
postgres=# SELECT r.relname as tabela,
           c.relname as indice,
           idx_scan as qtd_leituras
           FROM pg_stat_user_indexes i
           JOIN pg_class r ON i.relid=r.oid
           JOIN pg_class c ON i.indexrelid=c.oid
           JOIN pg_namespace nsp ON r.relnamespace=nsp.oid
           WHERE nspname NOT LIKE 'pg_%'
ORDER BY 1,2 DESC;
```

Monitorando espaço em disco

Normalmente monitoramos o espaço nos discos por ferramentas do Sistema Operacional, como o df. Porém, o PostgreSQL fornece algumas funções úteis para consultarmos o consumo de espaço por tabelas, índices e bases inteiras.

Tabela 5.3 Funções para consulta de consumo de espaço.

pg_database_size(nome)	Tamanho da base de dados.
pg_relation_size(nome)	Tamanho somente da tabela, sem índices e toasts.
pg_table_size(nome)	Tamanho de tabela e toasts, sem índices.
pg_indexes_size(nome)	Tamanho dos índices de uma tabela.
pg_tablespace_size(nome)	Tamanho de um tablespace.
pg_total_relation_size(nome)	Tamanho total, incluindo tabela, índices e toasts.
pg_size_pretty(bigint)	Converte de bytes para formato legível (MB,GB,TB etc.).

A seguinte consulta mostra os tamanhos da tabela, índices, tabela e toasts, além de tamanho total para todas as tabelas ordenadas pelo tamanho total decrescente, destacando no início as maiores tabelas:

```
curso=# SELECT  schemaname || '.' || relname as tabela,
                pg_size_pretty(pg_relation_size(schemaname || '.' || relname)) as t
am_tabela,
                pg_size_pretty(pg_table_size(schemaname || '.' || relname)) as tam_
tabela_toast,
                pg_size_pretty(pg_indexes_size(schemaname || '.' || relname)) as ta
m_indices,
                pg_size_pretty(pg_total_relation_size(schemaname || '.' || relname)
) as tam_total_tabela
                FROM pg_stat_user_tables
                ORDER BY pg_total_relation_size(schemaname || '.' || relname) DESC;
```

Existem diversas outras visões com informações estatísticas sobre índices, funções, sequences, dados de I/O e mais.

Configurando o Log do PostgreSQL para monitoramento de Queries

- log_destination.
- log_line_prefix.
- log_filename.
- log_rotation_age.
- log_rotation_size.
- log_min_duration_statement.
- log_statement.

O log do PostgreSQL é bastante flexível e possui uma série de recursos configuráveis. Vamos ver alguns parâmetros de configuração que permitirão registrar queries e eventos que ajudam a monitorar a saúde do banco.

Já vimos na sessão 2, em configuração do PostgreSQL, que podemos ligar a coleta da log em arquivos com o parâmetro logging_collector. Os arquivos serão criados por padrão no diretório "pg_log" sob o PGDATA.

Outros parâmetros que devem ser considerados:

- **log_destination:** indica onde a log será gerada. Por padrão para a saída de erro stderr. Para armazenar os arquivos com logging_collector, esse valor deve ser stderr ou csvlog. Com o valor syslog, pode-se também usar o recurso de log do Sistema Operacional, porém alguns tipos de mensagens não são registradas nesse modo;
- **log_line_prefix**: é o formato de um prefixo para cada linha a ser registrada. Existem diversas informações que podem ser adicionadas a esse prefixo, como a hora (%t), o usuário (%u) e o id do processo(%p). Porém, para usarmos ferramentas de relatórios de queries, como pgFouine e pgBadger, devemos utilizar alguns padrões nesse prefixo. Um padrão comumente utilizado é:

```
log_line_prefix = '%t [%p]: [%l-1] user=%u,db=%d '
```

Note que há um espaço em branco ao final, que deve ser mantido. Esse formato produzirá no log saídas como o exemplo a seguir.

```
2014-01-22 10:56:57 BRST [9761]: [16-1] user=aluno,db=postgres LOG:  duration:
1.527 ms  statement: SELECT pid, usename, query_start FROM pg_stat_activity WHERE
datname='curso' AND waiting AND (state_change + interval '1 hour') < now();
```

- **log_filename**: define o formato do nome de arquivo. O valor padrão inclui data e hora da criação do arquivo. Esse formato, além de identificar o arquivo no tempo, impede que este seja sobrescrito.

```
log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'
```

Exemplo de nome de arquivo:

```
postgresql-2014-01-22_000000.log
```

- **log_rotation_age:** define o intervalo de tempo no qual o arquivo de log será rotacionado. O padrão é de um dia;
- **log_rotation_size:** define o tamanho máximo a partir do qual o arquivo de log será rotacionado;
  - Ao fazer a rotação do log, se houver um padrão de nome para os arquivos de log que garanta variação, novos arquivos serão gerados (formato definido em log_filename). Se o formato para nomear arquivos for fixo, os arquivos previamente existentes serão sobrescritos. Isso vai ocorrer em função do limite estabelecido para o tamanho ou idade do log, não importando o que ocorrer primeiro. Esse processo de rotação também acontece a cada restart do PostgreSQL.
- **log_min_duration_statement:** indica um valor em milissegundos acima do qual serão registradas todas as queries cuja duração for maior do que tal valor. O valor padrão é -1, indicando que nada deve ser registrado. O valor 0 registra todas as queries independentemente do tempo de duração de cada uma delas. Toda query registrada terá sua duração também informada no log;
- **log_statement:** indica quais tipos de queries devem ser registradas. O valor padrão none não registra nada. Os valores possíveis são:
  - **ddl**: comandos DDL, de criação/alteração/exclusão de objetos do banco;
  - **mod**: comandos DDL mais qualquer comando que modifique dados;
  - **all**: registra todas as queries.

A definição do que deve ser registrado no log depende de diversos fatores, tais como a natureza dos sistemas envolvidos, as políticas de auditoria ou a necessidade de depuração de problemas. Uma boa prática pode ser usar log_statement = mod, para registrar qualquer alteração feita no banco para finalidade de auditoria e definir log_min_duration_statement para um valor que capture queries com problemas, por exemplo, 3000 (3 segundos).

Existem diversos outros parâmetros que definem o que deve ser registrado no log, como conexões, desconexões, arquivos temporários, ocorrências de checkpoints e espera por locks. Todas essas informações podem ajudar na resolução de problemas. Por outro lado, junto com as queries registradas, isso causa sobrecarga no sistema, já que há um custo para registrar todos esses dados. O administrador deve encontrar um meio termo entre o que é registrado e o impacto da coleta e gravação dessas informações.

Uma boa opção é gravar as logs em um disco separado: pode ser feito mudando o parâmetro log_directory ou usando links simbólicos e registrar apenas o que é crucial por padrão. Quando houver alguma situação especial, aí então ligar temporariamente o registro das informações pertinentes para avaliar tal situação.

Geração de relatórios com base no log

Ferramentas de Relatórios de Logs - pgBadger

- Parser da log para gerar relatórios HTML
- Escrito em perl
- Rankings de Queries
  - Queries mais Lentas
  - Queries mais tomaram tempo total (duração x nr execuções)
  - Queries mais frequentes
  - Query normalizada e exemplos
  - Tempo Total, número de Execuções e Tempo Médio
- Mais rápido, mais atualizado, mais funcionalidades que o pgFouine

O pgBadger é um analisador de logs do PostgreSQL, escrito em perl e disponibilizado como open source, fazendo o parser dos arquivos de log e gerando relatórios html. Ele possui muitas semelhanças com o pgFouine, outra ferramenta de análise de log, mas é mais rápido, mais completo e tem uma comunidade mais atuante que publica novas versões com mais frequência.

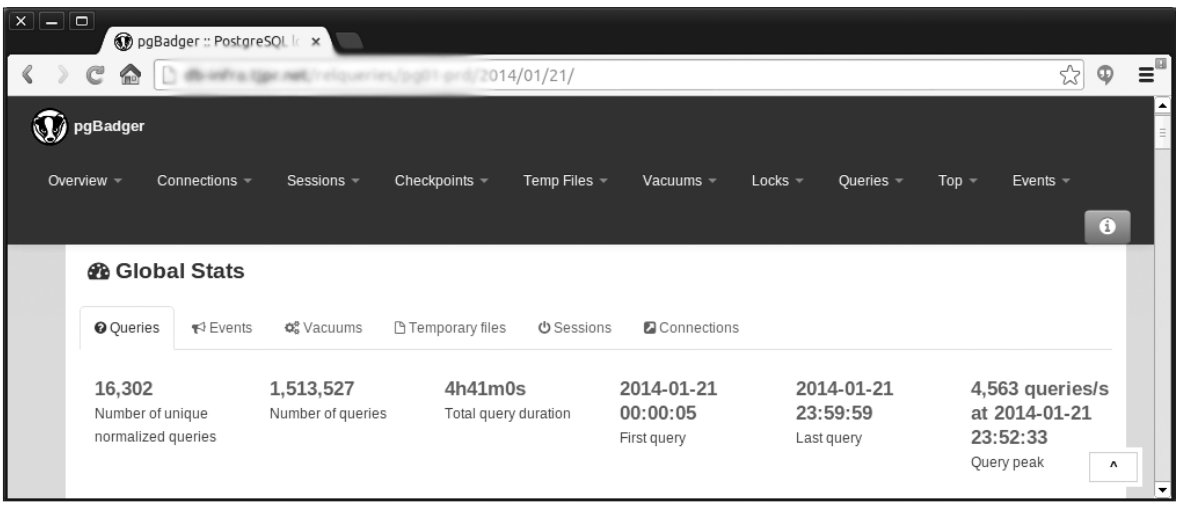


Figura 5.17
pgBadger: diversas informações coletadas no PostgreSQL.

Os relatórios exibem seções com rankings, como as queries mais lentas, queries que tomaram mais tempo e queries mais frequentes.

A seção mais útil é a que mostra as queries que tomaram mais tempo no total de suas execuções – seção “Queries that took up most time (N)” –, já que ao analisar uma query não devemos considerar somente seu tempo de execução, mas também a quantidade de vezes em que esta é executada. Uma query que tenha duração de 10 minutos uma vez ao dia consome menos recursos do que uma query que dure 5s, mas é executada mil vezes ao dia. Nessa seção é exibido o tempo total tomado por todas as execuções da query, o número de vezes e o tempo médio. A query normalizada é mostrada, junto com três exemplos com valores.

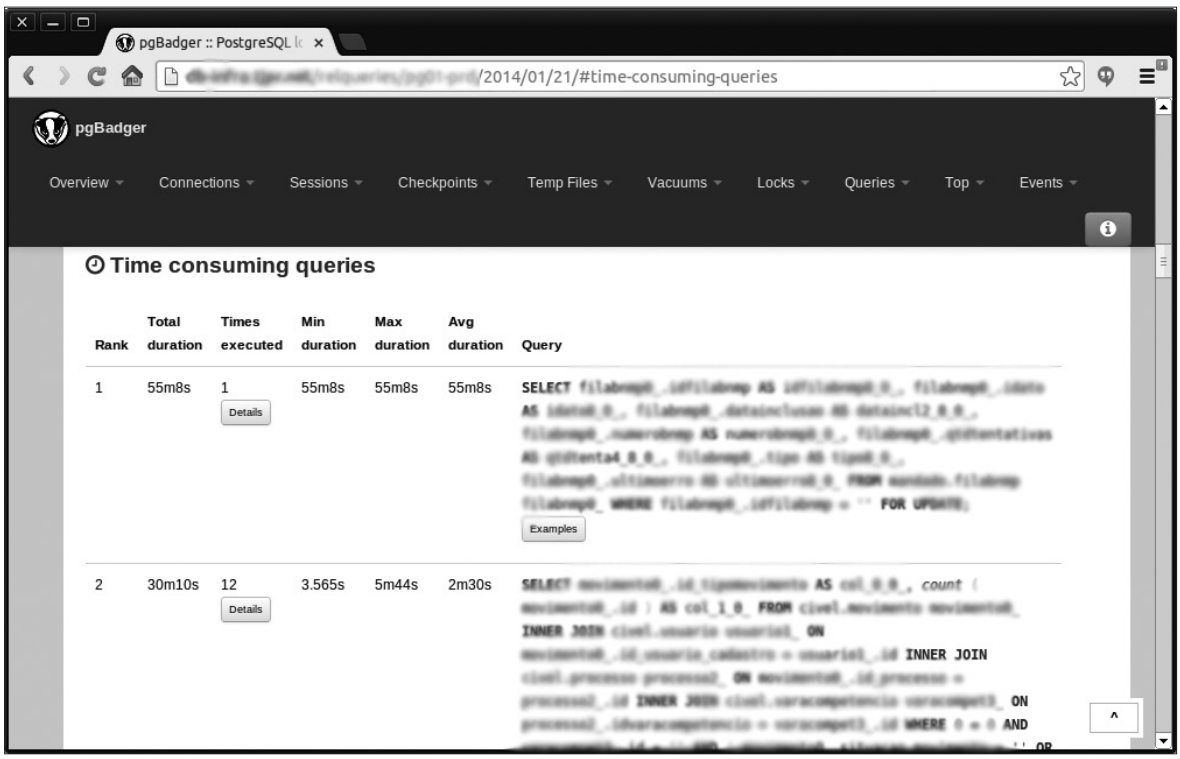


Figura 5.18 Seção 'Time consuming queries' do pgBadger, mostrando queries lentas.

Uma das funcionalidades mais interessantes do pgBadger é a geração de gráficos. São gerados gráficos de linhas para queries por segundo, de colunas e de pizza para número de conexões por usuário. É possível inclusive fazer zoom nesses gráficos e salvá-los como imagens, isso sem a necessidade de bibliotecas especiais no perl ou plugins no navegador.

Para usar o pgBadger é necessário configurar as opções de log do PostgreSQL como vimos anteriormente, para que as linhas nos arquivos tenham um determinado prefixo necessário para o processamento correto dos dados.

Depois de fazer o download do arquivo compactado (ver link no AVA), siga estes passos:

```
$   cd /usr/local/src/
$   sudo tar xzf pgbadger-4.1.tar.gz
$   cd pgbadger-4.1/
$   perl Makefile.PL
$   make
$   sudo make install
```

O pgBadger será instalado em /usr/local/bin e já deve estar no seu PATH. Assim, basta executar o pgBadger passando os arquivos de log que deseja processar como parâmetro:

```
$    pgbadger -f stderr /db/data/pg_log/*.log -o relatorio.html
```

Esse exemplo lerá todos os arquivos com extensão .log no diretório pg_log e vai gerar um arquivo de saída chamado *relatorio.html*. Você pode informar um arquivo de log, uma lista de arquivos ou até um arquivo compactado contendo um ou mais arquivos de log.

Extensão pg_stat_statements

- Extensão do PostgreSQL para capturar queries em tempo real
- Cria uma visão pg_stat_staments contendo:
  - queries mais executadas
  - número de execuções
  - tempo total
  - quantidade de registros envolvidos
  - volume de dados (através de números de blocos processados)

Na seção de aprendizagem 1 foi ilustrado como instalar uma extensão do PostgreSQL, sugerindo a instalação da pg_stat_statements. Essa extensão cria uma visão que contém as queries mais executadas e dados dessas queries, como o número de execuções, o tempo total, a quantidade de registros envolvidos e volume de dados através de números de blocos processados.

Uma boa prática é copiar, uma vez ao dia, os arquivos de log do dia anterior. Esses arquivos podem ser compactados e transferidos para uma área específica, fazendo o agendamento para que o pgBadger processo todos os arquivos colocados nessa área. Assim, diariamente pode-se ter o relatório do que aconteceu no dia anterior para análise.

A vantagem da pg_stat_statements é ter as informações em tempo real, sem precisar processar arquivos de log para encontrar as top queries.

Além do formato geral de instalação mostrado anteriormente, essa extensão precisa de algumas configurações no *postgresql.conf*, tais como configurar o carregamento de uma biblioteca, demandando um restart do banco para começar a funcionar.

Outro parâmetro que precisa ser configurado é a quantidade de queries que deve ser mantida pela view, através de pg_stat_statements.max, cujo valor padrão é 1000. O parâmetro pg_stat_statements.track indica se deve-se registrar todas as queries (all) e não considerar queries internas à funções (top) ou nenhum registro (none).

Exemplo de configuração no *postgresql.conf*:

```
shared_preload_libraries = 'pg_stat_statements'
pg_stat_statements.track = all
pg_stat_statements.max = 5000
```

Depois de reiniciar o PostgreSQL e criar a extensão em alguma base, a view já pode ser acessada.

```
postgres=# SELECT    query, calls, total_time,
    shared_blks_hit / nullif(shared_blks_hit + shared_blks_read, 0) AS hit
                     FROM pg_stat_statements
             ORDER BY total_time
    DESC LIMIT 10;
```

```
-[ RECORD 1 ]----------------------------------------------------------
query      | SELECT EXTRACT(? FROM NOW()) AS timestamp, SUM(pg_stat_get_
calls      | 1902
total_time | 17214806.98
hit        | 1
-[ RECORD 2 ]----------------------------------------------------------
query      | VACUUM pgbench_accounts;
calls      | 1
total_time | 794278.508
hit        | 0
-[ RECORD 3 ]----------------------------------------------------------
query      | UPDATE pgbench_accounts SET abalance=abalance+? WHERE abala
calls      | 1
total_time | 596450.499
hit        | 0
-[ RECORD 4 ]----------------------------------------------------------
query      | UPDATE pgbench_accounts SET abalance=abalance+? WHERE abala
calls      | 1
total_time | 316310.947
```

Figura 5.19 Queries mais executadas com o pg_stat_ statements.

pgBench

Benchmark: Testes que avaliam o desempenho de um ambiente ou um objeto específico, como um item de hardware ou software. Também é comum usar essas avaliações quando se está fazendo tuning para medir a eficácia de um ajuste em particular.

O pgbench é uma extensão do PostgreSQL usada para fazer testes de **benchmark** e avaliar o desempenho dos recursos e do banco.

O pgBench utiliza testes simples baseados no padrão TPC-B, antigo modelo de teste da Transaction Processing Performance Council (TPC), organização que cria modelos de testes de avaliação para sistemas computacionais baseado em uma taxa de transações, tps, e divulga resultados.

O teste do pgBench é uma transação simples, com updates, inserts e selects executadas diversas vezes, possivelmente simulando diversos clientes e conexões, e no final fornece a taxa de transações em TPS – transações por segundo.

O pgBench possui duas operações básicas:

- Criar e popular uma base;
- Executar queries contra essa base e medir o número de transações.

```
postgres@pg01:~$ pgbench -c 5 -T 30 bench
starting vacuum...end.
transaction type: TPC-B (sort of)
scaling factor: 10
query mode: simple
number of clients: 5
number of threads: 1
duration: 30 s
number of transactions actually processed: 2260
tps = 75.205549 (including connections establishing)
tps = 75.514556 (excluding connections establishing)
postgres@pg01:~$
```

Figura 5.20 Teste com o pgBench.

Resumo

Monitorando pelo Sistema Operacional:

- Usar o top para uma visão geral dos processos. Além das informações gerais, a coluna S (status), com valor D, deve ser observada, além do iowait (wa);
- A vmstat é uma excelente ferramenta para observar o comportamento das métricas, processos esperando disco (b), comportamento da memória e ocorrências de swap que devem ser monitoradas;
- A iostat exibe as estatísticas por device. Analisar o tempo para atendimento de requisições de I/O (await) e a saturação do disco ou canal (%util).

Monitorando pelo PostgreSQL:

- Torne o pg_activity sua ferramenta padrão. Com o tempo você conhecerá quais são as queries normais e as problemáticas que merecem atenção, detectará rapidamente processos bloqueados ou com transações muito longas. Atenção às colunas W (waiting) e IOW (aguardando disco), e aos tempos de execução: amarelo > 0,5 e vermelho > 1s;
- O pgAdmin pode ajudar a detectar quem está bloqueando os outros processos mais facilmente;
- Monitore seus PostgreSQL com o Nagios, Cacti e Zabbix, ou outra ferramenta de alertas, dados históricos e gráficos. Elas são a base para atendimento rápido e planejamento do ambiente;
- Analise as visões estatísticas do catálogo para monitorar a atividade do banco, crie seus scripts para avaliar situações rotineiras;
- Use o pgBadger automatizado para gerar relatórios diários do uso do banco e top queries. Priorize a melhoria das queries em “Time consuming queries”;
- Use a extensão pg_stat_statements para ver as top queries em tempo real;
- pgBench.

6

Manutenção do Banco de Dados

objetivos

Aprender as rotinas essenciais de manutenção do Banco de Dados PostgreSQL, principalmente o Vacuum e suas variações; Conhecer as formas de execução do vacuum manual e automático, o processo de atualização de estatísticas, além dos procedimentos Cluster e Reindex; Elencar os problemas mais comuns relacionados ao Autovacuum e soluções possíveis.

conceitos

Vacuum; Autovacuum; Analyze; Reindex; Bloated Index; Autovacuum Launcher; Autovacuum Worker e Dead Tuples.

Nas sessões anteriores, o termo Vacuum foi mencionado repetidas vezes. Trata-se de procedimento diretamente ligado ao processo de administração do PostgreSQL, que precisa ser realizado com a frequência necessária.

Vacuum

Como já vimos, o PostgreSQL garante o Isolamento entre as transações através do MVCC. Esse mecanismo cria versões dos registros entre as transações, cuja origem são operações de DELETE, UPDATE e ROLLBACK. Essas versões quando não são mais necessárias a nenhuma transação são chamadas dead tuples, e limpá-las é a função do VACUUM.

O vacuum somente marca as áreas como livres, atualizando informações no Free Space Map (FSM), liberando espaço na tabela ou índice para uso futuro. Essa operação não devolverá espaço para o Sistema Operacional, a não ser que as páginas ao final da tabela fiquem vazias.

Vacuum Full

O Vacuum Full historicamente é uma versão mais agressiva do vacum, que deslocará as páginas de dados vazias, como uma desfragmentação, liberando o espaço para o Sistema Operacional. Porém, a partir da versão 9.0 do PostgreSQL ele está um pouco mais leve, mas ainda é uma operação custosa e precisa de um lock exclusivo na tabela. Como regra, deve-se evitar o vacuum full. Ele também nunca é executado pelo autovacuum.

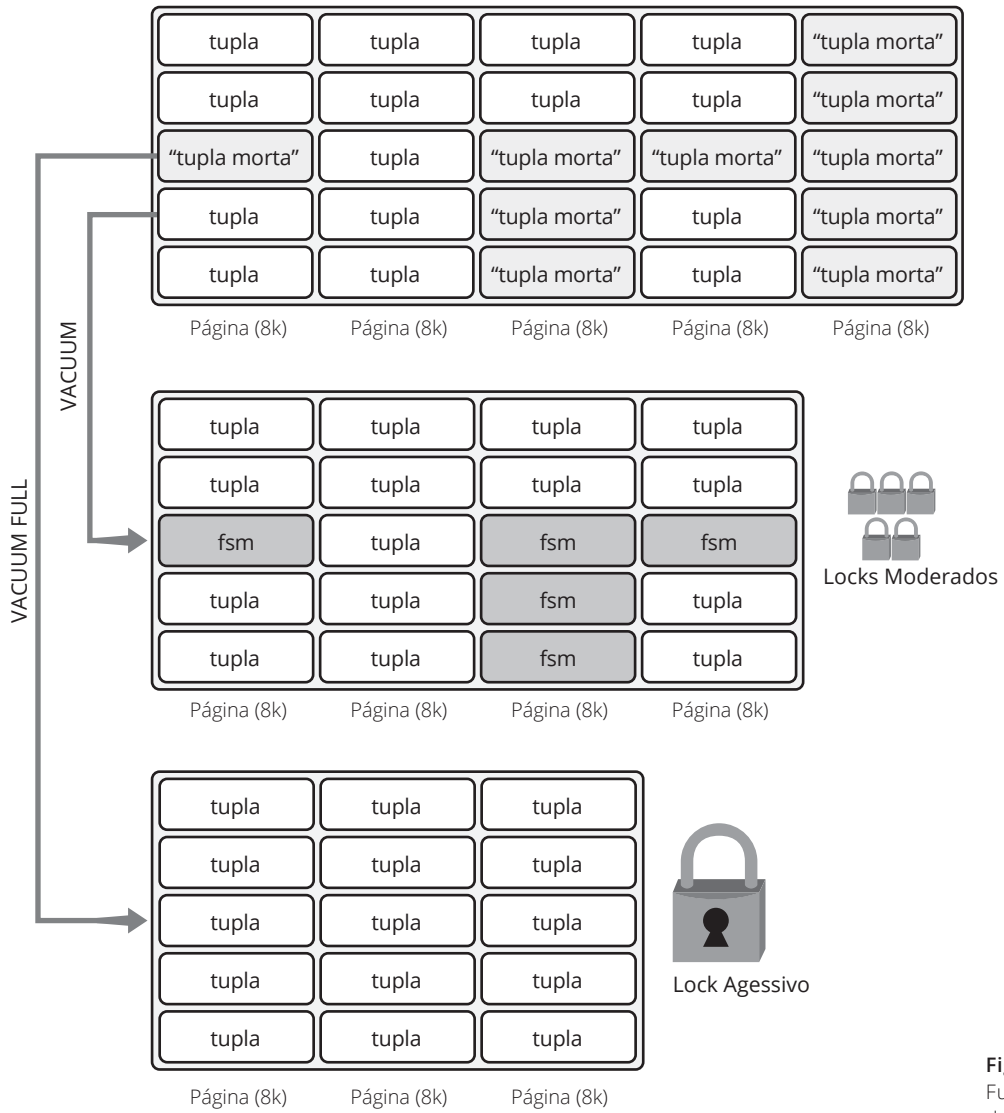


Figura 6.1
Funcionamento do Vacuum.

Executando o Vacuum

A principais opções para o comando *Vacuum* são:

- FULL: executa o vacuum full.
- VERBOSE: exibe uma saída detalhada.
- ANALYZE: executa também atualização das estatísticas.

Para executar o Vacuum manualmente, use o comando *sql VACUUM* ou o utilitário vacuumdb.

O comando a seguir executa o vacuum em todas as tabelas que o usuário possua permissão na base que se está conectado:

```
curso=# VACUUM;
```

É equivalente a:

```
$ vacuumdb -d curso;
```

A seguir são apresentados exemplos de execução do Vacuum com o comando SQL e o utilitário:

Executar um Vacuum Full:

VACUUM FULL;	vacuumdb -f -d curso

Vacuum com Saída Detalhada:

VACUUM VERBOSE;	vacuumdb -v -d curso

Com Atualização de Estatísticas:

VACUUM ANALYZE;	vacuumdb -z -d curso

Todas as Bases do Servidor:

	vacuumdb -a

Uma Tabela Específica:

VACUUM grupos;	vacuumdb -t grupos -d curso

Uma prática comum é o agendamento do Vacuum – por exemplo, na Cron (agendador de tarefas do Linux), para uma vez por dia, normalmente de madrugada. Pode-se executar o vacuum com atualização de estatísticas em todas as bases do servidor através do comando:

```
$ vacuumdb -avz;
```

Isso é equivalente a se conectar com cada base do servidor e executar o comando:

```
curso=# VACUUM ANALYZE VERBOSE;
```

Quando executar o vacuum com o parâmetro VERBOSE, serão geradas várias informações com detalhes sobre o que está sendo feito. A saída será algo como a seguir:

```
bench=# VACUUM ANALYZE VERBOSE contas;
INFO: vacuuming “public.contas”
INFO: scanned index “idx_contas” to remove 999999 row versions
DETAIL: CPU 1.26s/0.14u sec elapsed 1.94 sec.
INFO: “contas”: removed 999999 row versions in 15874 pages
DETAIL: CPU 1.64s/0.13u sec elapsed 8.80 sec.
INFO: index “idx_contas” now contains 1 row versions in 2745 pages
DETAIL: 999999 index row versions were removed.
2730 index pages have been deleted, 0 are currently reusable.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
INFO: “contas”: found 1 removable, 1 nonremovable row versions in 15874 out of 158
74 pages
DETAIL: 0 dead row versions cannot be removed yet.
There were 0 unused item pointers.
0 pages are entirely empty.
CPU 3.84s/0.32u sec elapsed 12.90 sec.
INFO: “contas”: truncated 15874 to 32 pages
DETAIL: CPU 0.63s/0.04u sec elapsed 0.76 sec.
INFO: analyzing “public.contas”
INFO: “contas”: scanned 32 of 32 pages, containing 1 live rows and 0 dead rows; 1
 rows in sample, 1 estimated total rows
VACUUM
```

A falta de vacuum pode causar problemas no acesso a tabelas e índices. Voltaremos a tratar disto nas próximas sessões.

Analyze

Vimos que o comando *VACUUM* possui um parâmetro ANALYZE que faz a atualização das estatísticas da tabela.

Existe também o comando *ANALYZE* somente, que executa apenas o trabalho da coleta das estatísticas sem executar o vacuum. A sintaxe é semelhante:

```
curso=# ANALYZE VERBOSE;
```

Esse comando atualizará as estatísticas de todas as tabelas da base. Pode-se executar para uma tabela específica:

```
curso=# ANALYZE VERBOSE times;
```

Ou mesmo apenas uma coluna ou uma lista de colunas:

```
curso=# ANALYZE VERBOSE times(nome);
```

O Analyze coleta estatísticas sobre o conteúdo da tabela e armazena na pg_statistic. Essas informações são usadas pelo Otimizador para escolher os melhores Planos de Execução para uma query. Essas estatísticas incluem os valores mais frequentes, a frequência desses valores, percentual de valores nulos etc.

Sempre que houver uma grande carga de dados, seja atualização ou inserção, ou mesmo exclusão, é importante executar uma atualização das estatísticas da tabela, para que não ocorram situações onde o Otimizador tome decisões baseado em informações desatualizadas.

Uma boa prática é um executar um VACUUM ANALYZE após qualquer grande alteração de dados.

Amostra estatística

A análise estatística é feita sobre uma amostra dos dados de cada tabela. O tamanho dessa amostra é determinado pelo parâmetro default_statistics_target. O valor padrão é 100.

Porém, em uma tabela muito grande que tenha muitos valores distintos, essa amostra pode ser insuficiente e levar o Otimizador a não tomar as melhores decisões. É possível aumentar o valor da amostra analisada, mas isso aumentará também o tempo e o espaço consumido pela coleta de estatísticas.

Em vez de alterar o *postgresql.conf* globalmente, pode-se alterar esse parâmetro por coluna e por tabela. Assim, se estiver analisando uma query em particular, ou se existirem tabelas grandes muito utilizadas em um sistema, é uma boa ideia incrementá-lo para as colunas muito utilizadas em cláusulas WHERE, ORDER BY e GROUP BY.

Para isso usa-se o ALTER TABLEs:

```
bench=# ALTER TABLE pgbench_accounts
                    ALTER COLUMN bid SET STATISTICS 1000;
```

Autovacuum

Em função da importância da operação de Vacuum para o PostgreSQL, nas versões mais recentes esse procedimento passou a ser executado de forma automática (embora esse comportamento possa ser desabilitado). Esse mecanismo é chamado de Autovacuum.

Na teoria, o Administrador PostgreSQL não precisa mais executar o vacuum manualmente, bastando configurar corretamente as opções relacionadas ao autovacuum. Na prática deve-se manter a recomendação de sempre executar um vacuum manual quando houver uma alteração grande de dados. Além disso, por precaução, é comum agendar um vacuum noturno.

O autovacuum sempre executa o Analyze. Assim, tem-se também um "auto-analyze" fazendo a coleta estatística frequentemente sem depender de agendamento ou execução do administrador.

Para executar essas tarefas, existe um processo inicial chamado Autovacuum Launcher e um número pré-determinado de processos auxiliares chamados Autovacuum Workers. A cada intervalo de tempo configurado, o Laucher chama um Worker para verificar uma base de dados. O Worker verifica cada tabela e executa o vacum, acionando o analyze se necessário. Se existem mais bases que o número configurado de Workers, as bases serão analisadas em sequência, assim que o trabalho terminar na anterior.

Configurando o Autovacuum

Para funcionar, o autovacuum precisa do parâmetro track_counts igual a true, que é o valor padrão.

O número de processos Workers é definido pelo parâmetro autovacuum_max_workers, sendo o padrão 3. A modificação desse parâmetro vai depender da frequência de atualização e tamanho das tabelas do seu ambiente. Inicie com o valor padrão e acompanhe se a frequência com que os workers estão rodando está muito alta. Em caso positivo, é recomendado incrementar esse número.

Quando um Worker determinar que uma tabela precisa passar por um vacuum, ele executará até um limite de custo, medido em operações de I/O. Depois de atingir esse limite de operações, ele "dormirá" por um determinado tempo antes de continuar o trabalho.

Esse limite de custo é definido pelo parâmetro autovacuum_vacuum_cost_limit, que por padrão é -1, significando que ele usará o valor de vacuum_cost_limit, que por padrão é 200.

O tempo em que o Worker dorme quando atinge o limite de custo é definido pelo parâmetro autovacuum_vacuum_cost_delay**,** que por padrão é 20 ms. Assim, o trabalho do autovacuum é feito de forma pulverizada, evitando conflitos com as operações normais do banco.

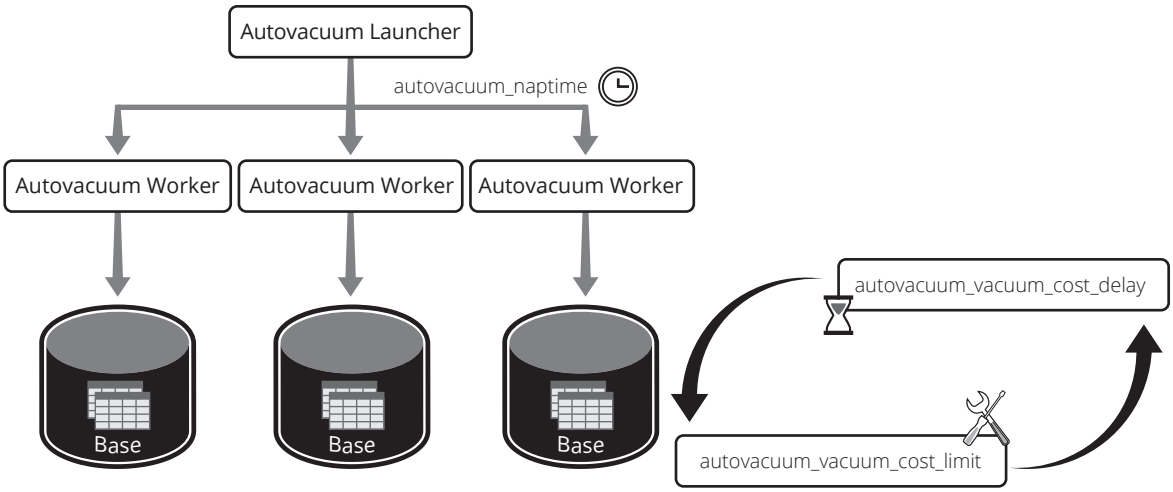


Figura 6.2
Funcionamento do Autovacuum.

Não é tarefa trivial ajustar esses valores, já que eles são relacionados à carga de atualizações de um determinado ambiente. Inicie com os valores default e caso o autovacuum possa estar atrapalhando o uso do banco, aumente o valor de autovacuum_cost_delay para, por exemplo, 100ms, e meça os resultados.

Pode-se também aumentar o parâmetro autovacuum_naptime, que é o intervalo que o Launcher executa os Workers para cada base (por padrão a cada 1 minuto). Caso seja necessário baixar a frequência do autovacuum, aumente o naptime com moderação.

Configurações por tabela

Se uma tabela precisar de configurações especiais de vacuum, como no caso de uma tabela de log, que é muito escrita mas não consultada, pode-se desabilitar o autovacuum nessa tabela e executá-lo manualmente quando necessário. Uma alternativa é ajustar os parâmetros de limite e delay específicos para essa tabela.

Para desabilitar o autovacuum em uma determinada tabela:

```
bench=# ALTER TABLE contas SET (autovacuum_enabled = false);
```

Por exemplo, para permitir que mais trabalho seja feito pelo autovacuum em uma tabela:

```
bench=# ALTER TABLE contas SET (autovacuum_vacuum_cost_limit = 1000);
```

Problemas com o Autovacuum

- Autovacuum executa mesmo desligado
- Autovacuum executando sempre
- Out of Memory
- Pouca frequência
- Fazendo muito I/O
- Transações Eternas

Uma situação relativamente comum é a execução do Vacuum ou do Autovacuum tomar muito tempo e o administrador decidir que isso é um problema e que deve ser evitado. Na verdade, se o vacuum está demorando, é justamente porque ele tem muito trabalho a ser feito e está sendo pouco executado. A solução não é parar de executá-lo, mas sim executá-lo com maior frequência.

Autovacuum executa mesmo desligado

O autovacuum pode rodar, mesmo se desabilitado no *postgresql.conf*, para evitar o problema conhecido como Transaction ID Wraparound, que é quando o contador de transações do PostgreSQL está chegando ao limite. Esse problema pode gerar perda de dados devido ao controle de visibilidade de transações e somente ocorrerá se você desabilitar o autovacuum e também não executar o vacuum manualmente.

Autovacuum executando sempre

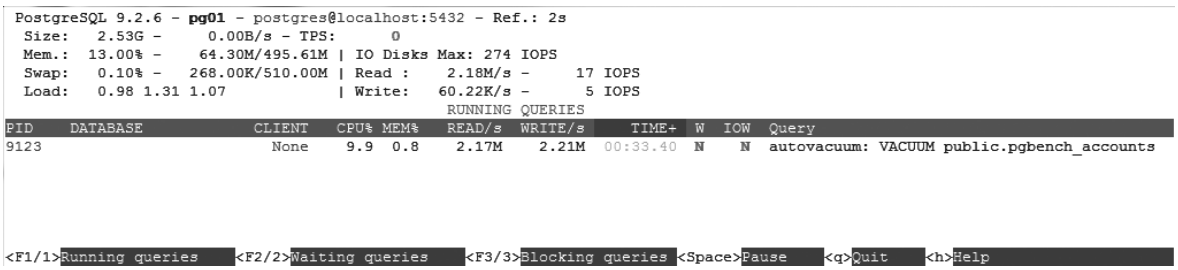


Figura 6.3
Um worker do Autovacuum em atividade.

Verifique se o parâmetro maintenance_work_mem está muito baixo comparado ao tamanho das tabelas que precisam passar pelo vacuum. Lembre-se de que o vacuum/autovacuum pode alocar no máximo *maintenance_work_mem* de memória para as operações. Ao atingir esse valor o processo para e começa novamente.

Outra possibilidade é se há um grande número de bases de dados no servidor. Nesse caso, como uma base não pode ficar sem passar pelo autovacuum por mais do que o definido em autovacuum_naptime, se existirem 30 bases, um worker disparará no mínimo a cada 2s. Se há muitas bases, aumente o autovacuum_naptime.

Out of Memory

Ao aumentar o parâmetro maintenance_work_mem, é preciso levar em consideração que cada worker pode alocar até essa quantidade de memória para sua respectiva execução. Assim, considere o número de workers e o tamanho da RAM disponível quando for atribuir o valor de maintenance_work_mem.

Pouca frequência

Em grandes servidores com alta capacidade de processamento e de I/O, com sistemas igualmente grandes, o parâmetro autovacuum_vacuum_cost_delay deve ter seu valor padrão de 20ms baixado para um intervalo menor, de modo a permitir que o autovacuum dê conta de executar sua tarefa.

Fazendo muito I/O

Se o autovacuum parecer estar consumindo muito recurso, ocupando muita banda disponível de I/O, pode-se aumentar autovacuum_vacuum_cost_delay para 100ms ou 200ms, buscando não atrapalhar as operações normais do banco.

Transações eternas

Pode parecer impossível, mas existem sistemas construídos de tal forma, propositalmente ou por bug, que transações ficam abertas por dias. O vacuum não poderá eliminar as dead tuples que ainda devem ser visíveis até essas transações terminarem, prejudicando assim sua operação. Verifique a idade das transações na pg_stat_activity ou com o pg_activity.

Reindex

O comando REINDEX pode ser usado para reconstruir um índice. Você pode desejar executar essa operação se suspeitar que um índice esteja corrompido, "inchado" ou, ainda, se foi alterada alguma configuração de armazenamento do índice, como FILLFACTOR, e que não tenha ainda sido aplicada. O REINDEX faz o mesmo que um DROP seguido de um CREATE INDEX.

É possível também usar o REINDEX quando um índice que estava sendo criado com a opção CONCURRENTLY falhou no meio da operação. Porém, o REINDEX não possui a opção concorrente, fazendo com que o índice seja recriado com os locks normais. Nesse caso é melhor remover o índice e criá-lo novamente com a opção CONCURRENTLY.

As opções do comando REINDEX são:

- Reindexar um índice específico:

```
curso=# REINDEX INDEX public.pgbench_branches_pkey;
```

- Reindexar todos os índices de uma tabela:

```
curso=# REINDEX TABLE public.pgbench_branches;
```

- Reindexar todos os índices da base de dados:

```
curso=# REINDEX DATABASE curso;
```

- Uma alternativa é a reconstrução dos índices do catálogo.

```
curso=# REINDEX SYSTEM curso;
```

Para tabelas e índices deve-se informar o esquema e para a base é obrigatório informar o nome da base e estar conectado a ela.

'Bloated Indexes'

Para verificar se um índice está inchado, basicamente devemos comparar o tamanho do índice com o tamanho da tabela. Apesar de alguns índices realmente poderem ser quase tão grandes quanto sua tabela, deve-se considerar as colunas no índice.

A seguinte query mostra o tamanho dos índices, de suas tabelas e a proporção entre eles.

```
bench=# SELECT       nspname, relname, round(100 * pg_relation_size(indexrelid)
    / pg_relation_size(indrelid)) / 100 as index_ratio,
pg_size_pretty(pg_relation_size(indexrelid)) as index_size,
pg_size_pretty(pg_relation_size(indrelid)) as table_size
 FROM pg_index I
LEFT JOIN pg_class C ON (C.oid=I.indexrelid)
LEFT JOIN pg_namespace N ON (N.oid = C.relnamespace)
 WHERE nspname NOT IN ('pg_catalog','information_schema','pg_toast')
  AND C.relkind = 'i' AND pg_relation_size(indrelid) > 0;
```

```
 nspname |        relname        | index_ratio | index_size | table_size
---------+-----------------------+-------------+------------+------------
 public  | pgbench_branches_pkey |           2 | 16 kB      | 8192 bytes
 public  | pgbench_tellers_pkey  |           1 | 40 kB      | 40 kB
 public  | pgbench_accounts_pkey |        0.17 | 302 MB     | 1713 MB
 public  | idx_accounts_bid      |        0.14 | 257 MB     | 1713 MB
 public  | idx_branches_bid_tbal |           1 | 40 kB      | 40 kB
 public  | idx_accounts_bid_parc |           0 | 48 kB      | 1713 MB
 public  | idx_accounts_bid_coal |        0.14 | 257 MB     | 1713 MB
 public  | idx_contas            |       85.78 | 21 MB      | 256 kB
```

Figura 6.4
Bloated Indexes.

No exemplo mostrado nessa figura, o índice idx_contas está 85 vezes maior que sua tabela. Certamente deve passar por um REINDEX.

Cluster e 'Recluster'

O recurso de CLUSTER é uma possibilidade para melhorar o desempenho de acesso a dados lidos de forma sequencial. Por exemplo, se há uma tabela Item que possui um ID da nota fiscal, provavelmente todos os itens serão frequentemente recuperados da tabela Item pela chave da nota. Nesse caso, ter um índice nesta coluna e fazer o CLUSTER por ele pode ser muito vantajoso, pois em uma mesma página de dados lida do disco haverá diversos registros com o mesmo ID da nota fiscal. Podemos "clusterizar" uma tabela com a seguinte sintaxe:

```
# CLUSTER pgbench_accounts USING idx_accounts_bid;
```

Esta é uma operação que usa muito espaço em disco, já que ela cria uma nova cópia da tabela inteira e seus índices de forma ordenada e depois apaga a original. Em alguns casos, dependendo do método de ordenação escolhido, pode ser necessário alocar espaço equivalente a duas vezes o tamanho da tabela, mais seus índices. Essa operação usa bloqueios agressivos, exigindo um lock exclusivo na tabela toda.

O cluster não ordena os dados que futuramente serão inseridos na tabela; assim, essa é uma operação que deve ser reexecutada frequentemente para manter os novos dados também ordenados.

Uma vez executado o CLUSTER, não é necessário informar o índice nas execuções seguintes, a não ser que seja necessário mudá-lo:

```
# CLUSTER pgbench_accounts;
```

É possível executar o CLUSTER em todas as tabelas já clusterizadas da base, bastando não informar nenhuma tabela. No exemplo a seguir, é mostrada também a opção VERBOSE, que gera uma saída detalhada:

```
# CLUSTER VERBOSE;
```

O cluster é uma operação cara de manter, sendo necessário ter janelas de tempo e espaço em disco às vezes generosos para executar a manutenção frequente do cluster. Por outro lado, pode trazer benefícios proporcionais para alguns tipos de queries.

Pode-se fazer o cluster inicial e verificar o ganho de desempenho alcançado, monitorando a degradação desse desempenho de modo a estabelecer uma agenda para novas execuções do cluster para garantir a ordenação dos novos dados que venham a ser incluídos.

Em todas as operações de manutenção mostradas nesta sessão (VACUUM, REINDEX e CLUSTER), o parâmetro maintenance_work_mem deve ser ajustado adequadamente.

Atualização de versão do PostgreSQL

Minor version

Para atualização de minor versions, por exemplo da versão 9.3.3 para a 9.3.4, não há alteração do formato de armazenamento dos dados. Assim, a atualização pode ser feita apenas substituindo os executáveis do PostgreSQL sem qualquer alteração nos dados.

Geralmente, pode-se obter os fontes da nova versão e os compilar em um diretório qualquer seguindo as instruções de instalação normais. Em seguida, desliga-se o PostgreSQL e substitui-se os executáveis, conforme a seguinte sequência de comandos:

```
$   pg_ctl stop -mf
$   rm -Rf /usr/loca/pgsql/
$   cp -r /diretório_compilada_nova_versao/*   /usr/local/pgsql/
$   pg_ctl start
```

Uma alternativa um pouco mais "elegante", e que fornece a vantagem de poder rapidamente retornar em caso de problemas, é seguir o procedimento de instalação ilustrado na sessão 1, adicionando o detalhe de instalar o PostgreSQL em um diretório nomeado com a respectiva versão. Por exemplo:

```
$   sudo tar -xvf postgresql-9.3.4.tar.gz
$   cd postgresql-9.3.4/
$   ./configure --prefix=/usr/local/pgsql-9.3.4
$   make
$   sudo make install
```

Supondo que já exista uma versão instalada, digamos 9.3.3, sob o diretório "/usr/local/", teríamos o seguinte:

```
    $       ls -l /usr/local/
    ...
    ... pgsql -> /usr/local/pgsql-9.3.3/
... pgsql-9.3.3/
... pgsql-9.3.4/
...
```

Nesse caso precisamos baixar o banco e alterar o link simbólico "pgsql" para apontar para uma nova versão. Por exemplo:

```
    $       pg_ctl stop -mf
    $       rm pgsql
    $       ln -s /usr/local/pgsql-9.3.4 pgsql
    $       pg_ctl start
```

Agora teríamos a seguinte situação:

```
    $       ls -l /usr/local/
    ...
    ... pgsql -> /usr/local/pgsql-9.3.4/
... pgsql-9.3.3/
... pgsql-9.3.4/
...
```

Assim, em caso de ocorrência de algum problema, é muito fácil retornar para a versão anterior simplesmente fazendo o mesmo procedimento recém-mostrado de modo a voltar o link simbólico para o diretório original.

Major version

Atualizações de major versions, ou seja, de uma versão 9.2.x para 9.3.x, podem trazer modificações no formato de armazenamento dos dados ou no catálogo de sistema. Nesse caso será necessário fazer um dump de todos os dados e restaurá-los na nova versão. Existem diferentes abordagens que podem ser seguidas em uma situação como essa, muito embora o primeiro passo para todas elas seja encerrar o acesso ao banco. Vejamos algumas situações daí em diante:

- Substituição simples:
  - Fazer o dump de todo o servidor para um arquivo;
  - Desligar o PostgreSQL;
  - Apagar o diretório dos executáveis da versão antiga;
  - Instalar a nova versão no mesmo diretório;
  - Ligar a nova versão do PostgreSQL;
  - Restaurar o dump completo do servidor.
- Duas instâncias em paralelo:
  - Instalar a nova versão em novos diretórios (executáveis e dados);
  - Configurar a nova versão em uma porta TCP diferente;
  - Ligar a nova versão do PostgreSQL;
  - Fazer o dump da versão antiga e o restore na versão nova ao mesmo tempo;
  - Desligar o PostgreSQL antigo;
  - Configurar a nova versão para a porta TCP original.
- Novo servidor:
  - Instalar a nova versão do PostgreSQL em um novo servidor;
  - Fazer o dump da versão antiga e transferir o arquivo para o novo servidor ou fazer o dump da versão antiga e o restore na versão nova ao mesmo tempo;
  - Direcionar a aplicação para o novo servidor;
  - Desligar o servidor antigo.

A escolha entre essas abordagens dependerão da janela de tempo e dos recursos disponíveis em seu ambiente.

Uma alternativa para o dump/restore na atualização de major versions é o utilitário pg_upgrade. Esse aplicativo atualiza a versão do PostgreSQL in-loco, ou seja, atualizando os arquivos de dados diretamente. É possível utilizá-lo em modo "cópia" ou em modo "link", ambos mais rápidos do que o dump/restore tradicional. No modo "link" é ainda possível fazer a atualização de uma base de centenas de gigas em poucos minutos.

Resumo

Vacuum:

- Mantenha o Autovacuum sempre habilitado;
- Agende um Vacuum uma vez por noite;
- Não use o Vacuum Full, a não ser em situação especial;
- Tabelas que estejam sofrendo muito autovacuum devem ter o parâmetro autovacuum_vacuum_cost_limit aumentado.

Estatísticas:

- O Auto-Analyze é executado junto com o Autovacuum, por isso mantenha-o habilitado;
- Na execução noturna do Vacuum, adicione a opção Analyze;
- Considere aumentar o tamanho da amostra estatística das principais tabelas.

Problemas com Autovacuum:

- Se existirem muitas bases, aumente autovacuum_naptime;
- Ao definir maintenance_work_mem, considere o número de workers e o tamanho da RAM;
- Em servidores de alto poder de processamento, pode-se baixar autovacuum_vacuum_cost_delay;
- Se o autovacuum estiver muito frequente e fazendo muito I/O, pode-se aumentar autovacuum_vacuum_cost_delay.

Reindex e Cluster:

- Use Reindex se mudar o fillfactor de um índice ou para índices inchados.
- Se existirem tabelas pesquisadas por uma sequência de dados, pode-se usar o Cluster;
- Tabelas que foram "Clusterizadas" devem ser reclusterizadas periodicamente.

Atualização de versão:

- Atualização de minor versions necessitam apenas da troca dos executáveis.
  - Uso de link simbólico facilita.
- Atualização de major versions necessitam dump/restore dos dados.
  - Fazer dump e substituir a instalação atual por uma nova;
  - Instalar duas instâncias em paralelo e fazer dump/restore direto entre elas;
  - Criar novo servidor e fazer dump/restore direto ou transferir arquivo.

7

Desempenho – Tópicos sobre aplicação

objetivos

Entender os problemas mais comuns relacionados ao desempenho do PostgreSQL ao gerenciar conexões ou aplicações; Conhecer alternativas para a sua solução, incluindo boas práticas que podem ajudar a contornar alguns desses obstáculos.

conceitos

Bloqueios; Deadlock; Índices; Índices Compostos; Índices Parciais; Índices com Expressões; Operadores de Classes; Planos de Execução; Índex Scan e Seq Scan.

Exercício de Nivelamento

O seu chefe fala para você: "O banco está lento, o sistema está se arrastando!"

O que você faz?

Introdução ao tuning

- Não existe uma receita que se aplique a todos os casos de todas as tabelas da base
- Diretrizes: linhas gerais e boas práticas
- Conhecimento Empírico
  - Depende do seu ambiente
  - Aplicar e Medir

Ajuste de desempenho, comumente descrito através do termo tuning, é uma das mais difíceis tarefas de qualquer profissional de infraestrutura de TI. E o motivo é simples: não existe uma receita que se aplique a todos os casos. Frequentemente uma ação que foi uma solução em um cenário pode ser inútil em outro, ou pior, ser prejudicial.

Mas também não chega a ser uma arte ou ciência exotérica. Existem linhas gerais, diretrizes e conjuntos de boas práticas que podem e devem ser seguidos. Mas sempre com as seguintes ressalvas: "depende de seu ambiente" ou "aplique e teste". Resumidamente, tuning é isto: um conhecimento empírico que deve ser testado em cada situação.

O ajuste de desempenho do PostgreSQL não é diferente. Para cada query lenta que se enfrente, diversas soluções possíveis existirão, sendo que cada uma delas deve ser analisada e empiricamente testada até se encontrar a que melhor resolva ou mitigue uma situação negativa.

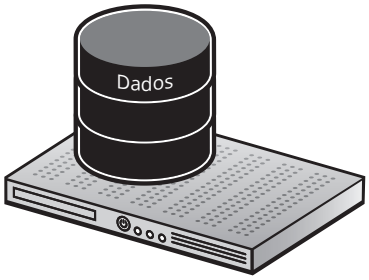


Figura 7.1
Servidor dedicado.

Uma das poucas afirmações que podemos fazer nesta área sem gerar qualquer controvérsia é a de que o Banco de Dados deve ser instalado em um servidor dedicado. Ainda que isso possa parecer muito óbvio, é comum encontrarmos servidores de aplicação, ou servidores web, na mesma máquina do Banco de Dados. Isso é especialmente comum para produtos de prateleira, "softwares de caixa", onde o aplicativo e o Banco de Dados são instalados no mesmo servidor. Poderão até existir situações onde não existir uma rede entre a aplicação e o banco possa compensar a concorrência por CPU, memória, canais de I/O, recursos de SO, instabilidades e manutenções duplicadas. Mas essas são muito raras.

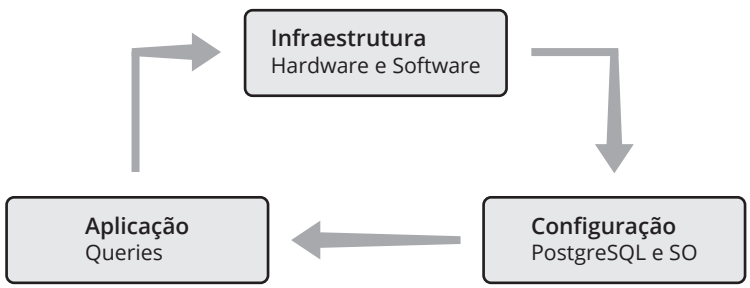


Figura 7.2
Ciclo de Tuning.

É importante que o administrador reconheça, o quanto antes, que tuning não é um projeto que tem início e fim. Tuning é uma atividade permanente. Ou seja, nunca acaba. Quando você achar que tudo o que era possível foi otimizado, alguma mudança de plataforma ou alteração drástica de negócios trará um novo cenário que demandará novos ajustes.

Tendo isso em mente, serão agora analisadas algumas das situações mais frequentemente encontradas por administradores PostgreSQL na busca por melhor desempenho, juntamente com as estratégias para a solução dos problemas mais conhecidos.

Essas soluções podem estar no modo como a aplicação usa o banco, na qualidade de suas queries, da eficiência do modelo, na quantidade de dados retornados ou ainda na quantidade de dados processados de forma intermediária. Muitas vezes a solução mais eficaz para a lentidão é simples, e até por conta disso muitas vezes esquecida: criação de Índices.

Depois de analisada a aplicação e o uso que está fazendo do banco, pode ser necessário analisar a configuração do PostgreSQL e do Sistema Operacional. Pode-se analisar se a memória para ordenação é suficiente, o percentual de acerto no shared buffer, e ainda os parâmetros de custo do Otimizador. É preciso se certificar também que a reorganização do espaço, o vacuum, e estatísticas estão sendo realizados com a frequência necessária.

Por fim, verificamos se a memória está corretamente dimensionada, a velocidade e organização dos discos, o poder de processamento, o filesystem e o uso de mais servidores com replicação. Ou seja, olha-se para infraestrutura de hardware e software.

São muitas as alternativas que precisam ser consideradas até encontrar uma solução para um problema de desempenho. Raramente alguém trará um problema parcialmente depurado, indicando que a rotina tal está com problemas. Geralmente o administrador recebe apenas uma reclamação genérica: "O sistema está lento!"

As informações colhidas através do monitoramento do banco, conforme foi visto na sessão 5, serão importantes para determinar se houve alterações no sistema ou no ambiente. Como estará a carga (load) do servidor? Se estiver alta, é importante localizar a origem da sobrecarga, de imediato tentando identificar se é um problema específico (existe um ou poucos processos ocasionando o problema) ou é uma situação geral.

Lentidão generalizada

Um problema comum é o excesso de conexões com o banco, resultando em excesso de processos. Cada processo tem seu espaço de endereçamento, memória alocada particular, além das estruturas para memória compartilhada e o tempo de CPU. Mesmo quando um processo não faz nada, ele ocupará tempo de processador quando é criado e quando é destruído.

Mas o que é excesso de processos? Qual é a quantidade normal de processos? Esta não será a única vez que vai ouvir como resposta: depende de seu ambiente! Depende das configurações do pool – se existe um pool, depende do uso do sistema, depende de seu hardware. Provavelmente, com o tempo o administrador já conhecerá um número de conexões para qual o servidor ou sistema em questão voa em velocidade de cruzeiro. Caso não saiba, o ideal é ter o histórico disso anualizado através das ferramentas Cacti, Zabbix ou similar, como já foi visto.

De qualquer modo, um número muito grande de processos (centenas) compromete a resposta de qualquer servidor. Para manipular milhares de conexões, você deve usar um software de pooling – ou agregador de conexões.

Se o sistema é uma aplicação web, rodando em um servidor de aplicação, ela provavelmente já faz uso de uma camada de pool. Nesse caso, deve-se verificar as configurações de pool. Por vezes o número mínimo de conexões, o máximo e o incremento são superdimensionados. O mínimo normalmente não é o grande vilão, pois é alocado quando a aplicação entra no ar. O incremento é o número de conexões a serem criadas quando faltarem conexões livres. Ou seja, quando todas as conexões existentes estiverem alocadas, não será criada apenas uma conexão adicional, mas sim a quantidade de conexões indicada no parâmetro relacionado ao incremento. Se esse número for 30, toda vez que se esgotarem as conexões disponíveis, serão solicitados ao SO e ao PostgreSQL a criação de 30 novos processos de uma só vez. Tenha em mente que o mecanismo de abertura de conexões no PostgreSQL não é barato. Assim, um número muito alto para o máximo de conexões que podem ser criadas pode também comprometer a performance do sistema como um todo.

Caso não exista um pool, como é comum em sistemas duas camadas ou desktop, a adoção de um software de pool pode ajudar, e muito. Sem essa camada, se o sistema estiver instalado em 300, 500 ou 1.000 estações, você terá uma conexão de cada cliente com o banco. Esse grande número de processos pode exaurir a memória do servidor e esgotar os processadores.

pgbouncer

Um software de pool para PostgreSQL é o pgbouncer. Ele é open source, de fácil instalação, leve e absurdamente eficiente. Já a sua configuração pode demandar o entendimento de conceitos mais complexos para os iniciantes, tornando o processo um pouco confuso. Nada que não possa ser resolvido com a leitura cuidadosa da documentação existente, e que certamente trará um resultado mais do que compensador.

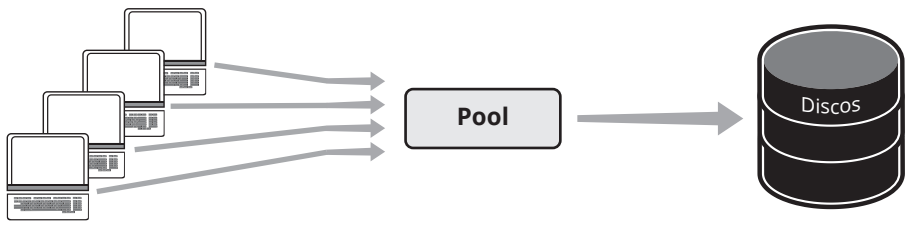


Figura 7.3
Pool de conexões.

Com o pgbouncer é possível atender a mil ou mais conexões de clientes através de um número significativamente menor de conexões diretas com o banco. Isso poderá trazer ganhos de desempenho notáveis.

A instalação do pgbouncer pode ser feita através dos seguintes comandos:

```
$   sudo apt-get install libevent-dev
$   cd /usr/local/src/
$   sudo wget http://pgfoundry.org/frs/download.php/3393/pgbouncer-1.5.4.tar.gz
$   sudo tar -xvf pgbouncer-1.5.4.tar.gz
$   cd pgbouncer-1.5.4/
$   ./configure --prefix=/usr/local --with-libevent=libevent-prefix
$   make
$   sudo make install
```

O passo seguinte é tratar da sua configuração, que no exemplo a seguir segue um esquema básico:

```
$   sudo mkdir /db/pgbouncer
$   sudo chown postgres /db/pgbouncer/
$   vi /db/pgbouncer/pgbouncer.ini
[databases]
curso = host=127.0.0.1 dbname=curso

[pgbouncer]
pool_mode = transaction
listen_port = 6543
listen_addr = 127.0.0.1
auth_type = md5
auth_file = /db/pgbouncer/users.txt
logfile = /db/pgbouncer/pgbouncer.log
pidfile = /db/pgbouncer/pgbouncer.pid
admin_users = postgres
stats_users = stat_collector
```

Finalmente, para executar o pgbouncer basta chamar o executável informando o arquivo de configuração recém-criado:

```
$    pgbouncer -d /db/pgbouncer/pgbouncer.ini
```

Para testar, use o psql para conectar na porta do pool (configurada no arquivo como 6543) e acesse alguma das bases permitidas na seção [databases]. No exemplo a seguir, a base escolhida é curso:

```
$    psql -p 6543 -d curso -U aluno
```

Importante: como nada é perfeito, o pgbouncer possui um contratempo. Como ele precisa autenticar os usuários, é preciso indicar um arquivo com usuários e senhas válidos.
Até a versão 8.3, o próprio PostgreSQL mantinha um arquivo assim que era utilizado pelo pgbouncer. A partir do PostgreSQL 9.0, esse arquivo foi descontinuado e atualmente é necessário gerá-lo "manualmente". Não é complicado gerar o conteúdo para o arquivo através de comandos SQL lendo o catálogo, inclusive protegendo as respectivas senhas de forma a que não fiquem expostas (são encriptadas).

Mesmo com um pool já em uso e bem configurado, se a carga normal do sistema exigir um número muito elevado de conexões no PostgreSQL, alternativas terão de ser consideradas. Será preciso analisar suas configurações e infraestrutura, e isso será abordado na próxima sessão.

Processos com queries lentas ou muito executadas

Usando as ferramentas que vimos na sessão sobre monitoramento, é possível identificar processos que estão demorando muito. Nesses casos, é comum encontrar processos que estão bloqueados, seja por estarem aguardando outros processos, seja por estarem aguardando operações de I/O. Se o problema é I/O, pode ser um problema de infraestrutura que será abordado na próxima sessão. Mas a causa pode ser também o volume de dados manipulado ou retornado pela query.

Outra possibilidade é ter uma mesma query sendo muito executada por diversos processos, situação que pode ser identificada analisando as queries pelo pg_activity e constatando que não são os mesmos IDs de processo. De qualquer modo, analisaremos em mais detalhes cada uma destas situações.

Volume de dados manipulados

Identificada uma query mais demorada, podemos testá-la para verificar a quantidade de registros retornados. É muito comum que os desenvolvedores das aplicações criem consultas que quando estão no ambiente de desenvolvimento ou homologação são executadas muito rapidamente porque não há grande volume de dados nesses ambientes. Em produção, com o tempo, o volume de dados cresce gradualmente ou pode sofrer algum tipo de carga de dados grande e a query começa a apresentar problemas.

Também temos de considerar que geralmente o desenvolvedor está preocupado em entregar aquela funcionalidade de negócio e desempenho não é sua preocupação principal. Infelizmente é comum encontrar consultas online que retornam milhares ou até dezenas de milhares de registros. Isso pode demorar a ser identificado, já que a consulta funciona corretamente e a aplicação exibirá esses dados paginados. Assim, o usuário vai consultar os dados na primeira ou segunda tela para logo em seguida passar para outra atividade, descartando o grande volume de dados excedente que foi processado e trafegado na rede.

Se esse for o caso, a query deve ser reescrita para ser mais restritiva. Basta aplicar um filtro, por exemplo, incluindo mais condições na cláusula WHERE da query, de modo que esta passe a retornar menos dados.

Caso a consulta esteja correta e não seja possível restringir a quantidade de dados retornados, a recomendação então é passar a fazer uso das cláusulas OFFSET e LIMIT, resultando no tráfego apenas dos dados que realmente serão sendo exibidos para o usuário. Na primeira página, passa-se OFFSET 0 e LIMIT 10 (supondo a paginação de tamanho 10), aumentando o OFFSET em 10 para as páginas subsequentes.

```
postgres=# SELECT generate_series(1,30) OFFSET 10 LIMIT 10;
 generate_series
-----------------
              11
              12
              13
              14
              15
              16
              17
              18
              19
              20
(10 rows)

postgres=#
```

Figura 7.4
Exemplo de paginação com OFFSET e LIMIT.

Exemplo de uso de OFFSET e LIMIT da figura 7.3 usando a função generate_series:

```
curso=#      SELECT generate_series(1,30) OFFSET 10 LIMIT 10;
```

Outro problema em potencial é a quantidade de registros manipulados nas operações intermediárias da query. Se for uma query complexa, com diversos joins, subqueries e condições, é possível que o resultado final seja pequeno, mas com milhões de registros sendo comparados nas junções de tabelas. A solução é a mesma sugerida na situação anterior, ou seja, tentar tornar a query mais restritiva de forma que diminua a quantidade de registros a serem ordenados e comparados, para montar o resultado final.

Além da quantidade de registros, a quantidade e os tipos de colunas podem influenciar o desempenho de uma query. Numa query que retorne 200 registros, cada registro com 40 colunas (pior ainda, se algumas dessas colunas forem campos texto ou conteúdo binário de arquivos ou imagens), o resultado final pode ser um volume muito elevado de dados a serem processados e trafegados.

A solução é reescrever a query e adaptar a aplicação para retornar um número mais enxuto de colunas, sempre o mínimo necessário. São poucas as aplicações que precisarão de consultas online retornando quantidade grande de registros, com todas as colunas e campos com dados multimídia. Por exemplo, se uma consulta retorna uma lista de registros e cada registro contém uma coluna do tipo bytea – para dados binários – contendo um arquivo, dificilmente a aplicação abrirá todos esses arquivos ao mesmo tempo. Nesse caso, tal coluna não precisaria ser retornada nessa consulta. Apenas quando o usuário explicitar o desejo de consultar ou manipular tal arquivo é que deve-se ir até o banco buscá-lo na coluna binária. O mesmo vale para campos text com grande volume de texto, muitas vezes com conteúdo html.

Importante: não é recomendável o armazenamento de arquivos no banco de dados.

Frequentemente os desenvolvedores vão querer aproveitar a facilidade de armazenar imagens, arquivos pdf e outros em campos do tipo bytea. Apesar da vantagem da integridade referencial, bancos de dados não são pensados, configurados e ajustados para trafegar e armazenar grandes volumes de dados armazenados em arquivos. Tipicamente bancos são voltados para sistemas transacionais, OLTP, para manipular registros pequenos, de um tamanho médio de 8kB. No PostgreSQL, em especial, isso causará sérios problemas com os dumps da base, aumentando significativamente o tempo de realização do backup e também o tamanho do arquivo resultante. Além disso, este tipo de coluna pode "poluir" o log e os relatórios gerados com o pgBadger, sem falar no "prejuízo" recorrente ao trafegar essas colunas na operações com o banco.

A melhor estratégia é armazenar esses arquivos externamente, mantendo no banco apenas um "ponteiro" para sua localização física. Existem ainda soluções próprias, como sistemas de GED ou storages NAS, com recursos como compactação e desduplicação.

De qualquer modo, se for inevitável armazenar arquivos no banco, adote procedimentos especiais para as tabelas que conterão arquivos, usando tablespaces separados e não fazendo dump delas, apenas backups físicos.

Relatórios e integrações

Se existem consultas no seu ambiente que apresentam os problemas com volume de dados conforme acima apresentados, e cujos resultados são demandados por um ou mais usuários, considere a possibilidade de não disponibilizar essas consultas de forma online, mas sim como relatórios cuja execução pode ser programada.

Existem diversas ferramentas, como o Pentaho, Jasper Server e outras comerciais, que possuem facilidades para geração de relatórios, executando as consultas agendadas e fazendo cache ou snapshots dos resultados. Desse modo, toda vez que um usuário pede determinado relatório, a consulta não é mais disparada contra o banco, mas extraída desse snapshot dos dados.

Um erro muito comum é criar relatórios para usuários, às vezes fechamentos mensais ou até anuais, e disponibilizar um link no sistema para o usuário gerá-lo a qualquer instante. Relatórios devem ser pré-processados, agendados para executar à noite e de madrugada, e apresentar o resultado para o usuário pela manhã.

Integrações entre sistemas por vezes também são processos pesados que não devem ser executados em horário de pico. Cargas grandes de escrita ou leitura para integração de dados entre sistemas feitas com ferramentas de ETL, Web Services ou outras soluções similares devem ter o horário controlado ou serem pulverizadas entre diversas chamadas com pouco volume de dados a cada vez.

Se ainda assim existem consultas pesadas que precisem ser executadas a qualquer momento, ou relatórios que não podem ter seus horários de execução restringidos, considere usar Replicação (será tratada na sessão 10) para criar servidores slaves, onde essas consultas poderão ser executadas sem prejudicar as operações normais do sistema.

Bloqueios

O PostgreSQL controla a concorrência e garante o Isolamento (o "I" das propriedades ACID) com um mecanismo chamado Multi-Version Concurrency Control (MVCC). Devido ao MVCC, problemas de bloqueios – ou locks – no PostgreSQL são pequenos. Esse mecanismo basicamente cria versões dos registros, que podem estar sendo manipulados simultaneamente por transações diferentes, cada uma tendo uma visão dos dados, chamada snapshot.
Essa é uma excelente forma de evitar contenção por locks. A forma mais simples de explicar a vantagem desse mecanismo é que no PostgreSQL uma leitura nunca bloqueia uma escrita e uma escrita nunca bloqueia uma leitura.

Explicado isso, fica claro que situações de conflitos envolvendo locks são restritas a operações de escritas concorrentes. Na prática, a maioria das situações estão ligadas a operações de UPDATE. Os INSERTs geram informação nova, não havendo concorrência. Já os DELETEs podem também apresentar problemas com locks, mas são bem mais raros. Até porque uma prática comum em muitos sistemas é não remover seus registros, apenas marcá-los como inativos ou não usados.

Mesmo as situações de conflitos geradas por locks em UPDATEs não chegam a ser um problema, já que são situações rotineiras na medida em que o lock é um mecanismo de controle de compartilhamento comum do banco. Problemas surgem de fato quando uma transação que faz um lock demora para terminar ou quando ocorre um deadlock. A seguir analisamos cada uma dessas situações.

Lembre-se, locks não são um problema. Problema é a transação não liberar o lock!

Quando um lock é obtido sobre um registro para ser feito um UPDATE, por exemplo, ele somente será liberado ao final da transação que o obteve. Se a transação tiver muitas operações após ter adquirido o lock, ou não envie o comando de final de transação por algum bug ou outro motivo qualquer, ela pode dar origem a vários processos bloqueados, podendo criar um problema em cascata.

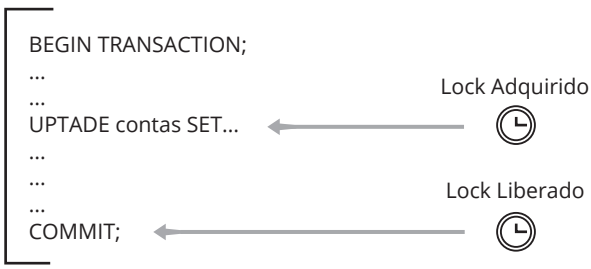


Figura 7.5
Transações longas e bloqueios.

Essa situação é mais frequente com o uso de camadas de persistência e frameworks, pois o desenvolvedor não escreve mais o código SQL. Ele não é mais responsável por abrir ou fechar transações explicitamente, muito provavelmente apenas "marcando" o seu método como transacional, deixando para as camadas de acesso a dados a execução das operações de BEGIN e COMMIT (ou ROLLBACK).

A solução é sempre fazer a transação o mais curta possível. Deve ser encontrado o motivo pelo qual a transação está demorando, providenciando a reescrita desta se necessário. Vimos na sessão de aprendizagem sobre monitoramento que podemos localizar os locks através do pg_activity, do pgAdmin e da tabela do catálogo pg_locks. É possível também rastrear queries envolvidas em longas esperas por locks ligando o parâmetro log_lock_waits.

No *postgresql.conf*, defina:

```
log_lock_waits = on
```

Serão registradas mensagens no log como esta (o prefixo da linha foi suprimido para clareza):

```
user=aluno,db=curso LOG:  process 15186 still waiting for ExclusiveLock on tuple (
0,7) of relation 24584 of database 16384 after 1002.912 ms
user=aluno,db=curso STATEMENT:  UPDATE grupos SET nome = 'X' WHERE id=7;
```

Com os IDs dos processos é possível localizar na log as operações correspondentes para entender o que as transações fazem e avaliar se podem ser melhoradas.

Se o bloqueio estiver causando problemas graves, a solução pode ser simplesmente matar o processo.

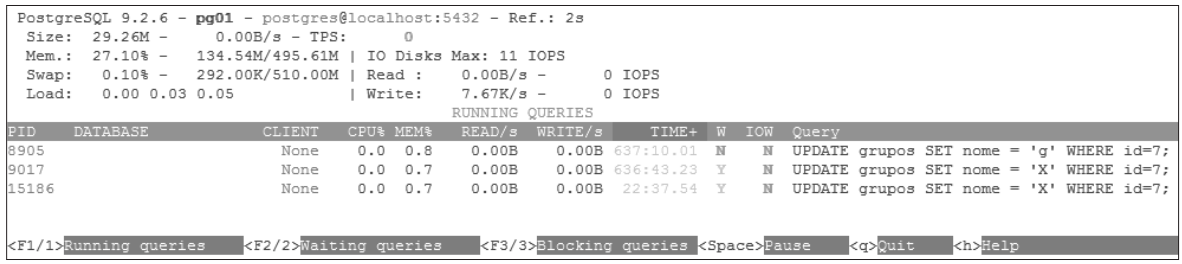


Figura 7.6
Transações antigas boqueando processos.

Se o processo bloqueante for eliminado, deverá ser visto no log algo como o seguinte:

```
user=postgres,db=postgres LOG:  statement: SELECT pg_terminate_backend('8905')
user=aluno,db=curso LOG:  process 15186 acquired ExclusiveLock on tuple (0,7) of r
elation 24584 of database 16384 after 1601203.086 ms
user=aluno,db=curso STATEMENT:  UPDATE grupos SET nome = 'X' WHERE id=7;
```

No exemplo da figura 7.5, o processo 8905 está há mais de 637 minutos executando. Um olhar mais cuidadoso nos dados de CPU, READ/s e WRITE/s permitirá concluir que o processo não está consumindo recursos, simplesmente está parado. O comando exibido, nesse caso um UPDATE, pode não estar mais em execução, embora tenha sido o último comando executado pela transação. Isso pode ser verificado na tabela pg_stat_activity, onde estado do processo é IDLE IN TRANSACTION. Processos nesse estado por longo tempo são o problema a ser resolvido, mas é um comportamento que varia de aplicação para aplicação.

Além dos problemas com locks relacionados a escritas de dados como UPDATE e DELETE, há as situações menos comuns e mais fáceis de identificar envolvendo DDL. Comandos como ALTER *TABLE* e *CREATE INDEX* também bloquearão escritas de dados. Essas alterações de modelo devem ocorrer em horário de baixa atividade do sistema.

Uma outra situação que pode ocorrer são bloqueios gerados por causa do autovacuum. Se uma tabela passar por uma grande alteração de dados, ela é grande candidata a sofrer autovacuum, potencialmente gerando problemas de performance. Se uma situação como essa ocorrer, uma alternativa é eliminar o processo e configurar a tabela para não ter autovacuum.

Mas o bloqueio mais "famoso" é o deadlock. Essa é uma situação especial de lock, necessariamente envolvendo mais de um recurso, no nosso caso provavelmente registros, onde cada processo obteve um registro e está esperando o do outro ser liberado, o que nunca acontecerá. É uma situação clássica na ciência da computação sobre concorrência de recursos.

Deadlocks somente ocorrerão se existirem programas que acessam registros em ordens inversas. Se houver uma lógica geral de ordem de acesso aos registros, garantindo que os programas sempre acessam os recursos na mesma ordem, um deadlock nunca acontecerá.

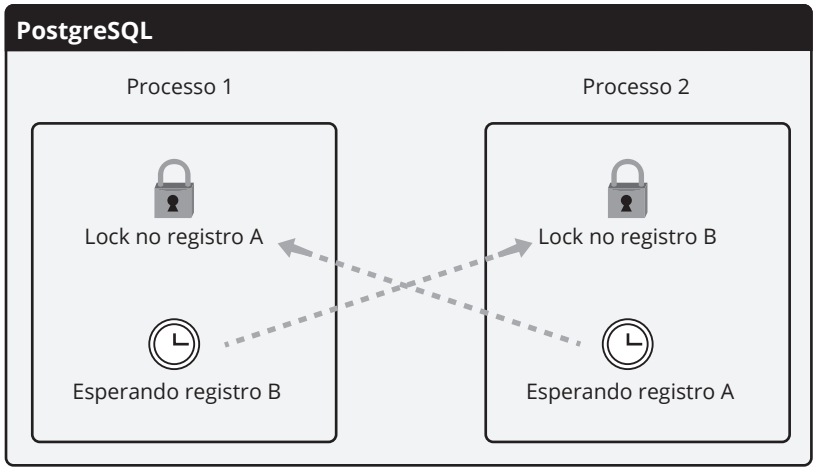


Figura 7.7
Deadlock.

O PostgreSQL detecta deadlocks, verificando a ocorrência deles em um intervalo de tempo definido pelo parâmetro deadlock_timeout, por padrão a cada 1 segundo. Se um deadlock for identificado, o PostgreSQL escolherá um dos processos como "vítima" e a respectiva operação será abortada. Nesses casos poderá ser vista a seguinte mensagem no log:

```
LOG:  process 16192 detected deadlock while waiting for ShareLock on transaction
837 after 1005.635 ms
STATEMENT:  UPDATE grupos SET nome='Z' WHERE id = 1;
ERROR:  deadlock detected
DETAIL:  Process 16192 waits for ShareLock on transaction 837; blocked by process
15186.
Process 15186 waits for ShareLock on transaction 838; blocked by process 16192.
```

O valor do parâmetro deadlock_timeout geralmente é razoável para a maioria dos usos. Caso estejam ocorrendo muitos locks e deadlocks, seu valor pode ser baixado para ajudar na depuração do problema, mas isso tem um preço, já que o algoritmo de busca por deadlocks é relativamente custoso.

Se deadlocks estão ocorrendo com frequência, então programas, scripts e procedures devem ser revistos e verificada a ordem que estão acessando registros.

Nas versões anteriores do PostgreSQL, havia algumas situações envolvendo Foreign Keys que podiam gerar deadlocks. Isso foi corrigido nas versões mais novas.

Tuning de queries

Depois de analisados e descartados problemas de bloqueios e as alternativas de diminuição do volume de dados retornados e/ou processados que não são o caso ou não podem ser aplicadas, resta analisar a query mais profundamente. Analisar a consulta será necessário também nas situações em que esta não parece lenta quando executada isoladamente (entre 0,5 ou 1 segundo), mas que por ser executada dezenas de milhares de vezes acaba gerando um gargalo.

Para entender como o banco está processando a consulta, devemos ver o Plano de Execução escolhido pelo SGBD para resolver aquela query. Para fazermos isso, usamos o comando *EXPLAIN*, conforme o exemplo a seguir:

```
bench=# EXPLAIN
SELECT *
FROM pgbench_accounts a
INNER JOIN pgbench_branches b ON a.bid=b.bid
INNER JOIN pgbench_tellers t ON t.bid=b.bid
WHERE a.bid=56;

                QUERY PLAN
-----------------------------------------------------------
 Nested Loop  (cost=0.00..296417.62 rows=1013330 width=813)
   ->  Seq Scan on pgbench_accounts a  (cost=0.00..283731.12 rows=101333 width=97)
       Filter: (bid = 56)
   ->  Materialize  (cost=0.00..19.90 rows=10 width=716)
       ->  Nested Loop  (cost=0.00..19.85 rows=10 width=716)
           ->  Seq Scan on pgbench_branches b  (cost=0.00..2.25 rows=1 width=364)
               Filter: (bid = 56)
           ->  Seq Scan on pgbench_tellers t  (cost=0.00..17.50 rows=10 width=352)
               Filter: (bid = 56)
```

O EXPLAIN nos mostra o Plano de Execução da query e os custos estimados. O primeiro nó indica o custo total da query.

```
Sort  (cost=0.00..296417.62 rows=1013330 width=813)
           |     |              |       |____ Tamanho estimado de cada registro
           |     |              |_____ Número de registros estimado
           |     |______ Custo total estimado da query
           |____  Custo inicial estimado para retornar o primeiro registro
```

Cada linha no plano com um -> indica uma operação. As demais são informações adicionais.

Todas as informações do EXPLAIN sem parâmetros são estimativas. Para obter o tempo real, ele deve executar a query de fato, através do comando *EXPLAIN ANALYZE*:

```
bench=# EXPLAIN (ANALYZE)
SELECT *
FROM pgbench_accounts a
INNER JOIN pgbench_branches b ON a.bid=b.bid
INNER JOIN pgbench_tellers t ON t.bid=b.bid
WHERE a.bid=56;
                                QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..296417.62 rows=1013330 width=813) (actual time=26750.84
5..44221.807 rows=1000000 loops=1)
   ->  Seq Scan on pgbench_accounts a  (cost=0.00..283731.12 rows=101333 width=97
) (actual time=26749.187..31234.702 rows=100000 loops=1)
       Filter: (bid = 56)
       Rows Removed by Filter: 9900000
   ->  Materialize  (cost=0.00..19.90 rows=10 width=716) (actual time=0.004..0.04
3 rows=10 loops=100000)
   ->  Nested Loop  (cost=0.00..19.85 rows=10 width=716) (actual time=1.593..1.88
6 rows=10 loops=1)
   ->  Seq Scan on pgbench_branches b  (cost=0.00..2.25 rows=1 width=364) (actual
time=0.156..0.167 rows=1 loops=
       Filter: (bid = 56)
       Rows Removed by Filter: 99
   ->  Seq Scan on pgbench_tellers t  (cost=0.00..17.50 rows=10 width=352) (actual
time=1.404..1.617 rows=10 loops=1)
       Filter: (bid = 56)
       Rows Removed by Filter: 990
 Total runtime: 48022.546 ms
```

Agora podemos ver informações de tempo. No primeiro nó temos o tempo de execução, aproximadamente 44s, e o número de registros real: 1 milhão. O atributo loops indica o número de vezes em que a operação foi executada. Em alguns nós, como alguns joins, será maior que 1 e o número de registros e o tempo são mostrados por iteração, devendo-se multiplicar tais valores pela quantidade de loops para chegar ao valor total.

Importante: o comando *EXPLAIN ANALYZE* executa de fato a query. Logo, se for feito com um UPDATE ou DELETE, tal ação será realizada de fato, alterando a base da dados. Para somente analisar queries que alteram dados, você pode fazer o seguinte:

```
BEGIN TRANSACTION;
    EXPLAIN ANALYZE UPDATE …
ROLLBACK;
```

Outro parâmetro útil do EXPLAIN é o BUFFERS, que mostra a quantidade de blocos, 8kB por padrão, encontrados no shared buffers ou que foram lidos do disco ou, ainda, de arquivos temporários que foram necessários ser gravados em disco.

```
bench=# EXPLAIN (ANALYZE, BUFFERS)  SELECT *
FROM pgbench_accounts a
...
                                QUERY PLAN
-------------------------------------------------------------------------------------------
Nested Loop  (cost=0.00..296417.62 rows=1013330 width=813) (actual time=29946.100..
48764.583 rows=1000000 loops=1)
 Buffers: shared hit=519 read=158218
 ->  Seq Scan on pgbench_accounts a  (cost=0.00..283731.12 rows=101333 width=97) (a
ctual time=29945.373..34818.143 rows=100000 loops=1)
     Filter: (bid = 56)
     Rows Removed by Filter: 9900000
      Buffers: shared hit=513 read=158218
  ->  Materialize  (cost=0.00..19.90 rows=10 width=716) (actual time=0.004..0.046 r
ows=10 loops=100000)
      Buffers: shared hit=6
```

Agora a saída mostra os dados shared hit e read. No nó superior, que mostra o total, vemos que foram encontrados no cache do PostgreSQL, shared hit, 519 blocos (~4MB) e lidos do disco, read, 158218 blocos (~1,2GB).

Índices

Nos exemplos com o EXPLAIN, vimos nos planos de execução várias operações de SEQ SCAN. Essa operação varre a tabela toda e é executada quando não há um índice que atenda a consulta, ou porque o Otimizador acredita que terá de ler quase toda a tabela de qualquer jeito, sendo um overhead desnecessário tentar usar um índice. Especial atenção deve ser dada a situações em que não há índice útil, mas poderia haver um.

Índices são estruturas de dados paralelas às tabelas que têm a função de tornar o acesso aos dados mais rápido. Em um acesso a dados sem índices, é necessário percorrer todo um conjunto de dados para verificar se uma condição é satisfeita. Índices são estruturados de forma a serem necessárias menos comparações para localizar um dado ou determinar que ele não existe. Existem vários tipos de índices, sendo o mais comum o BTree, baseado em uma estrutura em árvore. No PostgreSQL é o tipo padrão, e se não informado no comando *CREATE INDEX*, ele será assumido.

É comum haver confusão entre índices e restrições, ou constraints. Muitos desenvolvedores assumem que são a mesma coisa, criam suas constraints e não se preocupam mais com o assunto.

Índices e constraints são coisas distintas. Índices, como foi dito, são estruturas de dados, ocupam espaço em disco e têm função de melhoria de desempenho. Constraints são regras, restrições impostas aos dados. A confusão nasce porque as constraints do tipo PRIMARY KEY e UNIQUE de fato criam índices implicitamente para garantir as propriedades das constraints. O problema reside com as FOREIGN KEY.

Para uma FK, o PostgreSQL não cria índices automaticamente. A necessidade de um índice nesses casos deve ser analisada e este criado manualmente. Normalmente chaves estrangeiras são colunas boas candidatas a terem índice.

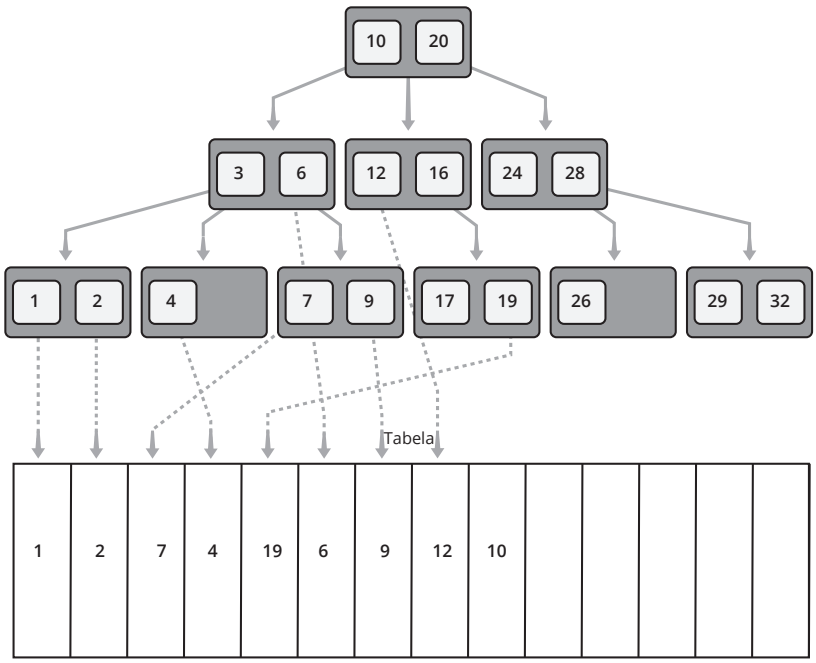


Figura 7.8
Índice Btree.

Índices simples

No exemplo de plano de execução recém-mostrado, vemos um SEQ SCAN com um filter. Esse é o candidato perfeito para criarmos um índice. Isso significa que o banco teve de varrer a tabela e aplicar uma condição (bid = 56) para remover os registros que não atendem esse critério.

```
->  Seq Scan on pgbench_accounts a  (cost=0.00..283731.12 rows=101333 width=97) (a
ctual time=29945.373..34818.143 rows=100000 loops=1)
    Filter: (bid = 56)
    Rows Removed by Filter: 9900000
```

Devemos então criar um índice na coluna bid e testar novamente.

```
bench=# CREATE INDEX idx_accounts_bid ON pgbench_accounts(bid);
```

> Índices criam locks nas tabelas que podem bloquear escritas. Se for necessário criá-lo em horário de uso do sistema, pode-se usar CREATE INDEX CONCURRENTLY, que usará um mecanismo menos agressivo de locks, porém demorará mais para executar.

Agora, executando a query novamente com EXPLAIN (ANAYZE), vemos o seguinte plano:

```
                                QUERY PLAN
-------------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..16964.96 rows=1013330 width=813) (actual time=44.298..
13742.038 rows=1000000 loops=1)
   ->  Index Scan using idx_accounts_bid on pgbench_accounts a  (cost=0.00..4278.
46 rows=101333 width=97) (actual time=43.903..1002.919 rows=100000)
         Index Cond: (bid = 56)
   ->  Materialize  (cost=0.00..19.90 rows=10 width=716) (actual time=0.004..0.04
2 rows=10 loops=100000)
   ->  Nested Loop  (cost=0.00..19.85 rows=10 width=716) (actual time=0.321..0.57
9 rows=10 loops=1)
   ->  Seq Scan on pgbench_branches b  (cost=0.00..2.25 rows=1 width=364) (actual
 time=0.117..0.128 rows=1 loops=1)
         Filter: (bid = 56)
         Rows Removed by Filter: 99
   ->  Seq Scan on pgbench_tellers t  (cost=0.00..17.50 rows=10 width=352) (actual
 time=0.147..0.326 rows=10 loops=1)
         Filter: (bid = 56)
         Rows Removed by Filter: 990
 Total runtime: 17445.034 ms
```

O custo do nó caiu de 28 mil para cerca de 4 mil. O tempo para o nó que era de quase 35s caiu para 10s e o tempo total da query de 44s para 13s.

Índices compostos

Outra possibilidade é a criação de índices compostos, com múltiplas colunas. Veja o seguinte exemplo:

```
bench=# EXPLAIN (ANALYZE)
SELECT *
FROM pgbench_tellers t
INNER JOIN pgbench_branches b ON t.bid=b.bid
WHERE t.bid = 15 AND t.tbalance = 0;
                                QUERY PLAN
-------------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..22.35 rows=10 width=716) (actual time=0.225..0.852 row
s=10 loops=1)
   ->  Seq Scan on pgbench_branches b  (cost=0.00..2.25 rows=1 width=364) (actual
 time=0.142..0.170 rows=1 loops=1)
         Filter: (bid = 15)
         Rows Removed by Filter: 99
   ->  Seq Scan on pgbench_tellers t  (cost=0.00..20.00 rows=10 width=352) (actua
l time=0.055..0.421 rows=10 loops=1)
         Filter: ((bid = 15) AND (tbalance = 0))
         Rows Removed by Filter: 990
 Total runtime: 1.066 ms
```

Há uma dupla condição filtrando os resultados. Podemos testar um índice envolvendo as duas colunas:

```
bench=# CREATE INDEX idx_branch_bid_tbalance ON pgbench_tellers(bid,tbalance);
```

Executando novamente a query temos o seguinte plano:

```
                                QUERY PLAN
-------------------------------------------------------------------------------------
 Nested Loop  (cost=4.35..11.85 rows=10 width=716) (actual time=0.309..0.741 rows
 =10 loops=1)
   ->  Seq Scan on pgbench_branches b  (cost=0.00..2.25 rows=1 width=364) (actual
 time=0.193..0.225 rows=1 loops=1)
         Filter: (bid = 15)
         Rows Removed by Filter: 99
   ->  Bitmap Heap Scan on pgbench_tellers t  (cost=4.35..9.50 rows=10 width=352)
  (actual time=0.085..0.191 rows=10 loops=1)
       Recheck Cond: ((bid = 15) AND (tbalance = 0))
   ->  Bitmap Index Scan on idx_branches_bid_tbalance  (cost=0.00..4.35 rows=10 w
 idth=0) (actual time=0.062..0.062 rows=10 loops=1)
       Index Cond: ((bid = 15) AND (tbalance = 0))
 Total runtime: 0.976 ms
```

O custo caiu de 22 para 11. Devemos sempre considerar o custo – uma métrica do PostgreSQL para estimar o custo das operações, em vez de somente o tempo, que pode variar conforme a carga da máquina ou os dados estarem ou não no cache.

Índices parciais

Outra excelente ferramenta do PostgreSQL são os índices parciais. Índices parciais são índices comuns, podem ser simples ou compostos, que possuem uma cláusula WHERE. Eles se aplicam a um subconjunto dos dados e podem ser muito mais eficientes com cláusulas SQL que usem o mesmo critério.

Suponha que exista uma query muito executada, faz parte da tela inicial dos usuários de alguma aplicação.

```
bench=# EXPLAIN ANALYZE
SELECT *
FROM pgbench_accounts
WHERE bid=90 AND abalance > 0;

                                QUERY PLAN
-------------------------------------------------------------------------------------
 Index Scan using idx_accounts_bid on pgbench_accounts  (cost=0.00..4336.39 rows=
 1 width=97) (actual time=0.358..60.152 rows=8 loops=1)
   Index Cond: (bid = 90)
   Filter: (abalance > 0)
   Rows Removed by Filter: 99992
 Total runtime: 60.370 ms
```

A consulta usa o índice da coluna bid, porém a segunda condição com abalance precisa ser filtrada. Uma alternativa é o uso de índices compostos, como vimos anteriormente, porém nesse caso supomos que o critério abalance > 0 não mude, é sempre esse.

Dessa form podemos criar um índice específico parcial para esta consulta:

```
bench=# CREATE INDEX idx_accounts_bid_parcial
ON pgbench_accounts(bid) WHERE abalance > 0;
```

Criamos um novo índice na coluna bid mas agora filtrando apenas as contas positivas. Executando novamente a query:

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using idx_accounts_bid_parcial on pgbench_accounts  (cost=0.00..4.27 r
 ows=1 width=97) (actual time=0.246..0.307 rows=8 loops=1)
   Index Cond: (bid = 90)
 Total runtime: 0.705 ms
```

O custo cai drasticamente de 4 mil para 4! Mas não saia criando índices parciais indiscriminadamente. Há custos de espaço, de inserção e organização dos índices. Cada situação deve ser analisada e medido o custo-benefício.

Índices parciais são especialmente úteis com colunas boolean, sobre a qual índices BTree não são eficazes, em comparação com NULL. Exemplos:

```
CREATE INDEX idx_conta_ativa ON conta(idconta) WHERE ativa = 'true';
CREATE INDEX idx_conta_freq ON conta(idconta) WHERE data IS NULL;
```

Índices com expressões

Um erro também comum é usar uma coluna indexada, porém ao escrever a query, aplicar alguma função ou expressão sobre a coluna. Por exemplo:

```
bench=# EXPLAIN SELECT *
FROM pgbench_accounts
WHERE COALESCE(bid,0) = 56;

                                QUERY PLAN
-------------------------------------------------------------------------
Seq Scan on pgbench_accounts  (cost=0.00..283752.00 rows=50000 width=97)
   Filter: (COALESCE(bid, 0) = 56)
```

A coluna bid é indexada, mas quando foi aplicada uma função sobre ela, foi feito SEQ SCAN. Para usar um índice, nesse caso é necessário criar o índice com a função aplicada.

```
bench=# CREATE INDEX idx_accounts_bid_coalesce
ON pgbench_accounts( COALESCE(bid,0) );
```

Verificando o plano novamente:

```
                                QUERY PLAN
-------------------------------------------------------------------------------------
 Bitmap Heap Scan on pgbench_accounts  (cost=828.63..106644.01 rows=50000 width=97)
   Recheck Cond: (COALESCE(bid, 0) = 56)
   ->  Bitmap Index Scan on idx_accounts_bid_coalesce  (cost=0.00..816.13 rows=500
00 width=0)
   Index Cond: (COALESCE(bid, 0) = 56)
```

Esse equívoco em esquecer de indexar a coluna com a função ou expressão que será aplicada pela query é bastante comum com o uso das funções UPPER() e LOWER().

Funções (Store Procedures)

O PostgreSQL implementa store procedures através de funções. Uma grande característica do PostgreSQL é permitir a criação de funções em diversas linguagens. A principal delas é a pl/pgsql.

Utilizar funções pode trazer grandes benefícios de desempenho, dependendo do tipo de processamento. Se para realizar uma operação complexa uma aplicação executar várias queries no banco, obter os resultados, processá-los e depois submeter novas queries ao banco, isso envolverá boa quantidade de recursos e tempo. Se for possível colocar todo esse processamento dentro de uma única função, certamente haverá ganhos de desempenho.

Escrevendo funções, pode-se evitar o custo de IPC, comunicação entre processos, o custo do tráfego pela rede e o custo do parse da query entre múltiplas chamadas.

Por outro lado, adicionar regras de negócio no Banco de Dados não é uma boa prática de engenharia de software.

8

Desempenho – Tópicos sobre configuração e infraestrutura

objetivos

Conhecer alternativas de soluções para problemas de desempenho relacionados ao ajuste fino dos parâmetros de configuração do PostgreSQL; Analisar a infraestrutura de hardware e software.

conceitos

Full-Text Search; Indexadores de Documentos; Índices GIN; Operadores de Classe; Cluster; Particionamento; Memória de Ordenação; Otimizador; Escalabilidade; Filesysteme RAID.

Busca em texto

Antes de tratar especificamente de questões de desempenho relacionadas com a configuração do PostgreSQL ou com a infraestrutura de hardware e software, veremos ainda questões relativas a busca textual e alternativas de organização de tabelas muito grandes.

LIKE

Uma situação que comumente gera problemas de desempenho em queries é o operador LIKE/ILIKE. O LIKE permite pesquisar um padrão de texto em outro conteúdo de texto, normalmente uma coluna. ILIKE tem a mesma função mas não faz diferença entre maiúsculas e minúsculas (case insensitive).

Ao analisar uma querie com EXPLAIN, podemos esbarrar com uma coluna do tipo texto que está indexada mas cujo índice não está sendo utilizando pela query que contém o LIKE. Isso pode acontecer porque no PostgreSQL é preciso utilizar um operador especial no momento da criação do índice para que operações com LIKE possam utilizá-lo.

Esse operador depende do tipo da coluna:

varchar	varchar_pattern_ops
Char	bpchar_pattern_ops
Text	text_pattern_ops

Por exemplo, para criar um índice que possa ser pesquisado pelo LIKE, simplesmente use a seguinte forma, supondo que a coluna seja varchar:

```
curso=#     CREATE INDEX idx_like ON times(nome varchar_pattern_ops);
```

Um outro motivo para o PostgreSQL não utilizar os índices numa query com LIKE é se for usado % no ínicio da string, significando que pode haver qualquer coisa antes. Por exemplo:

```
curso=#     SELECT * FROM times WHERE nome LIKE '%Herzegovina%';
```

Essa cláusula nunca usará índice, mesmo com o operador de classe sempre varrendo a tabela inteira.

Full-Text Search

Para pesquisas textuais mais eficientes e mais complexas do que aquelas que fazem uso do LIKE, o PostgreSQL disponibiliza os recursos FTS – Full-Text Search. O FTS permite busca por frases exatas, uso de operadores lógicos | (or) e & (and), ordenação por relevância e outras opções.

O uso do FTS requer, no entanto, preparação prévia. Primeiro é necessário preparar a tabela para utilizar esse recurso, inserindo uma coluna do tipo tsvector:

```
curso=# ALTER TABLE times ADD COLUMN historia_fts tsvector;
```

Em seguida, deve-se copiar e converter o conteúdo da coluna que contém o texto original para a nova coluna "vetorizada":

```
curso=# UPDATE times SET historia_fts = to_tsvector('portuguese', historia);
```

Finalmente, cria-se um índice do tipo GIN na coluna vetorizada:

```
curso=# CREATE INDEX idx_historia_fts ON times USING GIN(historia_fts);
```

Desse ponto em diante pode-se usar o FTS, bastando aplicar o operador @@ e a função ts_query:

```
curso=# SELECT nome, historia
      FROM times
   WHERE historia_fts @@ to_tsquery('camp & mund');
```

Esse exemplo é bem simples, dando apenas uma noção superficial de como funciona a solução de Full-Text Search do PostgreSQL.

Softwares indexadores de documentos

Se precisamos fazer buscas complexas, estilo Google, em uma grande quantidade de documentos de texto, a melhor alternativa talvez seja utilizar softwares específicos, como o Lucene e SOLR, ambos open source.

Esses softwares leem os dados do banco periodicamente e criam índices onde são feitas as buscas. Como normalmente os conteúdos de documentos mudam pouco, essa estratégia é melhor do que acessar o banco a cada vez.

Organização de tabelas grandes

Cluster de tabela

Se tivermos uma tabela grande, muito usada, já indexada, e ainda assim com a necessidade de melhorar o acesso a ela, uma possibilidade é o comando *CLUSTER*, abordado na sessão 6. Essa operação irá ordenar os dados da tabela fisicamente segundo um índice que for informado. É especialmente útil quando são lidas faixas de dados em um intervalo.

```
bench=# CLUSTER pgbench_accounts USING idx_accounts_bid;
```

Particionamento de tabelas

Se uma tabela ficar tão grande que as alternativas até aqui apresentadas não estejam ajudando a melhorar o desempenho das queries, uma a ser considerada é o particionamento. Esse procedimento divide uma tabela em outras menores, baseado em algum campo que você definir, em geral um ID ou uma data. Isso pode trazer benefícios de desempenho, já que as queries farão varreduras em tabelas menores ou índices menores. No PostgreSQL, o particionamento é feito usando herança de tabelas.

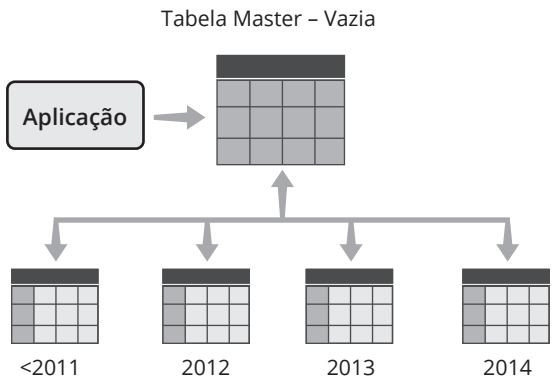


Figura 8.1 Tabela particionada.

Os passos necessários para realizar um particionamento, usando um exemplo de uma tabela financeira que vai ser particionada por anos, são:

- Criar tabela máster;
- Criar tabelas filhas herdando colunas da máster;
- Adicionar check constraint em cada filha para garantir faixa;
- Criar trigger na tabela master que direciona para faixa;
- Criar índice na coluna chave do particionamento nas filhas.

Criar uma tabela principal, que não terá dados:

```
curso=# CREATE TABLE item_financeiro (
                    iditem int,
                    data timestamp,
                    descricao varchar(50),
                    valor numeric(10,2)
    );
```

Criar as tabelas filhas, herdando as colunas da tabela principal:

```
curso=# CREATE TABLE item_financeiro_2012 () INHERITS (item_financeiro);
curso=# CREATE TABLE item_financeiro_2013 () INHERITS (item_financeiro);
curso=# CREATE TABLE item_financeiro_2014 () INHERITS (item_financeiro);
```

Adicionar uma CHECK constraint em cada tabela filha, ou partição, para aceitar dados apenas da faixa certa para a partição:

```
curso=# ALTER TABLE item_financeiro_2012
                    ADD CHECK (data >= '2012-01-01' AND data < '2013-01-01');
            ...
```

Criar uma trigger na tabela principal, que direciona os dados para as filhas.

```
curso=#
CREATE OR REPLACE FUNCTION itemfinanceiro_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    IF (NEW.data >= '2012-01-01' AND NEW.data < '2013-01-01') THEN
            INSERT INTO item_financeiro_2012 VALUES (NEW.*);
    ELSIF (NEW.data >= '2013-01-01' AND NEW.data < '2014-01-01') THEN
            INSERT INTO item_financeiro_2013 VALUES (NEW.*);
    ELSIF (NEW.data >= '2014-01-01' AND NEW.data < '2015-01-01') THEN
            INSERT INTO item_financeiro_2014 VALUES (NEW.*);
    ELSE
            RAISE EXCEPTION 'Data fora de intervalo válido';
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
curso=#
    CREATE TRIGGER t_itemfinanceiro_insert_trigger
            BEFORE INSERT ON item_financeiro
            FOR EACH ROW EXECUTE PROCEDURE itemfinanceiro_insert_trigger();
```

Apesar de não ser obrigatório para realizar o particionamento, recomenda-se criar índices na coluna chave do particionamento:

```
curso=# CREATE INDEX idx_data_2012 ON item_financeiro_2012(data);
...
```

Após ter inserido ou migrado os dados para as tabelas particionadas, é importante executar uma atualização de estatísticas.

Agora, usando o EXPLAIN para ver o plano de uma consulta à tabela a principal, podemos ver que foi necessário apenas acessar uma partição:

```
curso=# curso=# EXPLAIN ANALYZE
SELECT *
  FROM item_financeiro
WHERE data = DATE '2013-03-22';

                                  QUERY PLAN
------------------------------------------------------------------------------
Result  (cost=0.00..0.27 rows=2 width=96) (actual time=0.063..0.095 rows=1 loops=1)
 ->  Append  (cost=0.00..0.27 rows=2 width=96) (actual time=0.048..0.067 rows=1 loo
ps=1)
 ->  Seq Scan on item_financeiro  (cost=0.00..0.00 rows=1 width=162) (actual time=0
.007..0.007 rows=0 loops=1)
     Filter: (data = '2013-03-22'::date)
 ->  Index Scan using idx_data_2013 on item_financeiro_2013 item_financeiro  (cost=
0.00..0.27 rows=1 width=30) (actual time=0.0
     Index Cond: (data = '2013-03-22'::date)
```

Procedimentos de manutenção

Na sessão 6 foram apresentados inúmeros procedimentos que devem ser executados para garantir o bom funcionamento do PostgreSQL. Dentre eles, vale destacar 3 que podem impactar significativamente o desempenho do banco:

Vacuum

A operação de Vacuum pode afetar o desempenho das tabelas, especialmente se não estiver sendo executada, ou sendo executada com pouca frequência.

O exemplo a seguir envolve uma tabela que possui apenas um registo, mas que tem custo de 25874. Essa situação pode ser explicada pelo fato de o autovacuum estar desabilitado. Assim, a tabela está cheia de dead tuples, versões antigas de registros que devem ser eliminadas, mas que estão sendo varridos quando a tabela passa por um SCAN.

- Vacuum
- Estatísticas
- "Índices Inchados"

```
bench=# EXPLAIN ANALYZE SELECT * FROM contas;
                                  QUERY PLAN
------------------------------------------------------------------------------------
Seq Scan on contas  (cost=0.00..25874.00 rows=1000000 width=97) (actual time=117.78
9..173.880 rows=1 loops=1)
```

Ao analisar uma query específica, executar um Vacuum manual nas tabelas envolvidas pode ajudar a resolver alguma questão relacionada a dead tuples.

Estatísticas

Uma query pode estar escolhendo um plano de execução ruim por falta de estatísticas, ou por estatísticas insuficientes. Os procedimentos para gerar e atualizar estatísticas foram vistos anteriormente com mais detalhes, mas vale destacar que no exemplo recém-citado, ilustrando a não execução do autovacuum, também não está sendo feito o autoanalyze. Assim, o Otimizador acredita que há 1000000 registros na tabela, quando na verdade há apenas um registro válido.

Da mesma forma que acontece em relação ao Vacuum, ao analisar uma query em particular, executar o ANALYZE nas tabelas envolvidas pode ajudar o Otimizador a escolher um plano de execução mais realista.

'Índices inchados'

Bloated indexes, ou índices inchados, são índices com grande quantidade de dead tuples. Executar o comando *REINDEX* para reconstrução de índices nessa situação é apenas mais um exemplo de como as atividades de manutenção podem ser importantes para preservar e garantir o desempenho do banco.

Configurações para desempenho

- work_mem
- shared_buffers
- effective_cache_size
- Checkpoints
- Parâmetros de Custo
- statement_timeout

Até agora estivemos analisando situações que impactam o desempenho do banco e que estão diretamente relacionadas com aspectos das aplicações para acesso e manipulação de dados. Foram trabalhadas situações envolvendo principalmente queries e volumes de dados envolvidos, incluindo também a criação de estruturas de apoio, tais como índices, partições etc.

Uma abordagem diferente é buscar ajustar a configuração do PostgreSQL em situações específicas. Até porque é importante frisar que o tuning através dos parâmetros de configuração só deve ser feito apenas quando se está enfrentando problemas.

work_mem

É a quantidade de memória que um processo pode usar para operações envolvendo ordenação e hash – como ORDER BY, DISTINCT, IN e alguns algoritmos de join escolhidos pelo Otimizador. Se a área necessária por uma query for maior do que o especificado através deste parâmetro a operação será feita em discos, através da criação de arquivos temporários.

Ao analisar uma query em particular executando o EXPLAIN (ANALYZE, BUFFERS), um resultado a ser observado é se há arquivos temporários sendo criados. Se não for possível melhorar a query restringindo os dados a serem ordenados, deve-se estudar aumentar o work_mem.

Se esse parâmetro estiver muito baixo, muitas queries podem ter que ordenar em disco e isso causará grande impacto negativo no tempo de execução dessas consultas. Por outro lado, se esse valor for muito alto, centenas de queries simultâneas poderiam demandar memória, resultando na alocação de uma quantidade grande demais de memória, a ponto de até esgotar a memória disponível.

O valor default, 1MB, é muito modesto para queries complexas. Dependendo da sua quantidade de memória física, deve-se aumentá-lo para 4MB, 8MB ou 16MB e acompanhar se está ocorrendo ordenação em disco.

Lembre-se também que se sua base tiver muitos usuários, pode-se definir um work_mem maior apenas para as queries que mais estão gerando arquivos temporários, ou apenas para uma base específica, conforme foi visto na sessão de aprendizagem sobre configuração.

Pode-se verificar se está ocorrendo ordenação em disco através da view do catálogo pg_stat_database, mas esta é uma informação geral para toda a base. Através do parâmetro log_temp_files é possível registrar no log do PostgreSQL toda vez que uma ordenação em disco ocorrer, ou que passe de determinado tamanho. Essa informação é inclusive mostrada nos relatório do pgBadger.

O valores para log_temp_files são:

```
0   todos os arquivos temporários serão registrados no Log
-1  nenhum arquivo temporário será registrado
N   Tamanho mínimo em KB. Arquivos maiores do que N serão registrados
```

Poderão ser vistas mensagens como estas no log:

```
user=curso,db=curso LOG:  temporary file: path “base/pgsql_tmp/pgsql_tmp23370.25”,
size 269557760
user=curso,db=curso STATEMENT: SELECT ...
```

shared_buffers

Conforme já explicado, Shared Buffers é a área de cache de dados do PostgreSQL. Como o PostgreSQL tira proveito também do page cache do SO, não é correto assumir que aumentar o valor de shared_buffers será sempre melhor.

Se estiver enfrentando problemas de desempenho em queries que estão acessando muito disco, é aconselhável primeiro analisar as opções relacionadas com problemas originados pela aplicação. Se nenhuma delas tiver efeito, aí sim devemos analisar o aumento do parâmetro shared_buffers.

Nos casos em que o servidor não é dedicado ao Banco de Dados (o que já não é recomendável), não há garantia de que o page cache do SO contenha dados do banco (outros softwares poderão estar colocando dados nesse cache). Nesse cenário, aumentar o shared_buffers é uma possibilidade para forçar mais memória do sistema para o PostgreSQL.

Calcular a taxa de acerto no shared buffer através da view pg_stat_database, como já exemplificado anteriormente, pode ajudar a tomar uma decisão sobre aumentar essa área. É difícil dizer o que é um percentual de acerto adequado, mas se for uma aplicação transacional de uso frequente deve-se com certeza absoluta buscar trabalhar com taxas superiores a 90% ou 95% de acerto. Valores abaixo disso devem ser encarados com preocupação.

```
postgres=# SELECT datname,
           CASE WHEN blks_hit = 0 THEN 0
           ELSE (( blks_hit / (blks_read + blks_hit)::float) * 100)::float
           END as cache_hit
           FROM pg_stat_database
           WHERE datname NOT LIKE ‘template_’
           ORDER BY 2;
```

```
 datname  | cache_hit
----------+-----------
 bench    |     19.01
 postgres |     98.98
 rh       |     99.74
 projetox |     99.76
 curso    |     99.77
```

Figura 8.2
Taxa de acerto no shared buffers por base.

Também pode ser útil analisar o conteúdo do shared buffer. Ver quais objetos mais possuem buffers em cache pode, por exemplo, mostrar se há alguma tabela menos importante no sistema ocupando muito espaço, ou seja, sendo muito referenciada por alguma operação. Pode-se localizar o programa que está "poluindo" o shared buffer e tentar algo como mudá-lo de horário ou considerar reescrevê-lo.

effective_cache_size

O parâmetro effective_cache_size não define o tamanho de um recurso do PostgreSQL. Ele é apenas uma informação, uma estimativa, do tamanho total de cache disponível, shared_buffer + page cache do SO. Essa estimativa pode ser usada pelo Otimizador para decidir se um determinado índice cabe na memória ou se a tabela deve ser varrida, daí sua importância.

Para defini-la, some o valor do parâmetro shared_buffers ao valor observado da memória sendo usada para cache em seu servidor. O tamanho do cache pode ser facilmente consultado com free, mas também com top, vmstat e sar.

Checkpoints

A operação de Checkpoint é uma operação de disco cara. A frequência com que ocorrerão checkpoints é definida pelos parâmetros checkpoints_segments e checkpoints_timeout, melhor detalhados na tabela a seguir:

Parâmetro	Descrição	Valor
checkpoint_segments	Número de segmentos de log de transação (arquivos de 16MB) pre-enchidos para disparar o processo de Checkpoint	O valor padrão de 3 é muito baixo, podendo disparar Checkpoints com muita frequência e assim sobrecarregar o acesso a disco. Um valor muito alto tornará a recuperação após um crash muito demorada e ocupará N*16MB de espaço em disco. Inicie com um valor entre 8 e 16.
checkpoint_timeout	Intervalo de tempo máximo com que ocorrerão Checkpoints.	Um valor muito baixo ocasionará muitos Checkpoints, enquanto um valor muito alto causará uma recuperação pós-crash demorada. O valor padrão de 5min é adequado na maioria das vezes.

Tabela 8.1
Os parâmetros checkpoints_segments e checkpoints_timeout.

É possível verificar a ocorrência de checkpoints (se está muito frequente, quanto tempo está levando e a quantidade de dados sendo gravada) através de registros no log. Para isso, deve-se ligar o parâmetro log_checkpoint. Exemplo de mensagem registrando a ocorrência de um checkpoint pode ser visto a seguir:

```
LOG:  checkpoint complete: wrote 5737 buffers (1.1%); 0 transaction log file(s)
added, 0 removed, 0 recycled; write=127.428 s, sync=0.202 s, total=127.644 s; sy
nc files=758, longest=0.009 s, average=0.000
```

Parâmetros de custo

A configuração seq_page_cost é uma constante que estima o custo para ler uma página sequencialmente do disco. O valor padrão é 1 e todos as outras estimativas de custo são relativas a esta.

O parâmetro random_page_cost é uma estimativa para se ler uma página aleatória do disco. O valor padrão é 4. Valores mais baixos de random_page_cost induzem o Otimizador a preferir varrer índices, enquanto valores mais altos farão o Otimizador considerá-los mais caros.

Lembre-se de que não há fórmula pronta: todas as alterações devem ser testadas. Pode-se configurar esses parâmetros apenas para uma sessão e executar diversas queries para verificar o comportamento resultante, aplicando definições globalmente no *postgresql.conf* somente se os resultados forem satisfatórios.

Esse valor pode ser aumentado ou diminuído, dependendo da velocidade de seus discos e do comportamento do cache. Em ambientes onde há bastante RAM, igual ou maior ao tamanho do banco, pode-se testar igualar o valor ao de seq_page_cost. Veja que não faz sentido ele ser menor do que seq_page_cost.

Se o ambiente está fortemente baseado em cache, com bastante memória disponível, pode-se inclusive baixar os dois parâmetros quase ao nível de operações de CPU, utilizando, por exemplo, o valor 0.05.

É possível alterar esses parâmetros para um tablespace em particular. Isso pode fazer sentido em sistemas com discos de estado sólido, SSD, definindo um random_page_cost de 1.5 ou 1.1 apenas para tablespaces nesses discos.

```
curso=# SET random_page_cost = 1;
    curso=# EXPLAIN SELECT ...
```

statement_timeout

Uma medida mais drástica para evitar queries que estão sobrecarregando o banco é definir um tempo máximo de execução a partir do qual o comando será abortado. O parâmetro statement_timeout define esse tempo máximo em milissegundos.

Normalmente as camadas de pool possuem um timeout, não sendo necessário fazê-lo no PostgreSQL. Também é possível definir esse timeout apenas para um usuário ou base específica que esteja apresentando problemas.

Por exemplo, para definir um timeout de 30 segundos para todas as queries na base curso:

```
postgres=# ALTER DATABASE curso SET statement_timeout = 30000;
```

Se uma query ultrapassar esse tempo, será abortada com a seguinte mensagem:

```
ERROR:  canceling statement due to statement timeout
```

Infraestrutura e desempenho

Vamos analisar as seguintes possibilidades:

- Escalabilidade Horizontal;
- Balanceamento de Carga;
- Memória;
- Filesystems;
- Armazenamento:
  - RAID;
  - Separação de Funções.

- Virtualização;
- Processadores;
- Redes e Serviços.

Quando esgotadas as possibilidades de melhorias na Aplicação e nas Configurações do PostgreSQL e SO, temos de começar a analisar mudanças de infraestrutura, tanto de software quanto de hardware. Essas mudanças podem envolver adicionar componentes, trocar tecnologias, crescer verticalmente – mais memória, mais CPU, mais banda etc. – ou crescer horizontalmente – adicionar mais instâncias de banco.

Escalabilidade Horizontal

O PostgreSQL possui um excelente recurso de replicação que permite adicionar servidores réplicas que podem ser usados para consultas. Nas próximas sessões, esse mecanismo será explicado em detalhes.

Essas réplicas podem ser especialmente úteis para desafogar o servidor principal, redirecionando para elas consultas pesadas e relatórios. Também pode-se utilizar as réplicas para balancear a carga de leitura por 2, 3 ou quantas máquinas forem necessárias.

Balanceamento de carga

Para utilizar as réplicas, podemos adaptar a aplicação para apontar para as novas máquinas e direcionar as operações manualmente.

Outra abordagem é utilizar uma camada de software que se apresente para a aplicação como apenas um banco e faça o balanceamento automático da leitura entre as instâncias. Um software bastante usado para isso é o pgPool-II. Ele será estudado na sessão específica sobre Replicação.

Memória

Sistemas com pouca memória física podem prejudicar a performance do SGBD na medida em que poderão demandar muitas operações de SWAP e, consequentemente, aumentar significativamente as operações de I/O no sistema. Outro sintoma da falta de memória pode ser um baixo indice de acerto no page cache e shared buffer. Finalmente, devem ser consideradas situações especiais tais como "Cache frio" e "cache sujo".

Filesystem

Tuning e escolha de filesystem são tarefas complexas e trabalhosas, pois envolvem a análise de parâmetros que podem afetar de forma distinta questões relacionadas com o desempenho e com a segurança contra falhas, em ambos os casos exigindo testes exaustivos.

Nos Sistemas Operacionais de hoje, o sistema de arquivos mais usado, o EXT4, mostra-se bastante eficiente, bem mais do que o seu antecessor, o EXT3. Uma opção crescente é o XFS, que parece ter melhor desempenho. Podem ser utilizados filesystems diferentes para funções diferentes, por exemplo, privilegiando aquele com melhor desempenho de escrita para armazenar os WAL, escolhendo outro filesystem para os índices.

Um parâmetro relacionado ao filesystem que pode ser ajustado sem preocupação com efeitos colaterais, para bancos de dados, é o noatime. Definir esse parâmetro no arquivo /etc/fstab para determinada partição, indica ao kernel para não registrar a última hora em

O PostgreSQL não tem um modo “raw”, sem o uso de filesystem, onde o próprio banco gerencia as operações de I/O.

que cada arquivo foi acessado naquele filesystem. Esse é um overhead normalmente desnecessário para os arquivos relacionados ao Banco de Dados.

Armazenamento

Em Bancos de Dados convencionais, os discos e o acesso a eles são componentes sempre importantes, variando um pouco o peso de cada propriedade dependendo do seu uso: tamanho, confiabilidade e velocidade.

Atualmente as principais tecnologias de discos são SATA, que apresenta discos com maior capacidade (3TB), e SAS, que disponibiliza discos mais rápidos, porém menores. A tendência é que servidores utilizem discos SAS.

Existem também os discos de estado sólido, discos flash ou, simplesmente, SSD. Sem partes mecânicas, eles possuem desempenho muito superior aos discos tradicionais. Porém, trata-se de tecnologia relativamente nova, com debates sobre sua confiabilidade principalmente para operações de escrita. Os SSD são ainda muito caros e possuem pouca capacidade.

Se estiver considerando o uso de dessa tecnologia para Banco de Dados, não use marcas baratas e confirme com o fabricante o funcionamento do Write Cache, que é a bufferização de requisições de escrita até atingirem o tamanho mínimo de bloco para serem gravadas de fato. Se não houver um write cache com bateria, os dados ali armazenados podem ser corrompidos em caso de interrupção no fornecimento de energia (falta de luz).

Se estiver disponível, o uso de redes Storage Array Network (SAN) pode ser a melhor opção. Nesse cenário os discos estão em um equipamento externo, storage, e são acessados por rede Fiber Channel ou Ethernet. É verdade que um sistema SAN sobre ethernet de 1Gb pode não ser tão eficiente quanto discos locais ao servidor, mas redes fiber channel de 8Gb ou ethernet 10Gb resolvem esse ponto.

Normalmente esses storages externos possuem grande capacidade de cache, sendo 16GB uma capacidade comumente encontrada. Assim, a desvantagem da latência adicionada pelo uso da rede para acessar o disco é compensada por quase nunca ler ou escrever dos discos, já que os dados quase sempre estão no cache do storage. Esses caches são battery-backed cache, cache de memória com bateria. Assim o storage pode receber uma requisição de escrita de I/O, gravá-la apenas no cache e responder Ok, sem o risco de perda do dado e sem o tempo de espera da gravação no disco.

RAID

As opções de configuração de um RAID são:

- RAID 0: Stripping dos dados: desempenho sem segurança.
- RAID 1: Mirroring dos dados: segurança sem desempenho.
- RAID 1+0: Stripping e Mirroring: ideal para bancos.
- RAID 5: Stripping e Paridade: desempenho e segurança com custo.

É uma técnica para organização de um conjunto de discos, preferencialmente iguais, para fornecer melhor desempenho e confiabilidade do que obtidas com discos individuais. Isso é feito de forma transparente para o usuário, na medida em que os múltiplos discos são apresentados como um dispositivo único. Pode-se usar RAID por software ou hardware, sendo este último melhor em desempenho e confiabilidade.

RAID 0

Nessa organização dividem-se os dados para gravá-los em vários discos em paralelo. A leitura também é feita em paralelo a partir de todos os discos envolvidos. Isso fornece grande desempenho e nenhuma redundância. Se um único disco falhar, toda a informação é perdida. Essa quebra dos dados é chamada de striping.

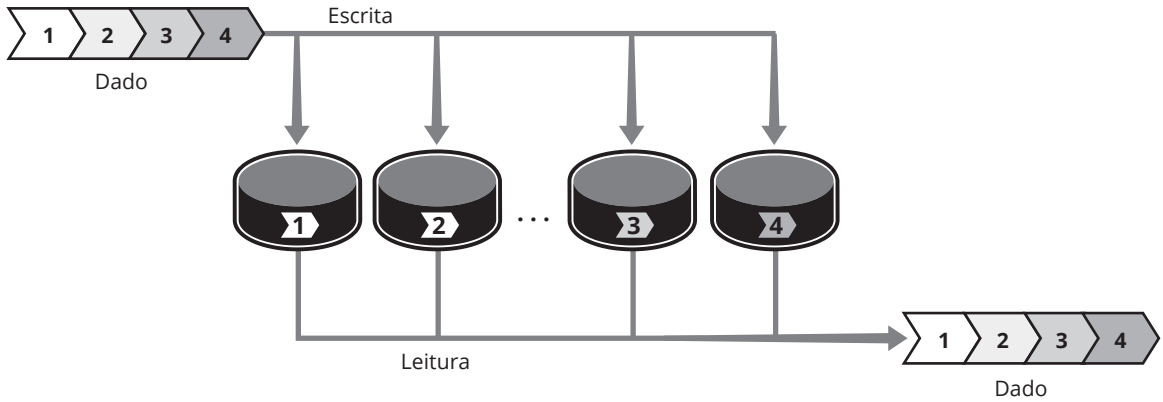


Figura 8.3
RAID 0 – o dado é quebrado em diversos discos: desempenho sem segurança.

RAID 1

No nível RAID 1, os dados são replicados em dois ou mais discos. Nesse caso o foco é redundância para tolerância a falhas, mas o desempenho de escrita é impactado pelas gravações adicionais. Esse recurso é chamado mirroring.

RAID 1+0

Também chamado RAID 10, é a junção dos níveis 0 e 1, fornecendo desempenho na escrita e na leitura com striping e segurança com o mirroring. É o ideal para Bancos de Dados, principalmente para logs de transação. A desvantagem fica por conta do grande consumo de espaço, necessário para realizar o espelhamento.

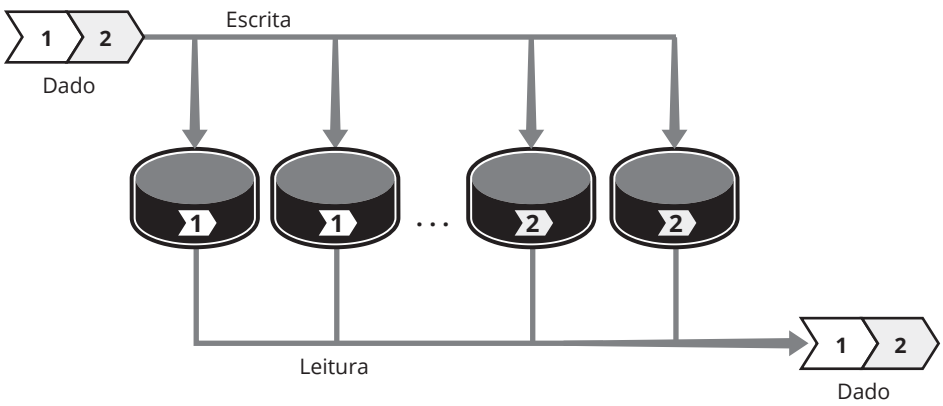


Figura 8.4
RAID 1+0: segurança com desempenho.

RAID 5

Esse layout fornece desempenho através de striping, e também segurança através de paridade. Esse é um recurso como um checksum, que permite calcular o dado original em caso de perda de algum dos discos onde os dados foram distribuídos. Em cada escrita é feito o cálculo de paridade e gravado em vários discos para permitir a reconstrução dos dados em caso de falhas. Fornece bom desempenho de leitura, mas um certo overhead para escrita, com a vantagem de utilizar menos espaço adicional do que o RAID1+0.

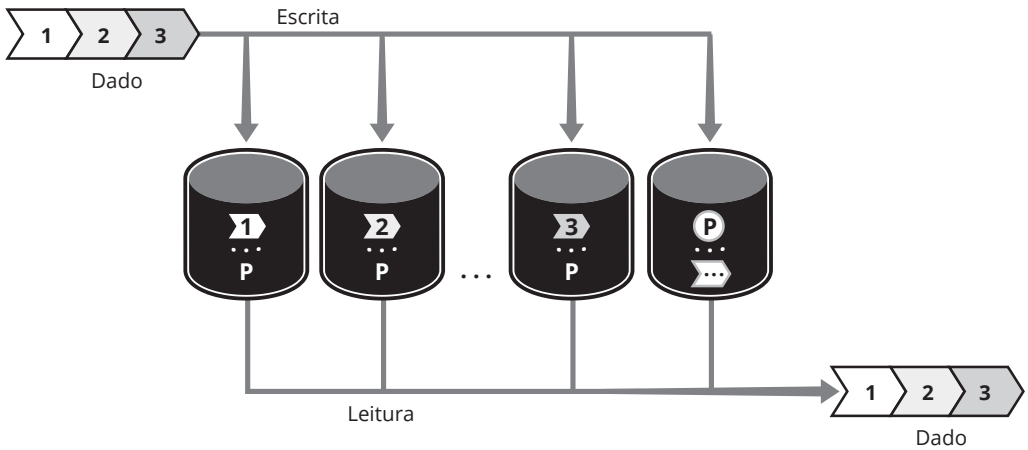


Figura 8.5 RAID 5: striping com cálculo de paridade para reconstrução dos dados.

Armazenamento: separação de funções

Durante as sessões anteriores, em diversos momentos foi comentada a possibilidade de separação dos dados do banco do log de transação, o WAL. Na verdade, em sistemas grandes com muita carga, pode-se estender essa política de separação em diversos níveis.

Colocar os arquivos de dados separados, seja em um disco, área de storage ou área RAID, pode não ser suficiente. Não é incomum ser necessário colocar bases de dados em discos separados, ou mesmo tabelas ou índices em discos separados. Até mesmo uma única tabela pode ter de ser particionada em diversos discos.

As possibilidades são diversas e devem ser unidos todos conhecimentos sobre tecnologia de disco, tecnologia de acesso aos discos, RAID e filesystem para chegar a melhor solução para a sua necessidade.

Uma excelente configuração, se os recursos estiverem disponíveis, é:

- Arquivos de dados em RAID 5 com EXT4 ou XFS.
- WAL em RAID 1+0 com EXT4.
- Índices em RAID 1+0 com XFS ou EXT4, possivelmente em SSD.
- Log de Atividades, se intensivamente usado, em algum disco separado.

Essa é apenas uma ideia geral, que pode ser desnecessária para um ambiente ou insuficiente para outra situação.

Virtualização

Não podemos deixar de tecer comentários sobre Bancos de Dados em máquinas virtuais. Esse é um tema muito controverso e não há resposta fácil. Para alguns administradores de Bancos de Dados, é uma verdadeira heresia sequer considerar bancos em ambientes virtualizados. O senso comum indica que um Banco de Dados instalado em uma máquina física executará melhor do que em um ambiente virtual. Até porque existe custo para adicionar uma camada adicional para acesso aos recursos do banco.

A questão que se coloca é se vale a pena enfrentar esse custo em função dos benefícios oferecidos. A virtualização pode garantir escalabilidade, vertical e horizontal, impensáveis para administradores de máquinas físicas. Em virtualizadores modernos pode-se clonar um máquina com um simples comando, enquanto criar uma nova máquina física poderia tomar uma semana de configuração, sem falar dos custos de aquisição.

Com respeito à escalabilidade vertical, bastam "dois cliques" para que um administrador de ambientes virtuais possa dobrar o número de núcleos de processadores e memória de uma máquina que esteja enfrentando sobrecarga. Seu banco pode estar trabalhando com 8 core e no instante seguinte passar a operar com 16 core, sem qualquer downtime. Essas facilidades não podem ser desprezadas.

Por outro lado, algumas situações ou cargas de sistemas não são tão facilmente virtualizadas.

A melhor estratégia é tentar tirar o melhor proveito dos dois mundos. No caso de discos para Bancos de Dados em ambientes virtuais, use sempre o modo raw, em que o disco apresentado para a máquina física é repassado diretamente para a máquina virtual, evitando (ou minimizando) a interferência do hipervisor nas operações de I/O.

Memória

Na sessão de monitoramento, foram vistas várias formas de monitorar os números da memória. Pode-se analisar a quantidade de memória ocupada e para cache, em conjunto com o tamanho do Shared Buffer, permitindo identificar se é necessário aumentar a memória física do servidor.

Um sinal de alerta é sempre a ocorrência de swap. Mas lembre-se de que o SO pode decidir fazer swap de determinados dados mesmo sem estar faltando memória. O Linux normalmente não deixa memória sobrando, alocando a maior parte da memória livre para cache e desalocando partes quando é necessário atender pedidos dos processos. Assim, é possível ocorrer swap com grandes quantidades de memória em cache.

O aumento de carga de I/O também pode ser evidência de falta de memória. Quando os dados não forem mais encontrados no shared buffer e no page cache, tornando mais frequente o acesso a disco, essa é uma situação que pode indicar a necessidade de mais memória, especialmente se o tuning na configuração do banco ou nas queries envolvidas já tiver sido realizado sem corrigir o problema.

É importante considerar que algumas situações específicas envolvendo a memória resultam de cache sujo ou vazio. Após reiniciar o PostgreSQL, o shared buffer estará vazio e todas as requisições serão solicitadas ao disco. Nesse caso ainda poderão ser encontradas no page cache, mas se a máquina também foi reiniciada, o cache do SO estará igualmente vazio. Nessa situação poderá ocorrer uma sobrecarga de I/O. Isso é frequentemente chamado de cold cache, ou cache frio. Deve-se esperar os dados serem recarregados para analisar a situação.

Já o cache sujo é quando alguma grande operação leu ou escreveu uma quantidade além do comum de dados que substituíram as informações que a aplicação estava processando. O impacto é o mesmo do cold cache.

Processadores

Não pretendemos fazer uma análise profunda sobre processadores, mas apenas sugerir a reflexão sobre a necessidade de ter maior quantidade de núcleos ou núcleos mais rápidos. A arquitetura de acesso à memória por chip, por zona ou simétrica, também merece alguma consideração.

Se a sua carga de sistema for online para muitos usuários, como a maioria dos sistemas web atuais, mais núcleos apresenta-se como a melhor opção. Caso seu ambiente demande quantidade menor de operações, mas essas operações sejam mais pesadas, como é o caso em soluções de BI, a melhor alternativa passa a ser uma quantidade menor de núcleos mais rápidos.

Rede e serviços

Em uma infraestrutura estável, é difícil que a rede se configure como um problema para o Banco de Dados. Algumas considerações talvez possam ser feitas quanto ao caminho entre servidores de aplicação e o servidor de Banco de Dados. Caso seja possível, respeitando as políticas de segurança da instituição, não ter um firewall entre o banco e aplicação pode melhorar bastante o desempenho e evitar gargalos.

Outro ponto a ser analisado é a dependência do desempenho do DNS na ligação entre servidores de aplicação e bancos de dados. Um erro numa configuração de uma entrada DNS ou DNS reverso pode causar uma lentidão na resolução de nomes que acabará refletindo no acesso ao banco.

Resumo

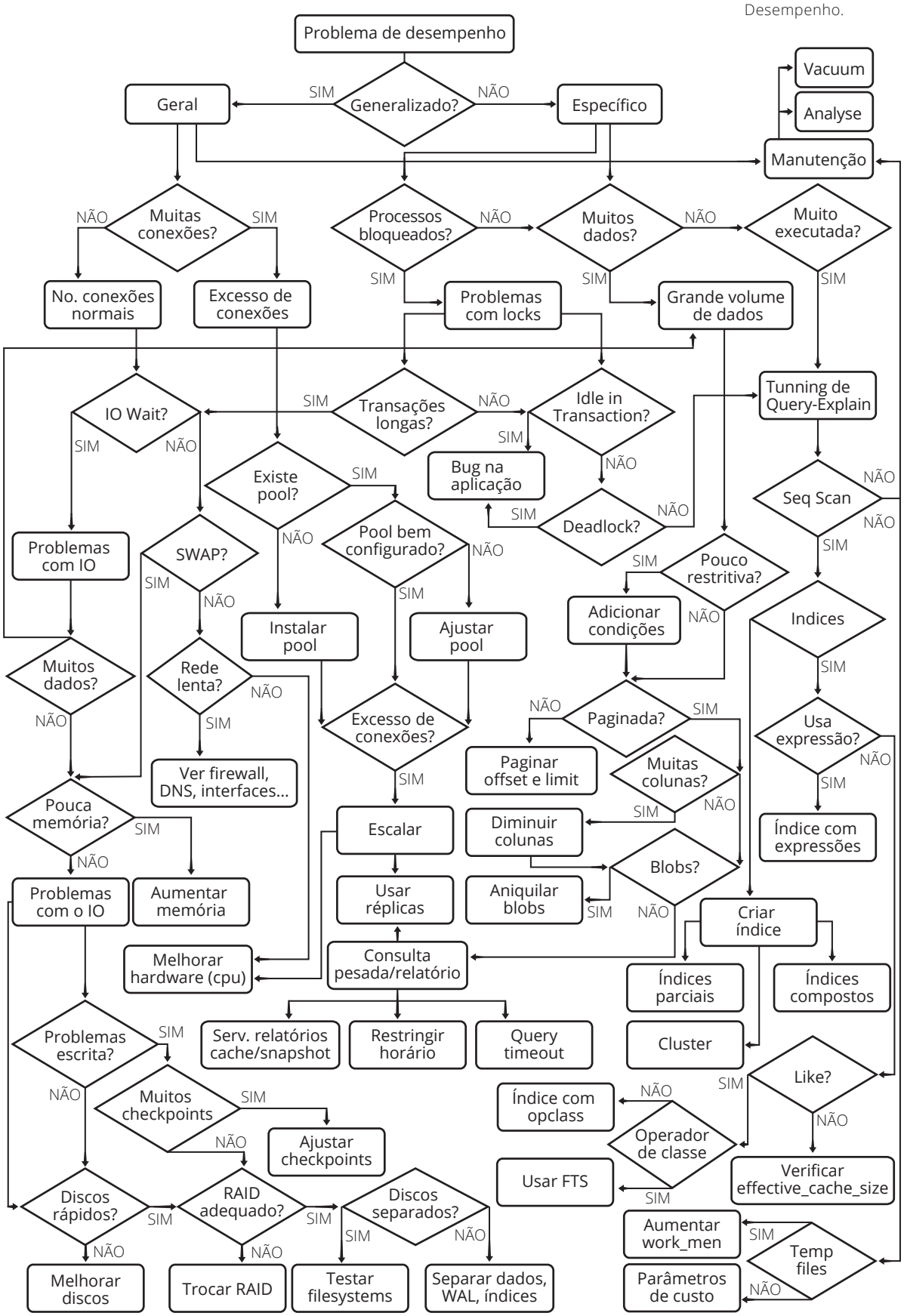


Figura 8.6
Problemas de Desempenho.

9

Backup e recuperação

objetivos

Conhecer as formas de segurança física de dados fornecidas pelo PostgreSQL, suas opções mais comuns e suas aplicações; Entender as variações do dump, ou backup lógico, e o mecanismo que viabiliza o Point-in-Time Recovery (PITR).

conceitos

Dump; Restore; Backup online; Backup Lógico e Físico; Dump Texto; Dump Binário; Point-in-Time Recovery e Log Shipping.

O PostgreSQL possui duas estratégias de backup: o dump de bases de dados individuais e o chamado backup físico e de WALs, para permitir o Point-in-Time Recovery.

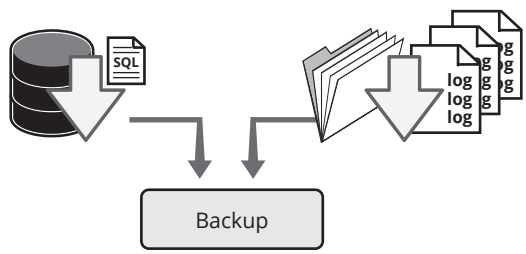


Figura 9.1 Estratégias de backup do PostgreSQL.

Backup lógico (dump)

O dump de uma base de dados, também chamado de backup lógico, é feito através do utilitário pg_dump. O dump é a geração de um arquivo com os comandos necessários para reconstruir a base no momento que o backup foi iniciado. Assim, no arquivo de dump não estarão os dados de um índice, mas sim o comando para reconstrução do índice.

O dump é dito consistente por tomar um snapshot do momento do início do backup e não considerar as alterações que venham a acontecer desse ponto em diante. O dump é também considerado um processo online, por não precisar parar o banco, remover conexões e nem mesmo bloquear as operações normais de leitura e escrita concorrentes. Os únicos comandos que não podem ser executados durante um dump são aqueles que precisam de lock exclusivo, como *ALTER TABLE* ou *CLUSTER*.

O arquivos de dump são altamente portáveis, podendo ser executados em versões diferentes do PostgreSQL ou de plataforma, como em um SO diferente ou em outra arquitetura de processadores (p.e., de 32 bits para 64 bits). Em função disso, o dump é muito usado no processo de atualização de versões do banco.

A Ferramenta pg_dump

O pg_dump pode ser usado remotamente. Com a maioria dos clientes do PostgreSQL, aplicam-se as opções de host (-h), porta (-p), usuário (-U) etc.

Por padrão, os backups são gerados como scripts SQL em formato texto. Pode-se também usar um formato de archive, chamado também binário, que permite restauração seletiva, por exemplo, de apenas um schema ou tabela.

Não existe um privilégio especial que permita fazer ou não backup, depende do acesso à tabela. Assim, é necessário ter permissão de SELECT em todas as tabelas da base para fazer um dump completo. Normalmente executa-se o dump com o superusuário postgres.

A sintaxe básica para executar um dump é:

```
$   pg_dump curso > /backup/curso.sql
```

ou

```
$   pg_dump -f /backup/curso.sql curso
```

Isso gerá um dump completo em formato texto da base. O pg_dump possui uma infinidade de opções e diversas combinações possíveis de parâmetros. As principais são apresentadas a seguir.

Formato

O parâmetro -F indica formato e pode ter os seguintes valores:

- c (custom) archive ou binário, é compactado e permite seleção no restore.
- t (tar) formato tar, limitado a 8GB.
- d (directory) objetos em estrutura de diretórios, compactados.
- p (plain-text) script SQL.

Para fazer um dump binário, usamos a opção -Fc.

- $ pg_dump **-Fc** -f /backup/curso.dump curso.

O dump nesse formato é compactado por padrão, e não humanamente legível.

Inclusão ou exclusão de Schemas

É possível especificar quais schemas farão parte do dump com o parâmetro -n. Ele incluirá apenas os schemas informados.

No exemplo a seguir, é feito um dump apenas do schema extra:

```
$   pg_dump -n extra -f /backup/extra.sql curso
```

Outro exemplo com os schemas extra e avaliacao, em formato binário, é apresentado:

```
$   pg_dump -Fc -n extra -n avaliacao -f /backup/extra_avaliacao.dump curso
```

Note que pode-se excluir um ou mais schemas selecionados através da opção -N. Isso fará o dump de todos os schemas da base, exceto os informados com -N. Um exemplo de dump de toda a base curso, exceto do schema extra:

```
$   pg_dump -N extra -f /backup/curso_sem_extra.sql curso
```

Inclusão ou exclusão de tabelas

Equivalente às opções para schema, existem os parâmetros -t e -T para tabelas. O primeiro fará o dump somente da tabela informada, e o segundo fará o dump de todas as tabelas, exceto daquelas informadas com -T.

Vejamos o exemplo para fazer um dump apenas da tabela times:

```
$   pg_dump -t public.times -f /backup/times.sql curso
```

Também pode-se informar -t múltiplas vezes para selecionar mais de uma tabela:

```
$   pg_dump -t public.times -t public.grupos -f /backup/times_grupos.sql curso
```

A exclusão de uma determinada tabela do dump com -T:

```
$   pg_dump -T public.cidades -f /backup/curso_sem_cidades.sql curso
```

Tanto para schemas quanto para tabelas, é possível utilizar padrões de string na passagem de parâmetros, conforme demonstrado a seguir:

```
$   pg_dump -t 'extra.log*' -f /backup/extra_log.sql curso
```

Somente dados

Para fazer dump somente dos dados, sem os comandos de criação da estrutura, ou seja, sem os CREATE SCHEMA, CREATE TABLE e assemelhados, utiliza-se o parâmetro -a.

```
$   pg_dump -a -f /backup/curso_somente_dados.sql curso
```

Note que para restaurar um dump destes é necessário garantir que os respectivos objetos já existam ou sejam previamente criados.

O dump da estrutura pode ser feito com a opção -s ou --schema-only. Por causa desse nome, às vezes é confundido com o dump de um schema apenas, que na verdade é feito com -n.

Somente estrutura

É possível fazer dump apenas da estrutura do banco (definição da base de dados e seus schemas e objetos).

Para isso, usa-se a opção -s:

```
$   pg_dump -s -f /backup/curso_somente_estrutura.sql curso
```

Dependências

Quando utilizado -n para escolher um schema ou -t para uma tabela, podem existir dependências de objetos em outros schemas. Por exemplo, pode existir uma foreign key ou uma visão apontando para uma tabela de outro schema. O pg_dump não fará dump desses objetos, levando a ocorrer erros no processo de restauração. Cabe ao administrador cuidar para que sejam previamente criados os objetos dessas relações de dependência.

Large objects

Os large objects são incluídos por padrão nos dumps completos, porém, em backups seletivos com -t, -n ou –s, eles não serão incluídos.

Nesses casos, para incluí-los, é necessário usar a opção -b:

```
$   pg_dump -b -n extra -f /backup/extra_com_blobs.sql curso
```

Excluir objetos existentes

No pg_dump é possível informar a opção -c para gerar os comandos de exclusão dos objetos antes de criá-los e populá-los. É útil para dump texto, pois com dumps binários é possível informar essa opção durante a restauração.

```
$   pg_dump -c -f /backup/curso.sql curso
```

Essa opção tem como desvantagem o fato de que os comandos são sempre executados durante a restauração, mesmo que os objetos não existam. Nessa situação, o comando de *drop* para um objeto que não existe exibirá uma mensagem de erro. Essas mensagens não impedem o sucesso da restauração, mas poluem a saída e complicam a localização de algum erro mais grave.

Criar a base de dados

É comum criar uma base de dados vazia antes de restaurar um dump, indicando-a como destino dos dados durante a restauração. Uma alternativa é adicionar uma instrução para criação da base dentro do próprio arquivo de dump. Isso se aplica para scripts – dump texto, onde é gerado o comando para criação da base seguido da conexão com esta, independentemente da base original.

Para dumps binários, esse comportamento pode ser escolhido pelo utilitário de restauração.

```
$   pg_dump -C -f /backup/curso.sql curso
```

Combinado com o parâmetro -c, emitir um comando de *drop* da base de dados antes de criá-la:

```
$   pg_dump -C -c -f /backup/curso.sql curso
```

Se a base indicada já existir, esta será excluída, desde que não existam conexões. Se houver conexões, será gerado um erro no DROP DATABASE. Como o comando *CREATE DATABASE* é executado em sequência, um novo erro será gerado, já que a base já existe (pois não foi possível excluí-la). Ao final do processo, contudo, será feito o \connect na base, que não foi nem excluída e nem recriada, e tudo parecerá funcionar normalmente.

Como o dump com -C tem a função de criar uma base nova, não são emitidos comandos para apagar objetos (somente a base é apagada). O resultado é que os comandos *CREATE* dos objetos na base também vão gerar mensagens de erro, uma vez que eles já existem.

Essas questões em torno da exclusão ou criação de objetos tornam o uso das respectivas opções no pg_dump um pouco confuso.

> A prática comum é o administrador manualmente apagar uma base pré-existente, criá-la vazia e restaurar os dumps sem utilizar parâmetros de exclusão ou criação, conseguindo assim uma restauração mais clara..

Permissões

Para gerar um dump sem privilégios, usa-se a opção -x:

```
$   pg_dump -x -f /backup/curso_sem_acl.sql curso
```

Não serão gerados os GRANTs, mas continuam os proprietários dos objetos com os atributos OWNER. Para remover também o owner, usa-se a opção -O:

```
$   pg_dump -O -x -f /backup/curso_sem_permissoes.sql curso
```

Compressão

É possível definir o nível de compactação do dump com o parâmetro -Z. É possível informar um valor de 0 a 9, onde 0 indica sem compressão e 9 o nível máximo. Por padrão, o dump binário (custom) é compactado com nível 6. Um dump do tipo texto não pode ser compactado.

A seguir apresentamos, respectivamente, exemplos de dump binário com compactação máxima e sem nenhuma compactação:

```
$   pg_dump -Fc -Z9 -f /backup/curso_super_compacatado.dump curso
$   pg_dump -Fc -Z0 -f /backup/curso_sem_compactacao.dump curso
```

Alterando Charset Encoding

O dump é sempre gerado com o encoding, codificação do conjunto de caracteres, da base de dados sobre a qual é aplicado.

Entretanto, é possível alterar esse formato com a opção -E:

```
$   pg_dump -E UTF8 -f /backup/curso_utf8.sql curso
```

Ignorando tablespaces originais

Quando o arquivo de dump é gerado, os comandos de criação de objetos são emitidos com seus tablespaces originais, que deverão existir no destino. É possível ignorar esses tablespaces com o argumento --no-tablespaces.

Nesse caso será usado o tablespace default no momento da restauração.

```
$   pg_dump --no-tablespaces -f /backup/curso_sem_tablespaces.sql curso
```

Desabilitar triggers

Para dumps somente de dados, pode ser útil emitir comandos para desabilitar triggers e validação de foreing keys. Para tanto, é necessário informar o parâmetro –disable-triggers.

Essa opção emitirá comandos para desabilitar e habilitar as triggers antes e depois da carga das tabelas.

```
$   pg_dump --disable-triggers -f /backup/curso_sem_triggers.sql curso
```

pg_dumpall

É um utilitário que faz o dump do servidor inteiro. Ele de fato executa o pg_dump internamente para cada base da instância, incluindo ainda os objetos globais – roles, tablespaces e privilégios– que nunca são gerados pelo pg_dump. Este dump é gerado somente no formato texto.

O pg_dumpall possui diversos parâmetros coincidentes com o pg_dump. A seguir destacamos aqueles que são diferentes. Sua sintaxe geral é:

```
$   pg_dumpall > /backup/servidor.sql
```

ou

```
$   pg_dumpall -f /backup/servidor.sql
```

Objetos globais

Para gerar um dump somente das informações globais – usuários, grupos e tablespaces – é possível informar a opção -g:

```
$   pg_dumpall -g -f /backup/servidor_soment_globais.sql
```

Roles

Para gerar o dump apenas das roles – usuários e grupos – use a opção -r:

```
$   pg_dumpall -r -f /backup/roles.sql
```

Tablespaces

Para gerar o dump apenas dos tablespaces, use a opção -t:

```
$   pg_dumpall -t -f /backup/tablespaces.sql
```

Dump com blobs

O dump de bases de dados com um grande volume de dados em large objects ou em campos do tipo bytea pode ser um sério problema para o Administrador. Tipicamente, esses campos são usados para armazenar conteúdo de arquivos, como imagens e arquivos no formato pdf, ou mesmo outros conteúdos multimídia.

Large objects são manipulados por um conjunto de funções específicas e são armazenados em estruturas paralelas ao schema e à tabela na qual são definidos. Ao fazer um dump de um schema ou tabela específicos, os blobs não são incluídos por padrão. Utilizar o argumento –b para os incluir fará com que todos os large objects da base sejam incluídos no dump, e não somente os relacionados ao schema ou tabela em questão.

Já os campos bytea não possuem essas inconsistências no dump. Porém, como os large objects, tem um péssimo desempenho e consomem muito tempo e espaço. É difícil aceitar que em certas situações o dump pode ficar maior do que a base de dados original (mesmo usando compactação). Lamentavelmente isso acontece frequentemente com bases com grande quantidade de blobs ou campos bytea.

Parte da explicação para isso está no mecanismo de TOAST, que já faz a compressão desses dados na base. Toda operação de leitura de um dado armazenado como TOAST exige que estes sejam previamente descomprimidos. Com um dump envolvendo dados TOAST não

é diferente. O pg_dump lê os dados da base e grava um arquivo. Se a base está cheia de blobs, serão realizadas grande quantidade de descompressões. Em seguida, tudo terá de ser novamente compactado (já que a compressão dos dados é o comportamento padrão do pg_dump). Esse duplo processamento de descompressão e compressão acaba consumindo muito tempo. Além disso, se o nível de compressão dos dados TOAST for maior que o padrão do pg_dump, o tamanho do backup poderá ser maior que o da base original.

O fato é que o dump dessas bases pode ser um pesadelo, alocando a janela de backup por várias horas e ao mesmo tempo consumindo grande quantidade de espaço em disco. Uma forma de amenizar esse comportamento é ajustar o nível de compressão.

Uma base com algumas centenas de GB de blobs ou bytea pode facilmente atingir 10 ou mais horas de tempo de backup, trabalhando com a compressão padrão (Z6), e ainda assim ficar com mais ou menos o mesmo tamanho da base original.

Baixar o nível de compressão, digamos para Z3, aumentará o espaço consumido em disco, mas consumirá menos tempo. Já a opção Z1 fornece uma opção boa de tempo com um nível razoável de compressão.

Se o dump tornar-se inviável para uma base nessas condições, pode-se adotar apenas o backup de filesystem, ou backup físico, como veremos adiante.

Restaurando Dump Texto – psql

Se o dump foi gerado em formato texto, ou seja, criando um script sql, ele é restaurado como qualquer outro script é executado: através do psql.

A forma geral é:

```
$   psql -d curso < /backup/curso.sql
```

O comando acima não criará a base curso (ela deve existir previamente). Outra opção é gerar o dump com a opção –C, conforme foi visto em "Criar a base de dados".

Pipe
Ou pipeline. É uma forma de comunicação entre processos largamente utilizada em sistemas baseados em Unix/Linux, através do operador "|", em que a saída de um programa é passada diretamente como entrada para outro. A vantagem de utilizá-lo em bancos de dados é que ele é uma estrutura apenas em memória, logo não é utilizado espaço em disco em sua execução.

Assim, pode-se conectar em qualquer base antes de executar o script, conforme o seguinte exemplo:

```
$   psql < /backup/curso_com_create.sql
```

Os usuários que receberão grants ou que serão donos de objetos também devem ser criados previamente. Se isso não for feito, teremos grande quantidade de erros, que não causarão a interrupção da restauração. Porém, ao final do processo, os privilégios previamente existentes não terão sido aplicados, e o owner dos objetos será o usuário executando a ação.

Para mudar esse comportamento, interrompendo a execução em caso de erro, pode-se informar o parâmetro ON_ERROR_STOP = on. É também possível executar a restauração em uma transação, desde que se informe o parâmetro –single-transaction.

Uma possibilidade interessante para o uso de dumps texto com o psql é numa transferência direta entre máquinas fazendo uso do **pipe**:

```
$   pg_dump -h servidor1 curso | psql -h servidor2 curso
```

Esse comando pode ser executado em uma terceira máquina, como uma estação de trabalho, que tenha acesso aos dois servidores. Reforçamos que essa operação não usará espaço em disco para a transferência devido ao uso do pipe.

Depois de toda restauração, é altamente recomendado atualizar as estatísticas da base, já que estas estarão inicialmente zeradas.

O dump em formato texto tem uma restauração simples e possui poucas opções. Para mais flexibilidade, deve-se usar o dump binário.

A Ferramenta pg_restore

Uma das principais vantagens do dump binário, além da compressão, é a possibilidade de restauração seletiva, seja de schemas, tabelas, dados ou somente da estrutura da base.

Para restaurar um dump em formato binário, usa-se o utilitário pg_restore, que possui algumas opções de seleção semelhantes ao pg_dump, com a vantagem de poder filtrar objetos de um arquivo de dump completo (enquanto o pg_dump irá gerar apenas a informação selecionada).

As seguintes opções tem significado análogo as do pg_dump:

```
-n                  restaurar apenas o schema
-t                  restaurar apenas a tabela
-a                  restaurar apenas os dados
-s                  restaurar apenas a estrutura
-c                  executar DROP dos objetos antes de criá-los
-C                  executar o CREATE da base de dados
-x                  não executar os GRANTs
-O                  não atribuir os donos de objetos
--no-tablespaces    não aplicar as alterações de tablespace
--disable-triggers  desabilitar triggers
```

A forma geral do pg_restore é:

```
$   pg_restore -d curso /backup/curso.dump
```

Como na restauração com o psql, a base curso deve existir previamente ou a opção -C deve ser usada.

Apresentamos agora algumas opções específicas do pg_restore.

Seleção de objetos

É possível especificar apenas um índice, trigger ou função para ser restaurado com as opções -I, -T e –P, respectivamente. Porém, esses objetos possuem fortes dependências, exigindo muito cuidado na hora da restauração. Para se restaurar os índices, é necessário que as tabelas existam. Para restaurar uma trigger, é necessário que a tabela e a trigger function existam.

Vejamos um exemplo restaurando a tabela e o índice:

```
pg_restore -t item_financeiro -I idx_data -d curso curso.dump
```

Para restaurar uma trigger e sua função:

```
$   pg_restore -T t_grupos_update -P “t_grupos()” -d curso curso.dump
```

Processamento paralelo

Por padrão, a restauração é executada sequencialmente, usando um processo. É possível informar com o parâmetro -j um número de processos paralelos para restaurar os dados do dump. Definir o valor desse parâmetro depende do número de processadores e velocidade de disco do ambiente em que o restore será realizado.

Por exemplo, para restaurar um dump com quatro processos paralelos:

```
$   pg_restore -j 4 -d bench bench.dump
```

Controle de erros

O pg_restore não interrompe uma restauração por conta de eventuais erros. As mensagens são emitidas e ao final é exibido um totalizador com os erros ocorridos.

Mas é possível mudar esse comportamento através da opção –e, causando uma interrupção do restore caso qualquer erro aconteça.

```
$   pg_restore -e -d bench bench.dump
```

É possível também executar a restauração em uma única transação, exatamente como no psql, informando o parâmetro –single-transaction.

```
$   pg_restore --single-transaction -d bench bench.dump
```

Gerar lista de conteúdo

A opção -l gera uma lista com o conteúdo do arquivo de dump. Esse argumento causa apenas a geração do arquivo, não ocorrendo um restauração de fato.

```
$   pg_restore -l -d bench bench.dump > lista.txt
```

```
; Selected TOC Entries:
;
1980; 1262 58970 DATABASE - bench postgres
5; 2615 2200 SCHEMA - public postgres
1981; 0 0 COMMENT - SCHEMA public postgres
1982; 0 0 ACL - public postgres
173; 3079 11765 EXTENSION - plpgsql
1983; 0 0 COMMENT - EXTENSION plpgsql
170; 1259 58977 TABLE public pgbench_accounts postgres
168; 1259 58971 TABLE public pgbench_branches postgres
171; 1259 58980 TABLE public pgbench_history postgres
169; 1259 58974 TABLE public pgbench_tellers postgres
1974; 0 58977 TABLE DATA public pgbench_accounts postgres
1972; 0 58971 TABLE DATA public pgbench_branches postgres
1975; 0 58980 TABLE DATA public pgbench_history postgres
1973; 0 58974 TABLE DATA public pgbench_tellers postgres
1865; 2606 58989 CONSTRAINT public pgbench_accounts_pkey postgres
1860; 2606 58985 CONSTRAINT public pgbench_branches_pkey postgres
1862; 2606 58987 CONSTRAINT public pgbench_tellers_pkey postgres
1863; 1259 58990 INDEX public idx_accounts_filler postgres
```

Figura 9.2
Lista de conteúdo de arquivo de dump gerada com a opção –l.

Informar lista de restauração

É possível informar uma lista de objetos a serem restaurados, inclusive em uma ordem específica, através do parâmetro -L. O mais comum é primeiro gerar uma lista completa, conforme visto em "Gerar lista de conteúdo". Esse arquivo é então editado para refletir os filtros e ordenação desejados.

A lista é então aplicada conforme o exemplo:

```
$   pg_restore -L lista.txt -d curso curso.dump
```

Backup Contínuo: Backup Físico e WALs

Uma outra técnica de backup do PostgreSQL é a que permite o Point-in-Time Recovery (PITR), ou seja, restaurar o banco de acordo com um ponto determinado no tempo.
Essa técnica é chamada de Backup Contínuo ou Backup Físico e Arquivamento de Wals.
Basicamente consiste em:

- Habilitar o backup de todos os logs de transação ou arquivamento de WAL;
- Fazer um backup físico, ou backup base, através de uma cópia dos dados diretamente do filesystem, cujos logs podem ser aplicados após uma restauração.

Além da possibilidade da restauração PITR, essa técnica de backup possibilita montar um servidor standby usando o backup base e continuamente aplicando os segmentos de WALs arquivados. Isso é chamado de replicação por Log Shipping.

Uma característica importante que deve ser salientada é que só é possível fazer backup da instância inteira, não é possível fazer um backup físico de uma base e aplicar logs de transação apenas dessa base. Os segmentos de WAL são globais da instância.

Os backups de dump feitos com pg_dump ou pg_dumpall não podem ser usados com esse procedimento.

Habilitando o Arquivamento de WALs

Como já foi explicado, o Write Ahead Log (WAL), é o log de transações do PostgreSQL, normalmente dividido em arquivos de 16 MB, chamados segmentos de log. Para ser possível fazer uma restauração em um determinado momento, é necessário fazer o backup de todos os segmentos de log.

Para tanto, é necessário configurar três opções do *postgresql.conf*:

- wal_level = archiv
  Deve ser definido como archive ou hot_standby. As duas opções permitem backup de wals;
- archive_mode = on
  Deve estar ligado para ser possível executar o comando de arquivamento;
- archive_command = "cp %p /backup/pitr/atual/wals/%f"
  O comando para fazer o backup dos segmentos de log. Pode ser um simples cp para um diretório local, um scp para outra máquina, um script personalizado, um rsync ou qualquer comando que possa arquivar o segmento de log.

Para que essa configuração tenha efeito, será necessário reiniciar o PostgreSQL. Após o banco estar no ar, acompanhe se o backup de logs está ocorrendo. No nosso exemplo isso ocorre no diretório "/backup/pitr/atual/wals".

No próximo passo, veremos detalhes sobre a criação desses diretórios dinamicamente, a cada vez que se executa um backup base. Atente para o fato de que poderão ocorrer erros se no momento do arquivamento o diretório não existir. Um exemplo de erro desse tipo, registrado no log:

```
WARNING:  transaction log file "000000010000000200000074" could not be archived: t
oo many failures
cp: cannot create regular file
`/backup/pitr/atual/wals/000000010000000200000074': No such file or directory
LOG:  archive command failed with exit code 1
DETAIL:  The failed archive command was: cp
pg_xlog/000000010000000200000074
/backup/pitr/atual/wals/000000010000000200000074
```

Fazendo um backup base

Há duas formas gerais de fazer esse backup inicial: manualmente, caso precise-se de mais flexibilidade, ou através do utilitário pg_basebackup.

Backup base manual

O processo de criação do backup base se resume em:

a. Colocar o banco em modo de backup;
b. Fazer a cópia da área de dados e de qualquer outra área de tablespaces que exista, utilizando como ferramenta: cp, scp, rsync, tar, algum software de backup etc.;
c. Retirar o banco do modo de backup.

Não precisam ser feitos backups do diretório pg_xlog e dos arquivos postmaster.pid e postmaster.opts.

Exemplo:

```
$   psql -c "select pg_start_backup('backup_label');"

$   rsync -av --exclude=pg_xlog/*
                --exclude=postmaster.pid
                --exclude=postmaster.opts
                /db/data/
                /backup/pitr/atual/

$   psql -c "select pg_stop_backup();"
```

A função pg_start_backup recebe uma string como parâmetro, que deve ser o nome de um arquivo temporário criado dentro do pgdata, com informações sobre o backup. Pode ter um rótulo qualquer, por exemplo, "backup_base_20140131".

Para perfeito funcionamento e organização, antes desses passos do backup, pode-se adicionar a criação de um diretório para o backup com a data corrente e também de um link simbólico chamado "atual". Assim, tanto o processo do backup principal quanto o arquivamento de wals sempre gravarão no caminho "/backup/pitr/atual".

```
$   mkdir /backup/pitr/`date +%Y-%m-%d`
$   ln -s /backup/pitr/`date +%Y-%m-%d` /backup/pitr/atual
$   mkdir /backup/pitr/atual/wals
```

```
postgres@pg01:~$ ls -l /backup/pitr/
total 8
drwxrwxr-x 2 postgres 4096 Apr  3 22:07 2014-02-22/
drwxrwxr-x 3 postgres 4096 Feb 23 21:09 2014-02-23/
lrwxrwxrwx 1 postgres 23 Feb 23 13:34 atual -> /backup/pitr/2014-02-23/
postgres@pg01:~$
```

Figura 9.3
Definição do diretório de backup com link simbólico.

Nesse momento você possuirá um backup base e os arquivos de wal gerados sob o mesmo diretório "atual". Pode-se copiá-lo para fitas, outros servidores da rede etc.

A geração do backup base pode ser agendada uma vez por dia ou por semana, dependendo da carga e de quanto tempo para o processo de restauração é tolerável no seu ambiente.

Quanto maior o intervalo de tempo entre os backups base, maior a quantidade de log de transações produzido. Consequentemente, maior será o tempo de restauração. Aumenta também a quantidade de arquivos e a probabilidade de um deles estar corrompido.

A Ferramenta pg_basebackup

Esse utilitário pode ser usado para fazer o backup base de um backup contínuo, bem como para construir servidor standby. Ele coloca automaticamente o banco em modo de backup e faz a cópia dos dados da instância para o diretório informado.

Antes de usar o pg_basebackup, entretanto, é necessário providenciar as seguintes configurações:

- O acesso do tipo replication, a partir da própria máquina, precisa estar liberado no *pg_hba.conf*. A linha a seguir mostra como deve ser essa entrada:

```
local     eplication     postgres     trust
```

- É preciso ter pelo menos um slot disponível para replicação. A quantidade é definida pelo parâmetro max_wal_sender no *postgresql.conf*. O valor deve ser pelo menos um, podendo ser um número maior de acordo com a quantidade de réplicas que se possua:

```
max_wal_senders = 1
```

- Finalmente, o parâmetro wal_level deve estar configurado para archive (configuração que já deverá ter sido feita para permitir o backup dos wals).

Tendo confirmado que essas configurações estão em vigor, a seguinte linha de comando executará o backup físico do servidor local:

```
$ pg_basebackup -P -D /backup/pitr/atual
```

A partir desse ponto, o backup está concluído se o arquivamento de WALs já estiver funcionando. Lembre-se de que mesmo usando pg_basebackup é necessário criar a estrutura de diretórios para o backup base e WALs.

Cuidados adicionais devem ser observados caso sejam usados tablespaces além do padrão pg_default (pgdata/base). Isso porque o modo padrão de funcionamento do pg_basebackup supõe que será feita uma cópia do backup base entre máquinas diferentes, para uso com replicação. Nessa situação, os tablespaces existentes na origem serão movimentados exatamente para o mesmo caminho no destino.

Mas se o backup é feito na mesma máquina e houver tablespaces além do padrão, o seguinte erro é gerado:

```
$   pg_basebackup -P -D /backup/pitr/atual

pg_basebackup: directory "/db2/tbs_indices" exists but is not empty
```

A mensagem é gerada quando o utilitário tenta criar o tablespace no mesmo lugar do original, uma vez que um tablespace nunca é criado em um diretório com dados.

Para evitar esse problema, usa-se a opção -Ft, que empacotará o conteúdo dos tablespaces com tar e colocá-los no mesmo diretório do backup principal. A opção -z também compactará o backup.

```
$   pg_basebackup -Ft -z -P -D /backup/pitr/atual
1094167/1094167 kB (100%), 3/3 tablespaces
```

```
postgres@pg01:~$ ls -l /backup/pitr/atual/
total 8
drwxrwxr-x 2 postgres    88196 Feb 20 08:07 50766.tar.gz
drwxrwxr-x 3 postgres    88190 Feb 23 21:09 50767.tar.gz
lrwxrwxrwx 1 postgres 72957512 Feb 23 13:34 base.tar.gz
postgres@pg01:~$
```

Figura 9.4
Área de dados padrão e tablespaces compactados com pg_basebackup.

Point-in-Time Recovery – PITR

Como já foi comentado, o backup físico junto com o backup dos segmentos de log permite a recuperação do banco para um estado mais atualizado possível em caso de um crash ou de algum conjunto de dados cruciais ter sido removido.

A recuperação do backup em uma situação como essas acontece da seguinte forma:

- Restaura-se o backup base para um diretório;
- Disponibiliza-se os segmentos de wal;
- Executa-se o PostgreSQL no diretório restaurado com o arquivo *recovery.conf* apontando para o diretório onde estão os WALs.

Depois de obter os arquivos de backup, seja via fita, via outro servidor da rede ou mesmo via algum disco local, deve-se decidir onde estes serão restaurados.

No caso de um crash no servidor de produção, o desafio é tentar colocá-lo no ar o mais rapidamente possível. Para isso, os dados do servidor atual deverão ser sobrescritos com os dados vindos do backup. Já em uma situação onde é preciso obter uma informação que foi removida, mas não se deseja fazer a recuperação sobre o servidor de produção, o mais aconselhável é usar outro servidor qualquer, ou ainda, usar um outro diretório na mesma máquina.

Os passos necessários para a restauração são:

1. Parar o PostgreSQL

```
$   pg_ctl stop
```

Se for o caso de sobrescrever os dados no servidor, os seguintes passos adicionais são recomendados:

1.1. Fazer uma cópia de todo pgdata ou pelo menos do diretório pg_xlog.

1.2. Apagar tudo abaixo do pgdata:

```
$  rm -Rf /db/data/*
```

2. Copiar o backup base, descompactando se necessário, para o pgdata. Se existirem tablespaces estes devem ser copiados para suas áreas.

```
$   rsync -av --exclude=wals /backup/pitr/2014-01-15/  /db/data/
```

3. Se existir o diretório pg_xlog vindo do backup, esvaziar seu conteúdo.

Se não existir, providenciar sua criação, juntamente com o subdiretório "pg_xlog/archive_status".

```
$   mkdir /db/data/pg_xlog
$   mkdir /db/data/pg_xlog/archive_status
```

4. Criar o arquivo *recovery.conf*.

Informar o diretório onde estão localizados os WALs em restore_command.

Pode-se informar o momento no tempo a ser usado como referência através do parâmetro recovery_targe_time. Se este não for informado, serão aplicados todos os logs disponíveis.

```
$   vi /db/data/recovery.conf
    restore_command  = 'cp /backup/pitr/2014-01-15/wals/%f %p'
    recovery_target_time      =  '2014-01-15 16:20:00 BRT'
```

5. O passo final é iniciar o PostgreSQL.

O banco iniciará o processo de recovery, lendo e aplicando os WALs. Quando estiver terminado, o arquivo *recovery.conf* será renomeado para *recovery.done*.

```
user=,db= LOG:  restored log file "0000000100000000200000007D" from archive
cp: cannot stat `/backup/pitr/2014-02-20/wals/0000000100000000200000007E': No such f
ile or directory
user=,db= LOG:  could not open file "pg_xlog/0000000100000000200000007E" (log file 2
, segment 126): No such file or directory
user=,db= LOG:  redo done at 2/7D000074
user=,db= LOG:  restored log file "0000000100000000200000007D" from archive
cp: cannot stat `/backup/pitr/2014-02-20/wals/00000002.history': No such file or d
irectory
user=,db= LOG:  selected new timeline ID: 2
cp: cannot stat `/backup/pitr/2014-02-20/wals/00000001.history': No such file or d
irectory
user=,db= LOG:  archive recovery complete
user=,db= LOG:  database system is ready to accept connections
```

Figura 9.5
Log do PostgreSQL exibindo mensagem de recuperação concluída.

Durante o processo de restauração, pode-se impedir a conexão de usuários através do arquivo *pg_hba.conf*.

Se o backup for recuperado de uma área temporária, sem apagar o pgdata principal, pode-se executar o banco em outro diretório. Por exemplo:

```
$   pg_clt start -D /backup/restaurado
```

Importante entender que o conteúdo de "/backup/restaurado" será alterado com a aplicação dos logs de transação. Assim, não se deve levantar o banco sobre a área de backup original.

Essa opção também não funcionará se estão sendo usados tablespaces separados. Nesse caso, será necessário sobrescrever todas as áreas de dados ou usar outra máquina.

Outra informação importante é que a versão do PostgreSQL onde a restauração será feita deve ser a mesma onde o backup foi executado, por tratar-se de um backup físico.

Resumo

Dump

Dump em formato texto:

```
pg_dump -f /backup/curso.sql curso
```

Dump em formato binário:

```
pg_dump -Fc /backup/curso.dump curso
```

Dump seletivo:

```
-n/-N       Apenas o schema/Não incluir schema
-t/-T       Apenas a tabela/Não incluir tabela
-a/-s       Apenas dados / Apenas estrutura
-c/-C       Excluir objetos antes de criá-los / Criar base de dados
```

Dump de todas as bases e objetos globais

```
pg_dumpall -f /backup/servidor.sql
```

Não armazene arquivos no banco.

Restauração de Dump

Restaurar dump texto

```
psql -d curso < /backup/curso.sql
```

Restaurar dump binário

```
pg_restore -d curso /backup/curso.dump
```

Backup Contínuo

Habilitar Arquivamento de WALs :

```
Definir wal_level    = archive
Definir archive_mode = on
Definir archive_command       = cp …
```

Fazer Backup Base.

Manual.

Criar diretórios, start backup, copiar, stop backup:

```
pg_basebackup
    Definir max_wal_sender > 0
    Permitir “replication” no pg_hba.conf
    Criar diretórios, pg_base_backup
```

Recuperação: Point-in-Time Recovery – PITR

- Parar banco;
- Apagar tudo no pgdata;
- Copiar arquivos do backup para o pgdata;
- Criar *recovery.conf*;
- Localização dos backups de WALs;
- Subir o banco para iniciar a recuperação.

10

Replicação

objetivos

Saber o que é replicação, as possibilidades para seu uso e os modos suportados pelo PostgreSQL para esse recurso; Conhecer o funcionamento das formas de replicações nativas, Log Shipping e Streaming Replication, assim como a configuração e operação básica de cada uma delas.

conceitos

Alta Disponibilidade; Warm Standby; Hot Standby; Replicação por Log Shipping; Stream Replication; Replicação Síncrona; Replicação Assíncrona e Balanceamento de Carga.

Visão geral

Replicação é a possibilidade de se ter uma ou mais cópias de um banco de dados repetidas em outros ambientes. Apesar de ser um conceito amplo, diferentes softwares e fabricantes fazem sua implementação de forma distinta. Em geral, a replicação implica em termos uma fonte de dados primária e réplicas desses dados em outros servidores.

O uso de replicação está normalmente associado a:

- Alta Disponibilidade (HA-High Availability)
- Melhoria de desempenho – distribuir carga entre réplicas
- Contingência – Backup online
- Distribuição Geográfica – Escritórios e filiais

Havendo falha no servidor, a carga do sistema pode ser rapidamente direcionada para uma réplica desse mesmo servidor, garantindo a alta disponibilidade. Já a melhoria de desempenho pode ser alcançada através de Balanceamento de Carga, distribuindo as operações entre as réplicas disponíveis. Outro uso para a replicação é como uma forma de backup, tendo sempre uma cópia online dos dados e desse modo reforçando ainda mais a segurança física do ambiente. Outros arranjos incluem o uso de réplicas distribuídas geograficamente para atender a demanda de diferentes escritórios ou filiais, ou, ainda, a criação de uma réplica para atender exclusivamente demandas de relatórios ou BI.

Existem uma série de ferramentas que fornecem replicação para o PostgreSQL, tais como:

- Slony-I
- Bucardo
- pgPool-II
- pl/Proxy.

Todas essas ferramentas apresentam vantagens e desvantagens, sendo que a maioria demanda uma configuração complexa. O Bucardo, por exemplo, é baseado em triggers para replicar os dados. O pgPool-II funciona como um middleware que recebe as queries e as envia para os servidores envolvidos no "cluster". Já a pl/Proxy é uma linguagem através da qual deve-se escrever funções para operar os dados particionados em nós.

Merece destaque o fato de o PostgreSQL possuir duas formas nativas de replicação: Log Shipping e Stream Replication.

Descreveremos brevemente a primeira e analisaremos mais profundamente a segunda, que fornece a mais moderna e eficiente solução de escalabilidade horizontal do PostgreSQL.

De qualquer modo, ao utilizar replicação, deve-se ter em mente que as alterações executadas no banco master serão aplicadas nas réplicas. Assim, quando se cria um tablespace no servidor principal apontando para determinada área de disco, esse caminho também já deverá existir na réplica.

Outra consideração importante é que, apesar de não ser obrigatório, o uso de replicação com hardware "idêntico" ajuda muito na administração e na resolução de problemas.

Por fim, os servidores envolvidos na replicação devem ter a mesma versão do PostgreSQL.

Log Shipping e Warm-Standby

O PostgreSQL possui nativamente, desde a versão 8.2, a replicação por Log Shipping. Trata-se de um mecanismo largamente utilizado por bancos de dados, que consiste no envio dos arquivos de log de transações, WAL, para que sejam aplicados na réplica. Esse processo é muito semelhante ao mecanismo de PITR que vimos na sessão anterior.

Com a replicação por log shipping é obtido um nível de replicação chamado Warm Standby, onde a réplica está continuamente sendo atualizada pela restauração de arquivos de WAL que estão sendo arquivados pelo servidor máster. Isso se traduz em Alta Disponibilidade, já que o servidor replicado pode ser usado em caso de falha do servidor principal.

Porém, uma réplica Warm Standby não pode ser acessada para operações, estando disponível apenas para casos de falha. Nessa situação a réplica deve ser retirada do estado de restauração e poderá passar a receber conexões com operações sobre os dados. Ou seja, tratará as operações que forem direcionadas para ela, mas não estará mais recebendo alterações sendo feitas em outro servidor. Assim, não podem contribuir para um balanceamento de carga.

Outra questão que deve ser considerada é o tempo de atraso na replicação dos dados. Com o log shipping, os segmentos de WAL são disponibilizados para consumo pela réplica apenas após serem arquivados no servidor principal. Esse arquivamento é feito somente quando um segmento de WAL está cheio (16MB de informações de transações), ou então pelo tempo definido no parâmetro archive_timeout, tipicamente um intervalo de alguns minutos.

Ao diminuir o intervalo de arquivamento, diminuindo o valor de archive_timeout, é possível diminuir a diferença que vai sempre existir entre o servidor principal e suas réplicas. O problema é que a operacionalização disso ficará extremamente cara, pois atingido o intervalo de tempo estipulado em archive_timeout, o restante do arquivo é preenchido até atingir o tamanho de 16MB. Assim, definir archive_timeout para uns poucos segundos causará um consumo enorme de espaço em disco. Isso sem mencionar o tráfego de rede envolvido na transferência das centenas ou milhares de arquivos.

De qualquer modo, a configuração do PostgreSQL para construir uma replicação Log Shipping / Warm Standby é:

- No servidor master, arquivar os segmentos de WAL em um local acessível pelo servidor standby. Isso pode ser feito através do comando *archive_command*:

```
archive_command = 'scp %p postgres@pg02:/archives/%f';
```

- No servidor standby, depois de instalado o PostgreSQL, restaurar um backup base do servidor master, da mesma forma visto em Backup Contínuo: manualmente ou com o pg_basebackup;
- Criar um arquivo *recovery.conf*, com o comando *restore_command* definido com o caminho onde estão os arquivos de WAL, da mesma forma feita para o Backup Contínuo. A única diferença é definir o parâmetro standby_mode = on.

Essa é uma visão geral do processo de replicação por Log Shipping. Como há poucos motivos para usar esse método, não entraremos em um nível de mais detalhes, passando imediatamente a descrever o processo de replicação mais recomendado, que é a replicação por Streaming Replication com Hot Standby.

Streaming Replication com Hot Standby

A partir da versão 9, o PostgreSQL passou a contar com o recurso de Streaming Replication. Essa forma de replicação possui a vantagem de não precisar aguardar o arquivamento dos segmentos de WAL, diminuindo muito o atraso (delay) de replicação entre uma réplica e o servidor principal. A Streaming Replication praticamente acabou com a necessidade de utilizar ferramentas de terceiros para replicação, apresentando vantagens significativas em relação ao método por Log Shipping.

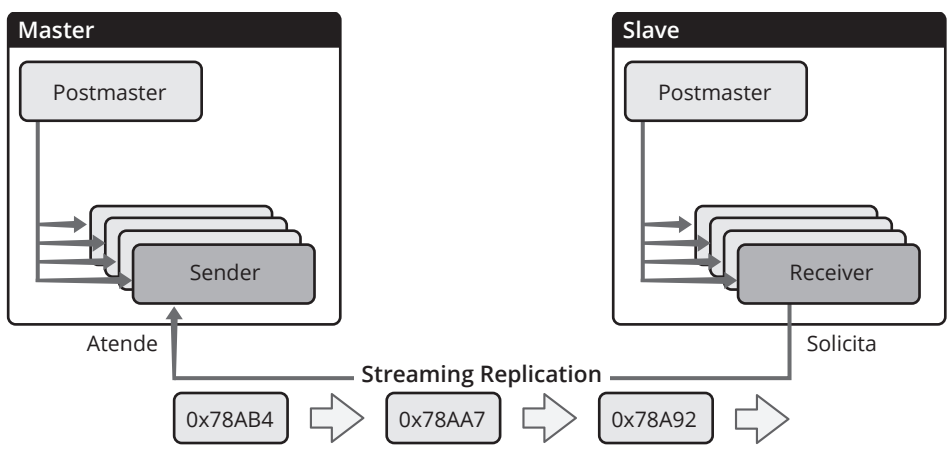


Figura 10.1 Streaming Replication.

Com o Streaming Replication, as informações são replicadas logo após serem comitadas e a transferência é feita pela comunicação entre serviços do próprio PostgreSQL através da rede, sem a necessidade de criação de arquivos intermediários. Por conta disso é conhecido também como replicação binária, evitando todos os contratempos ligados ao arquivamento de WALs e, ao mesmo tempo, qualquer problema proveniente de falta de espaço em disco (para salvar um archive) ou de falha de permissão do filesystem. Outra vantagem é o Hot Standby, que é o nome dado para a capacidade de poder utilizar as réplicas criadas em operações de leitura.

Cada uma dessas etapas é apresentada em mais detalhes a seguir.

A diminuição do tempo de atraso na propagação dos dados e o fato de esse recurso não exigir a instalação de ferramentas ou módulos extra, aliados à simplicidade da sua configuração e ao aumento do poder de escalabilidade através do hot standby, fazem do Streaming Replication a melhor escolha para implementar a replicação de dados.

Preparar o Streaming Replication é muito parecido com o que é feito na replicação por log shipping, com algumas poucas configurações adicionais:

- Configurar o servidor principal para aceitar conexões de replicação;
- Fazer um backup base do servidor principal e restaurar no servidor réplica;
- Criar o arquivo recovery.conf informando dados para conectar com o servidor principal;
- Alterar o postgresql.conf do servidor réplica para torná-lo um hot standby, aceitando conexões.

Configuração do servidor principal

Para permitir o Streaming Replication no servidor principal, devemos permitir conexões do tipo replicação no arquivo *pg_hba.conf,* ajustando também os parâmetros max_wal_senders e wal_level.

Ao editar o arquivo *pg_hba.conf* deve-se permitir uma conexão vinda do servidor réplica nos seguintes moldes:

```
host replication     all 192.168.25.93/32      trust
```

Sem essa liberação, uma máquina réplica não conseguirá se autenticar contra o servidor principal para executar a replicação binária.

No arquivo *postgresql.conf* devem ser feitos os seguintes ajustes:

O parâmetro wal_level, definido como archive para permitir o arquivamento de logs de transação, deve ser redefinido para hot_standby, permitindo assim a realização das duas operações: arquivamento e replicação por streaming.

```
wal_level = hot_standby
```

O parâmetro max_wal_senders define o número máximo de processos senders. Na medida em que cada réplica utiliza um sender, este parâmetro deve refletir a quantidade de réplicas previstas.

```
max_wal_senders = 1
```

Backup Base

Na sessão 9, sobre backup e recuperação, foi apresentada a ferramenta pg_basebackup, que deve ser usada para fazer o backup e a restauração inicial de dados no processo de criação de servidores réplicas.

Conectado no servidor réplica, com o PostgreSQL já instalado e parado: remova, se houver, qualquer conteúdo do pgdata. Se existir alguma área de tablespace fora do pgadata, esta também deve ser limpa.

```
$   rm -Rf /db/data/*
```

Execute o backup base:

```
$ pg_basebackup -h servidor-master -P -D /db/data/
```

Esse comando fará a conexão com o servidor principal (master) e copiará todo o conteúdo do pgdata de lá para o diretório informado; no caso, o pgdata do servidor réplica. O procedimento deve terminar com uma mensagem similar a:

```
NOTICE:  pg_stop_backup complete, all required WAL segments have been archived
```

Criar o arquivo 'recovery.conf'

Após o termino do backup base, mas antes de iniciar o PostgreSQL no servidor réplica, deve-se criar e configurar corretamente o arquivo *recovery.conf*, de forma análoga ao que foi feito em uma restauração PITR. A primeira providência é editar o arquivo r*ecovery.conf* e fazer os seguintes ajustes:

Configurar o parâmetro standby_mode, indicando que o servidor deve executar em modo de recuperação, ou seja, aplicando logs de transação:

```
standby_mode = 'on'
```

Configurar o parâmetro primary_conninfo, indicando a string de conexão com o servidor principal (máster).

```
primary_conninfo = 'host=servidor_master'
```

Nesse argumento, se necessário, pode-se definir, usuário, senha, porta etc.

Por último, informar o caminho do trigger_file. Esse parâmetro indica o nome do arquivo de gatilho usado para interromper o modo standby, ou seja, colocar o banco em modo de leitura e escrita e parar a replicação.

```
trigger_file = /db/data/trigger.txt
```

Configurar o servidor réplica para Hot Standby

O passo final consiste em editar o arquivo de configuração do servidor réplica, o *postgresql.conf*, definindo o parâmetro hot_standby para on. Se isso não for feito, o PostgreSQL executará no modo warm standby, não aceitando conexões.

```
hot_standby = on
```

Desse ponto em diante já é possível executar o PostgreSQL no servidor réplica, lembrando que este deve começar a funcionar em modo de recuperação por conta da presença do arquivo. *recovery.conf.*

Acompanhe o log para entender como o PostgreSQL se comporta e verifique a ocorrência de algum problema. As seguintes mensagens confirmam o modo Hot Standby:

```
LOG:  database system is ready to accept read only connections
LOG:  streaming replication successfully connected to primary
```

Mensagens de arquivo não encontrado não são necessariamente um erro, já que pode ter havido uma tentativa de encontrar o arquivo pelo comando *restore_command.*

```
2014-02-23 21:30:26 BRT [2147]: [3-1] user=,db= LOG:  entering standby mode
2014-02-23 21:30:31 BRT [2147]: [4-1] user=,db= LOG:  restored log file "0000000
20000000200000082" from archive
2014-02-23 21:30:31 BRT [2147]: [5-1] user=,db= LOG:  redo starts at 2/82000020
2014-02-23 21:30:31 BRT [2147]: [6-1] user=,db= LOG:  consistent recovery state
reached at 2/820000C8
2014-02-23 21:30:31 BRT [2145]: [2-1] user=,db= LOG:  database system is ready t
o accept read only connections
scp: /backup/pitr/atual/wals/000000020000000200000083: No such file or directory
2014-02-23 21:30:35 BRT [2173]: [1-1] user=,db= LOG:  streaming replication succ
essfully connected to primary
```

Figura 10.2 Replicação concluída.

Tuning

Foram apresentados os procedimentos básicos para colocar em funcionamento a replicação binária. Mas existem diversos ajustes que podem ser feitos nas configurações dos servidores envolvidos na replicação, especialmente no caso de situações especiais.

Se uma carga de alteração muito grande ocorrer no servidor principal, pode acontecer que os segmentos de WAL precisem ser reciclados antes que os servidores réplica possam obter todas as informações que precisam. Nessa situação poderá surgir uma mensagem de erro como esta:

```
LOG:  streaming replication successfully connected to primary
FATAL:  could not receive data from WAL stream: FATAL:  requested WAL segment
000000020000000200000082 has already been removed
```

Dois ajustes podem ser feitos ajudando a evitar este tipo de erro:

- Ajustar o parâmetro wal_keep_segments para um valor mais alto. Esse é o número mínimo de arquivos de wal que devem ser mantidos pelo servidor master antes de reciclá-los. Observe que estipular um valor muito alto para este parâmetro implicará no uso de grande quantidade de espaço em disco;
- Definir o parâmetro restore_command do recovery.conf no servidor réplica de modo a apontar para a área onde o servidor principal guarda os logs arquivados. Assim, se não for possível obter os arquivos por Streaming Replication, pode-se consegui-los pelo arquivamento do servidor principal.

Outra situação que merece atenção é quando os servidores réplica em modo Hot Standby estão simultaneamente recebendo alterações através do canal de replicação e processando queries de usuários. Em alguns momentos ocorrerão situações em que a replicação precisa alterar um recurso que está sendo usado por uma consulta.

Nesse momento, o PostgreSQL precisa tomar uma decisão: aguardar a query terminar para aplicar a alteração ou matar a query. Por padrão, o banco aguardará 30 segundos, valor definido pelo parâmetro max_standby_streaming_delay.

Definir esse parâmetro é uma escolha entre priorizar as consultas ou priorizar a replicação, optando por um dos seguintes valores:

- **0**: sempre replica, mata as queries;
- **-1**: sempre aguarda as queries terminarem;
- **N**: tempo que a replicação aguardará antes de encerrar as queries conflitantes.

Essa decisão deve considerar a prioridade do seu ambiente. É também possível configurar duas ou mais réplicas, pelo menos uma priorizando a replicação e as demais priorizando as consultas.

Monitorando a replicação

Uma forma simples para saber se a replicação está executando é verificar a existência do processo sender na máquina principal.

```
$   ps -ef | grep -i sender
```

```
/usr/local/pgsql/bin/postmaster -D /db/data
  \_ postgres: logger process
  \_ postgres: checkpointer process
  \_ postgres: writer process
  \_ postgres: wal writer process
  \_ postgres: archiver process   last was 000000020000000200000093
  \_ postgres: stats collector process
  \_ postgres: postgres bench 192.168.25.19(52683) idle
  \_ postgres: wal sender process postgres 192.168.25.95(57388) streaming
2/947A6ECC
```

Figura 10.3
Processo sender executando no servidor principal.

Analogamente, no servidor réplica podemos verificar se o processo receiver está em execução.

```
$   ps -ef | grep -i receiver
```

```
/usr/local/pgsql/bin/postmaster -D /db/data
  \_ postgres: logger process
  \_ postgres: startup process   recovering 000000020000000200000094
  \_ postgres: checkpointer process
  \_ postgres: writer process
  \_ postgres: stats collector process
  \_ postgres: wal receiver process   streaming 2/947A6F50
```

Figura 10.4
Processo receiver executando no servidor réplica.

Em ambos os casos, se a informação do registro de log estiver mudando com o tempo, isso significa que a replicação está em andamento.

Uma alternativa para monitorar a replicação é a função pg_is_in_recovery(), que retorna true se o servidor está em estado de recover, ou seja, é uma máquina standby.

No servidor principal esta função retornará sempre false.

```
postgres=# select pg_is_in_recovery();
```

Uma outra função, que pode ser usada no servidor réplica para exibir o último log record recebido do servidor principal, é:

```
postgres=# select pg_last_xlog_receive_location();
```

- Para saber qual foi a réplica, basta utilizar:

```
postgres=# select pg_last_xlog_replay_location();
```

Por fim, temos o script check_postgres.pl, já mencionado na sessão sobre monitoramento, mais especificamente no monitoramento de replicação pelo Nagios.

Ele compara o total em bytes de diferença entre os servidores informados com os argumentos warning e critical:

```
$check_postgres.pl --action=hot_standby_delay
--host=192.168.25.92,192.168.25.95
--warning=1000
--critical=10000
POSTGRES_HOT_STANDBY_DELAY OK: DB “postgres” (host:192.168.25.95) 0 | time=0.05s r
eplay_delay=0;1000;10000  receive-delay=0;1000;10000
```

Replicação em cascata

A partir da versão 9.2 do PostgreSQL, passou a ser possível fazer a replicação em cascata, ou seja, uma réplica obtendo os dados de outra réplica. Os passos para essa configuração são os mesmos descritos até agora, apenas informando no lugar do servidor master os dados de um servidor réplica como sendo a origem dos dados em *recovery.conf*. A réplica que servirá como origem dos dados deve atender os mesmos requisitos estipulados para um servidor principal, ajustando os seus parâmetros wal_level e max_wal_senders.

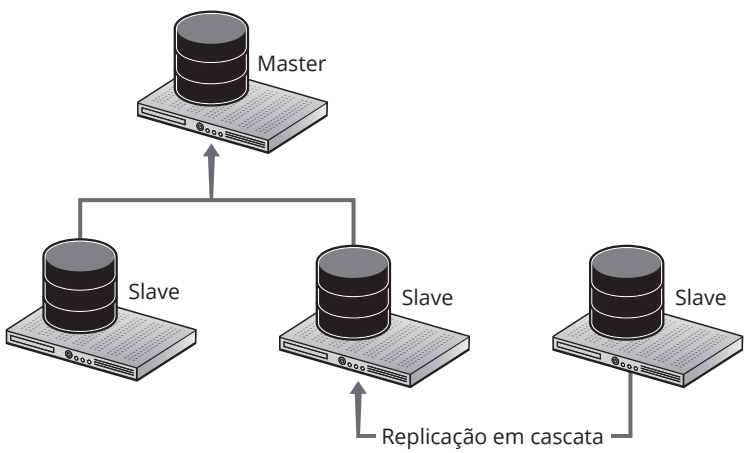


Figura 10.5
Replicação em cascata.

Replicação Síncrona

O funcionamento da replicação mostrada até agora é dito Assíncrono: as alterações são aplicadas na máquina principal de forma totalmente indiferente ao estado das réplicas. Ou seja, a replicação das alterações no servidor réplica é feita de forma assíncrona ao que ela realmente acontece no servidor principal.

Entretanto, é possível habilitar a execução de replicação síncrona com uma única réplica, de tal forma que quando ocorrer um COMMIT no servidor principal essa alteração será propagada para a réplica de forma síncrona. Nessa configuração, a transação original só é concluída após a confirmação de sua execução também no servidor réplica.

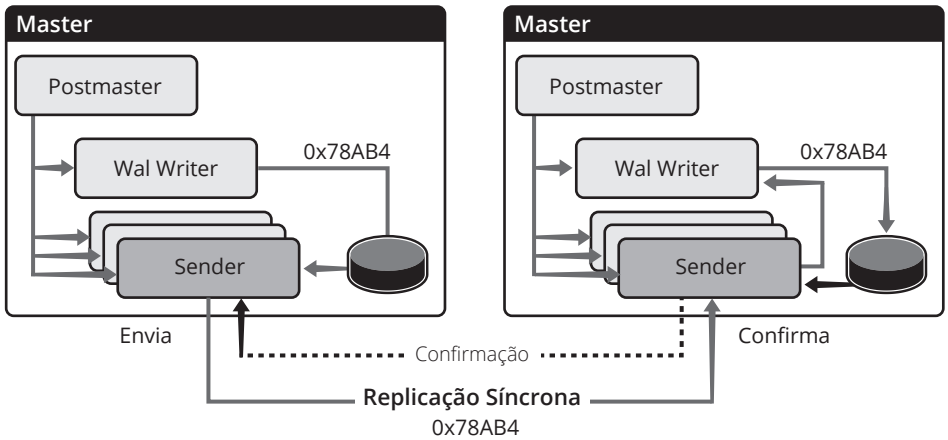


Figura 10.6 Replicação síncrona.

Para configurar a replicação síncrona é necessário apenas definir no parâmetro synchronous_standby_names o servidor que será a réplica síncrona.

> Deve-se ter em mente que o impacto nas requisições de escrita pode ser drástico e causar grande contenção e problemas com locks.

Balanceamento de carga

A infraestrutura de replicação do PostgreSQL fornece uma enorme possibilidade a ser explorada para o crescimento horizontal com carga de leitura. Como a maioria dos sistemas tem uma parcela de leitura de dados muito maior do que a de escrita, os recursos de Streaming Replication e Hot Standby trazem uma solução sob medida para esses cenários.

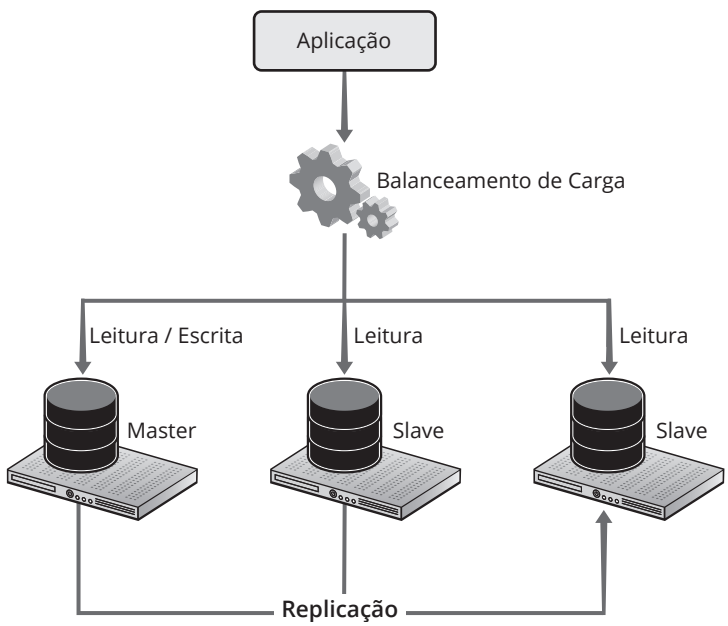


Figura 10.7 Balanceamento de carga.

O desafio passa a ser as aplicações tirarem proveito dessa arquitetura. Uma possibilidade é balancear a leitura em uma ou mais réplicas enquanto escritas são enviadas para um servidor específico. Essa tarefa pode ser executa pela própria aplicação, construída para tirar proveito desse modelo de replicação. Uma alternativa é utilizar um software que faça o papel de balanceador de carga.

Soluções em nível de rede e conexões não podem ser utilizadas, pois o comando sendo executado deve ser interpretado para ser identificado como uma operação de escrita ou leitura. O pgPool-II provê essa funcionalidade de balanceamento. Ele funciona como um middleware e atua de forma transparente para a aplicação (que o enxerga como se fosse o próprio banco de dados).

Por outro lado, o pgPool-II esbarra em uma configuração complexa, já que não é uma ferramenta específica para balanceamento e nem foi pensado para trabalhar com a Streaming Replication nativa do PostgreSQL. De qualquer modo, possui funcionalidade de agregador de conexões bem como recursos próprios para replicação e paralelismo de queries.

Se houver necessidade de escalar operações de escrita, outras soluções, como o Postgres-XC, devem ser consideradas.

O Postgres-XC é um projeto open source de solução de cluster para PostgreSQL, que tem como objetivo fornecer uma solução onde se possa ter tantos nós de escrita quanto forem necessários, fornecendo escalabilidade de escrita que não se pode alcançar com uma única máquina. É um projeto relativamente recente, com uma arquitetura complexa, mas com uma proposta ambiciosa e aparentemente promissora.

Resumo

Configurando um Ambiente Streaming Replication.

- No servidor principal:

```
No pg_hba.conf
    host replication all      192.168.25.93/32   trust
```

- No arquivo *postgresql.conf*, alterar:

```
wal_level = hot_standby
max_wal_senders = 1
```

- No servidor réplica:

Executando e restaurando o backup base.

```
$   pg_clt stop
$   rm -Rf /db/data/*
$   pg_basebackup -h servidor-master -P -D /db/data/
Criar o arquivo recovery.conf
    standby_mode = 'on'
    primary_conninfo = 'host=servidor-master'
    trigger_file = /db/data/trigger.txt
    restore_command='scp postgres@servidor-master:/backup/pitr/atual/wals/%f %p'
```

Editando o arquivo *postgresql.conf*:

```
host_standby = on
Executando a Réplica
$   pg_ctl start
```

Bibliografia

- RAMAKRISHNAN, Raghu; GEHRKE, Johannes. Sistemas de Gerenciamento de Banco de Dados. Ed. McGraw-Hill Interamericana do Brasil, 2008.
- PostgreSQL 9.2 Documentation http://www.postgresql.org/docs/9.2/static/index.html
- SMITH, Gregory. PostgreSQL 9.0 High Performance. Packt Publishing, 2010
- The PostgreSQL Global Development Group. The PostgreSQL 9.0 Reference Manual, Volume 3: Server Administration Guide. Network Theory LTD, 2010
- RIGGS, Simo; KROSING Hannu. PostgreSQL 9 Administration Cookbook. Packt Publishing

Fábio Caiut É graduado em Ciência da Computação na Universidade Federal do Paraná (2002) e Especialista em Banco de Dados pela Pontifícia Universidade Católica do Paraná (2010). Trabalha com TI desde 2000, atuando principalmente em Infraestrutura e Suporte ao Desenvolvimento. É Administrador de Banco de Dados PostgreSQL há 5 anos no Tribunal de Justiça do Paraná, focado em desempenho e alta disponibilidade com bancos de dados de alta concorrência. Tem experiência como desenvolvedor certificado Lotus Notes e Microsoft .NET nas softwarehouses Productique e Sofhar, DBA SQL Server e suporte à infraestrutura de desenvolvimento na Companhia de Saneamento do Paraná.

esr.rnp.br

LIVRO DE APOIO AO CURSO

O objetivo do curso é o desenvolvimento das competências necessárias para administrar o PostgreSQL - sistema gerenciador de banco de dados em software livre que é considerado um dos mais completos e robustos do mercado. Será apresentada a arquitetura geral deste SGBD, conceitos e práticas para tarefas de instalação, operação e configuração. O aluno aprenderá técnicas de monitoramento e otimização de desempenho (tuning), bem como rotinas comuns de manutenção do ambiente, incluindo aspectos de segurança, backup e recuperação de dados. O curso também abordará alternativas de replicação de dados para distribuição de carga e alta disponibilidade.

ISBN 978-85-63630-51-3
9 788563 630513