Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Informática

DBMS-Analyzer: Um Framework para Análise Holística de Desempenho de SGBDs

Mestrando
Camilo Porto Nunes
nunes@dsc.ufcg.edu.br

Orientadores

Cláudio de Souza Baptista, PhD.
baptista@dsc.ufcg.edu.br

Marcus Costa Sampaio, Dr.
sampaio@dsc.ufcg.edu.br

Campina Grande
Julho — 2008

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Informática

DBMS-Analyzer: Um Framework para Análise Holística de Desempenho de SGBDs

Camilo Porto Nunes

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Sistemas de Informação e Banco de Dados

Cláudio de Souza Baptista, PhD.
(Orientador)
Marcus Costa Sampaio, Dr.
(Orientador)

Campina Grande, Paraíba, Brasil

N972d

2008 Nunes, Camilo Porto

DBMS-Analyzer: um framework para análise holística de desempenho de SGBDs / Camilo Porto Nunes. – Campina Grande, 2008.

139f. : il. Color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Informática e Engenharia Elétrica.

Referências.

Orientadores: Prof. Dr. Cláudio de Souza Baptista, Prof. Dr. Marcus Costa Sampaio.

1. Banco de Dados Autônomos. 2. PostgreSQL. 3. Gerência de Memória. I. Título.

CDU – 004.65(043)

"DBMS-ANALYZER: UM FRAMEWORK PARA ANÁLISE HOLÍSTICA DE DESEMPENHO DE SGBDS"

CAMILO PORTO NUNES

DISSERTAÇÃO APROVADA EM 29.07.2008

PROF. CLÁUDIO DE SOUZA BAPTISTA, Ph.D
Orientador

PROF. MARCUS COSTA SAMPAIO, Dr.
Orientador

PROF. ULRICH SCHIEL, Dr.
Examinador

PROF. SÉRGIO LIFSCHITZ , Dr.
Examinador

CAMPINA GRANDE – PB

Resumo

Os sistemas de gerência de banco de dados vêm se tornando cada vez mais complexos. Com o crescimento dessa complexidade, também cresce o custo de manter esse software funcionando satisfatoriamente a seus usuários, principalmente o seu desempenho. Para reduzir este custo, a computação autônoma propõe a autonomia dos sistemas de maneira que eles executem tarefas de gerência de forma automática, reduzindo a intervenção humana no processo de gerência.

Esta dissertação apresenta um framework de gerência automática de desempenho de SGBDs que utiliza redes de filas e análise operacional para avaliar o desempenho desses softwares e detectar eventuais problemas. O framework implementa o ciclo básico de gerência automática, característico da computação autônoma, que possui quatro etapas básicas: monitorar, analisar, planejar e executar.

Também é proposta, nesta dissertação, uma estratégia para ajustar automaticamente as estruturas de memória do SGBD PostgreSQL. O algoritmo de ajuste leva em conta características da carga de comandos SQL a que o PostgreSQL está submetido, tais como freqüência de acesso às tabelas do banco de dados, tamanho dessas tabelas, tamanho de índices, freqüência de comandos que exijam ordenação, dentre outras. A estratégia de ajuste foi implementada como uma extensão do framework para gerenciar o desempenho do PostgreSQL.

Testes elaborados para a extensão do framework mostram que ele foi capaz de reduzir em pelo menos 16% o tempo de resposta dos comandos SQL submetidos ao PostgreSQL, apenas ajustando seus parâmetros shared_buffer e work_mem.

Palavras chaves: Banco de Dados Autônomo; PostgreSQL; Gerência de Memória.

Abstract

Database Management Systems (DBMS) are becoming more complex. As a consequence, the cost to maintain this software at satisfactory levels is also enhancing. In order to reduce this cost, the autonomic computing field has been investigated, so that complex system may incorporate self-healing, self-management, self-tuning, and so on, aiming to reduce human intervention in administrating such systems, and of course the overall costs.

This dissertation presents a framework for self-tuning databases, called as DBMS-Analyzer, which is based on queue networks and operational analysis. DBMS-Analyzer implements the basic cycle for self-management database, which comes from autonomic computing. This cycle has four steps: monitoring, analyzing, planning and executing.

Furthermore, this dissertation proposes an approach for self-tuning of memory structures of the PostgreSQL DBMS. The self-tuning algorithm proposed takes into account the loading of SQL statements submitted to the PostgreSQL. This load includes, but it is not limited to, the number of table accesses, table sizes, index sizes, and number of sortings. The self-tuning algorithm was implemented as an extension of the DBMS-Analyzer framework in order to monitor the PostgreSQL DBMS performance.

Tests were executed to validate the proposed framework and the results demonstrated that there was a reduction of up to 16% in response time of SQL statements, using the TPC-Benchmark. This reduction was achieved by just tuning the PostgreSQL shared_buffer and work_mem parameters.

Key Words: Self-Tuning Databases; PostgreSQL; Self-Tuning Memory Management.

Agradecimentos

A Deus, por me dar força e perseverança para concluir este trabalho.

A minha família, por me animar nos momentos de fraqueza.

A minha noiva Elita, por aturar meu estresse e minhas lamúrias nos momentos difíceis.

Aos orientadores Cláudio Baptista e Marcus Sampaio, por guiar meu trabalho e me dar bastante experiência.

Aos amigos da faculdade, em especial Mirna e Laisa, pelos momentos de descontração e por me ajudarem nas horas de desespero.

Aos meus padrinhos Edivaldo e Maria das Neves (Guia), pela hospedagem 5 estrelas durante o período que morei em Campina Grande.

Aos primos, em especial Arthur e Allan, pelas longas *Happy Hours* nas horas de folgas.

A Aninha (Copin), por facilitar tanto a minha vida.

E a todos que me ajudaram a seguir o meu caminho e concluir este mestrado.

Conteúdo

Lista de Símbolos

TCO - *Total Cost of Ownership*

TI - *Tecnologia da Informação*

SGBD - *Sistema de Gerência de Banco de Dados*

DBA - *Database Administrator*

SQL - *Structured Query Language*

OLAP - *Online Analytical Processing*

OLTP - *Online Transaction Processing*

LRU - *Least Recently Used*

DAT - *Data Access Time*

DRF - *Dynamic Reconfiguration Algorithm*

JDBC - *Java Database Conectivity*

SNMP - *Simple Network Management Protocol*

SSH - *Secure Shell*

TPC - *Transaction Processing Performance Council*

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Ao longo dos anos, os sistemas computacionais vêm se tornando cada vez mais complexos. Desde os sistemas de processamento de dados em *mainframes*, até os atuais sistemas amplamente distribuídos, novas tecnologias, novas funções, novos dispositivos e novos componentes foram incorporados.

Devido à grande pressão exercida pelo mercado, as arquiteturas dos sistemas baseiam-se, cada vez mais, em *web services*, Internet, servidor de aplicação, servidor de banco de dados, processadores de transação, *clusters*, *storages*, dentre outros. Além disso, os atuais sistemas são acessados cada vez mais por *browsers*, celulares, PDAs, e outros sistemas.

Com tanta complexidade, os atuais sistemas computacionais demandam um grande volume de investimentos para o seu funcionamento satisfatório. Tais investimentos vão desde a aquisição e instalação do *software* até a manutenção, administração e atualização, que envolvem, dentre outras coisas, pessoal cada vez mais especializado. Todo esse custo, denominado *total cost of ownership* (TCO), tende a crescer de forma exponencial à complexidade adicionada ao software.

Esta situação é altamente indesejável aos usuários de Tecnologia da Informação (TI), uma vez que o foco da TI deve ser fornecer serviços de informação básicos que ajudem o negócio de seus usuários sob a forma de facilidade de acesso a informação, agilidade, simplicidade e competitividade. Em outras palavras, usuários de TI não querem direcionar o melhor de seus esforços para administração de TI, mas sim, utilizá-la como ferramenta de apoio ao negócio de sua empresa ou instituição.

Notando esta tendência de os sistemas agregarem cada vez mais complexidade, a co-

munidade científica e as grandes empresas uniram-se para criar um novo paradigma de desenvolvimento de sistemas computacionais: a computação autônoma (do inglês, *autonomic computing*).

Segundo *Kephart* [KC03], esse novo paradigma deve focar o desenvolvimento de sistemas no usuário final, ou seja, aquele que vai interagir com o *software*. O fundamento desse paradigma é que o usuário final adquire um software com o principal objetivo de usá-lo, e não de administrá-lo. Idealmente, os sistemas seriam implementados de modo a interagir, de forma automática e transparente, com outros componentes de *software* proporcionando o ambiente e funcionalidades adequadas ao uso satisfatório pelos seus usuários. O sistema ideal também se adaptaria automaticamente a mudanças ocorridas, tais como falha de um componente, aumento de carga no sistema, falha de segurança, atualização de componentes, dentre outras.

Todos esses requisitos reunidos permitirão atingir o principal objetivo da computação autônoma [KC03]: *system-wide policy-based self-management system*, ou seja, um *software* em que o usuário define um nível de serviço aceitável e o sistema se encarrega de se auto configurar (*self-configuring*) e ajustar automaticamente seu desempenho (*self-optimizing*) para atender aos níveis de serviço acordados. O sistema também se encarregará de garantir a sua disponibilidade e acessibilidade (*self-healing*), além da sua segurança (*self-protecting*).

Sistemas de Gerência de Banco de Dados (SGBD) são componentes cruciais de um sistema computacional, uma vez que o maior valor de uma empresa são os seus dados, pois de um SGBD são extraídas informações que podem trazer um diferencial competitivo para o negócio de um empresa.

Os SGBDs tornaram-se bastante complexos ao longo dos anos, isto é, deixaram de ser meramente um *software* que oferece acesso, integridade e segurança a dados, para se tornarem um sistema altamente interoperável com componentes de acesso à Internet, integração com outros SGBDs, *web services*, servidores de aplicação e portais.

Segundo *Sam Lightstone* et. al.[LSZK03], essa evolução traz consigo dificuldades em gerir e manter um *software* deste porte. Isso acarreta impactos consideráveis nas atividades desempenhadas pelo responsável por um SGBD, o Administrador de Banco de Dados (do inglês, *Database Administrator* - DBA). Em princípio, um DBA deveria conhecer, de forma profunda, todos os novos serviços e componentes arquiteturais de um particular SGBD para

que fosse possível desempenhar sua atividade de forma satisfatória. O grande problema é que nem sempre isso é possível.

A alta complexidade dos atuais SGBDs está diminuindo drasticamente a capacidade humana de administrá-los [LSZK03]. Os seres humanos, as metodologias de gestão e as tradicionais ferramentas de administração não acompanham o crescimento desta complexidade. Neste sentido, existe uma forte tendência de os SGBDs tornarem-se cada vez mais autônomos [LSZK03]. O ramo da computação autônoma que investiga a autonomia em SGBDs é denominado banco de dados auto-gerenciáveis (do inglês, *self-management databases*) [LSZK03].

Em particular, existe um ramo de *self-management databases* com grande preocupação em desempenho de SGBDs. Esse ramo, denominado *self-tuning databases*, preocupa-se em tornar automática a atividade de gerência de desempenho dos SGBDs. Essa atividade envolve, dentre outras, o diagnóstico automático de problemas de desempenho, a gerência automática de índices e a gerência automática de memória. Para auxiliar a contextualização do tema desta dissertação, é apresentada, na figura 1.1, parte dos ramos da computação autônoma aplicados a SGBDs. O foco desta dissertação é a gerência automática de memória.

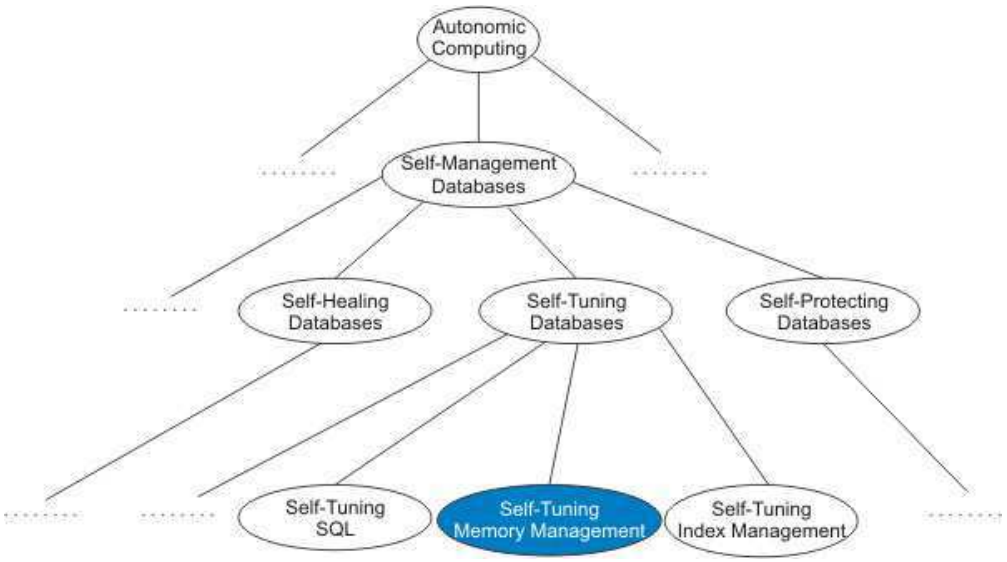

Figura 1.1: *Ramos da computação autônoma aplicados a SGBDs*

Um dos recursos mais utilizados e, conseqüentemente, mais consumidos pelos SGBDs é a memória RAM. Este fato decorre da enorme diferença existente entre o tempo de acesso a um dado em disco e o tempo de acesso ao mesmo dado na memória. Este segundo é

bem menor do que o primeiro. Por este motivo, o SGBD procura disponibilizar na memória RAM vários objetos "de valor", ou seja, aqueles mais freqüentemente acessados, para evitar acessos a disco [Mul02]. Ademais, algumas áreas da memória são alocadas e reservadas para algum tipo especial de processamento, tais como ordenação de dados, armazenamento temporário, dentre outros.

Uma tarefa de grande importância desempenhada pelo DBA é o ajuste dessas áreas de memórias alocadas para o SGBD, os chamados *buffers* de memória (ou *pools* de memória). Essa atividade não é simples de ser efetuada, pois a memória RAM utilizada pelo SGBD é limitada. O grande problema consiste em, dada uma carga de comandos submetida ao SGBD, uma quantidade fixa de memória e um conjunto de *buffers*, determinar o tamanho de cada *buffer* de modo a otimizar o processamento da carga submetida ao SGBD. Isso exige muita experiência do DBA e, muitas vezes, essa experiência não é suficiente para ajustar os *buffers* de memória satisfatoriamente.

Ademais, o tempo necessário para o DBA efetuar uma análise da situação do SGBD e, em seguida, decidir qual ajuste deve ser feito é bastante longo em relação ao tempo aceitável para que o SGBD seja adaptado. Isso se deve ao grande número de recursos e componentes de software que são utilizados pelo SGBD, dentre os quais destacam-se a CPU, o disco, o *cache* de dados, o *cache* de comandos, o otimizador de consultas, o compilador de comandos, dentre outros. Uma análise rápida, detalhada e manual de todos esses componentes para diagnosticar quais deles mais estão contribuindo para a degradação de desempenho do SGBD (quais deles são os gargalos do sistema) é, muitas vezes, inviável. A detecção rápida do componente do SGBD que mais está contribuindo para a degradação de seu desempenho é de suma importância para apoiar a decisão do ajuste que deve ser procedido no sistema para restabelecimento de seu desempenho.

1.1 Objetivos

Esta dissertação almeja contribuir para o amadurecimento da atividade de gestão automática de desempenho de SGBDs através do desenvolvimento de um *framework* de gerência automática de desempenho de SGBDs que analise o desempenho desse sistema, de forma holística e, caso necessário, proceda um ajuste a fim de melhorar o seu desempenho.

1.1.1 Objetivos Gerais

Desenvolver um *framework* de gerência automática de desempenho de SGBDs. Nas propostas apresentadas na literatura, os componentes de gestão automática de desempenho dificilmente são reaproveitados em outro SGBD. Cada componente foi desenvolvido para executar em um SGBD específico. Há então a necessidade de elaboração de um gerenciador automático de desempenho que seja extensível para outros SGBDs.

1.1.2 Objetivos Específicos

- **Estender o *framework* para gerenciar o desempenho do PostgreSQL através de ajustes em suas estruturas de memória**. A maioria dos trabalhos em gestão automática de memória (e *self-management database* em geral) são implementados em SGBDs comerciais, tais como Oracle, DB2 e SQL Server. SGBDs livres como o PostgreSQL e o MySQL Server são pouco visados, deixando de lado a enorme comunidade de usuários e desenvolvedores adeptos desses SGBDs.

- **Validar o *framework* desenvolvido com o uso do Benchmark TPC-C [(TP07]**. Esse Benchmark será utilizado por ser um padrão largamente utilizado pela indústria de SGBDs para medir desempenho desse tipo de *software*.

1.1.3 Motivação

A gerência automática de memória ainda é uma atividade de vital importância pois, embora o custo de memória RAM tenha baixado bastante nos últimos anos, várias entidades ainda sofrem com escassez de recursos para aquisição de equipamentos de TI; por exemplo, órgãos governamentais, principalmente os municipais. Sendo assim, uma ferramenta que auxilie essas entidades a gerenciar a memória RAM de seus SGBDs a fim de beneficiar suas atividades será de grande valia.

A tarefa de diagnóstico de problemas de desempenho também é de suma importância, pois ela precede todas as atividades de ajuste (primeiro é preciso detectar um problema para posteriormente corrigi-lo). Nesse sentido, o desenvolvimento de um diagnosticador reusável e extensível para diferentes SGBDs torna-se de grande importância, pois suas decisões de

diagnóstico servirão de apoio para todas as decisões de ajustes em diferentes SGBDs.

1.2 Estrutura da Dissertação

Os demais capítulos desta dissertação estão organizados da seguinte forma:

- **Capítulo 2**: apresenta a fundamentação teórica necessária para uma boa compreensão do trabalho desenvolvido. Seu conteúdo aborda: conceitos da computação autônoma, seu propósito e o ciclo básico utilizado para gerência automática de sistemas; *self-tuning databases*, apresentando como os conceitos da computação autônoma são aplicados a gerência automática de SGBDs, bem como alguns aspectos inerentes a administração automática desse tipo de *software*; uma visão geral da arquitetura de SGBDs, focando o processamento de comandos em um servidor de banco de dados assim como os recursos de *hardware* e *software* utilizados durante esse processo; conceitos relevantes de rede de filas e análise operacional - um framework de gerência de desempenho de sistemas - e como esse modelo pode ser aplicado na gerência automática de SGBDs; uma visão geral da arquitetura do SGBD PostgreSQL, enfatizando seu processamento de comandos e suas estruturas de memória;

- **Capítulo 3**: discorre acerca dos principais trabalhos relacionados à gerência automática de memória em SGBDs. Também apresenta uma discussão sobre os requisitos desejados para uma ferramenta de gerência automática de memória e quais desses requisitos são atendidos pelos trabalhos apresentados.

- **Capítulo 4**: apresenta o *framework* implementado na dissertação. Está dividida em seis partes: a primeira parte apresenta uma visão geral do *framework* e introduz as funções de seus quatro componentes básicos (gerente de monitoramento, análise, planejamento e ajuste). Nas etapas seguintes, são detalhados os componentes do *framework* e apresentada uma discussão acerca de seus aspectos de extensibilidade.

- **Capítulo 5**: apresenta a extensão do *framework* para o SGBD PostgreSQL, introduzindo o modelo de rede de filas criado para o SGBD; o mapeamento elaborado entre as métricas presentes no arquivo de log do PostgreSQL e as variáveis operacionais das

filas do modelo de rede de filas elaborado; as estratégias de ajuste implementadas para dimensionar o *shared buffer* e a *work memory* do referido SGBD.

- **Capítulo 6**: apresenta os resultados obtidos com os testes da aplicação desenvolvida para o PostgreSQL. O SGBD foi submetido a diferentes tipos de situações e os ajustes sugeridos pelo *framework*, bem como os impactos causados no desempenho do SGBD foram coletados e analisados.

- **Capítulo 7**: conclui o documento com uma avaliação dos resultados obtidos e um conjunto de trabalhos que podem ser desenvolvidos no futuro.

Capítulo 2

Referencial Teórico

O presente capítulo apresenta alguns conceitos básicos necessários a uma melhor compreensão desta dissertação. A seção 2.1 inicia o capítulo apresentando os conceitos fundamentais da computação autônoma. Em seguida, a seção 2.2 discorre sobre a aplicação da computação autônoma em SGBDs. A seção 2.3 apresenta um modelo genérico de arquitetura de SGBDs enfatizando os principais componentes desse software. Conceitos básicos sobre redes de filas são apresentados na seção 2.4, enquanto a seção 2.5 apresenta os conceitos da análise operacional. O capítulo encerra-se com a seção 2.6 discutindo os principais aspectos da arquitetura do SGBD PostgreSQL.

2.1 Computação Autônoma

A computação autônoma é um novo paradigma de desenvolvimento de software que visa a incorporar aos novos sistemas a capacidade de se auto-gerenciar, administrando seu desempenho, sua segurança, sua disponibilidade e sua configuração, de acordo com objetivos definidos pelo administrador do sistema.

A inspiração para a computação autônoma vem do sistema nervoso central do corpo humano [KC03]. Funções básicas e vitais do corpo humano são executadas de forma inconsciente e automaticamente. Durante um exercício físico, por exemplo, o ser humano não se preocupa em respirar, ajustar o ritmo do batimento cardíaco e ajustar a temperatura do corpo. Ele simplesmente concentra-se no exercício. As funções básicas necessárias a sua execução são desempenhadas de forma automática pelo corpo humano e coordenadas pelo

sistema nervoso central.

Para que isso seja possível, o sistema nervoso central desempenha um papel fundamental e de grande importância: é o responsável por monitorar o corpo humano (batimento cardíaco, temperatura, respiração), analisar as mudanças ocorridas e enviar mensagens (estímulos nervosos) para efetuar um eventual ajuste em cada componente do corpo humano a fim de proporcionar um ambiente adequado para o desempenho de atividades (correr, pular, estudar, dormir, praticar esportes, dentre outras).

Segundo *Sterritt* et. al. [SPTU05], para um software tornar-se autônomo, ele deve incorporar em suas características alguns requisitos, a saber: *self-governing*, *self-adaptation*, *self-organization*, *self-optimization*, *self-configuration*, *self-diagnosis of fault*, *self-protection*, *self-healing*, *self-recovery*. Todavia, segundo a IBM [KC03], os principais requisitos resumem-se a: *self-configuring*, *self-healing*, *self-optimizing* e *self-protecting*.

Self-configuring significa que o sistema autônomo é capaz de gerenciar sua própria configuração [KC03]. Essa característica é importante pois, durante a instalação do software, por exemplo, o usuário final não precisa se preocupar em fornecer informações sobre o ambiente de instalação. O próprio sistema irá descobrir essas informações e selecionar a configuração que melhor se adapta ao ambiente. Caso alguma mudança ocorra no ambiente de execução do sistema, este deve, automaticamente, alterar suas configurações para se manter funcionando de forma satisfatória.

Outra característica dos sistemas autônomos é *self-healing*. Esse conceito está ligado à noção de tolerância a falhas, uma vez que, um sistema *self-healing* deve detectar uma falha que o torne incapaz de continuar operando, ou o deixe operando de maneira insatisfatória, e tomar uma decisão para contornar o problema e continuar operando de maneira minimamente satisfatória [KC03].

Self-protecting, por sua vez, está ligado à segurança. O sistema com essa característica deve promover segurança, privacidade e proteção aos dados [KC03]. Isso inclui, dentre outras coisas, que o sistema deve: detectar uma possível invasão e reagir de maneira a evitar danos; e atualizar-se sempre que necessário. Tudo isso de maneira automática e transparente ao usuário final.

O último requisito de um sistema autônomo, segundo a IBM, é *self-optimizing*. Esse requisito determina que um sistema autônomo deve tomar iniciativas que mantenham o seu

desempenho aceitável ao usuário final [KC03]. Tais iniciativas incluem gerenciar o uso de disco, de memória RAM, de CPU, dentre outros recursos de hardware e software.

Para atingir esses quatro requisitos, a maioria dos componentes autônomos dos sistemas utiliza o ciclo básico de ajuste da computação autônoma proposto em *Kephart* [KC03]. Esse ciclo básico é utilizado por um gerente autônomo, responsável por gerenciar um sistema qualquer, e é composto de quatro fases: monitorar, analisar, planejar e executar; como ilustra a figura 2.1

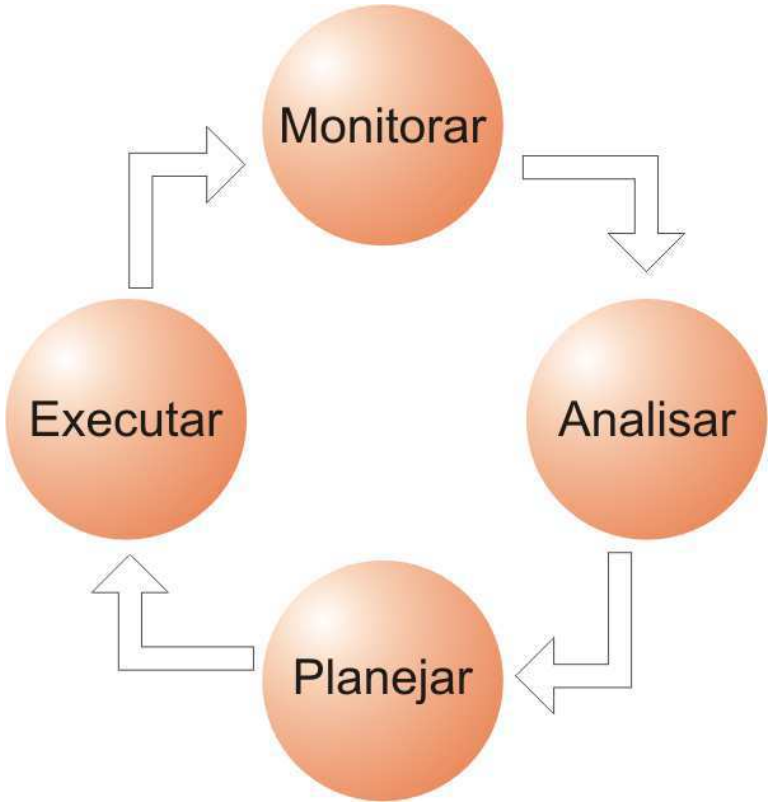

Figura 2.1: *Ciclo Básico da Computação Autônoma*

A primeira fase, o monitoramento, consiste na coleta de informações do sistema gerenciado e do ambiente onde ele é executado. Tais informações podem incluir uso de memória, uso de CPU, taxa de ocupação de disco, vazão do sistema, tempo de resposta, dentre outras.

A fase de análise consiste na manipulação das informações coletadas no monitoramento a fim de avaliar a situação do sistema gerenciado. Através da análise dessas informações, o gerente autônomo pode detectar problemas no sistema gerenciado e criar um plano de ação para corrigi-los ou evitá-los.

A próxima fase, o planejamento, visa a criar um plano de ação, com base nas informações

coletadas na etapa de monitoramento e nos problemas detectados na fase de análise, que corrija eventuais problemas no sistema gerenciado.

Por último, na fase de execução, o plano criado na fase anterior é implementado e o sistema gerenciado tem suas configurações ajustadas.

Todo esse ciclo básico é guiado por metas e objetivos definidos pelos usuários finais dos sistemas gerenciados ou seus administradores. Dessa forma, o objetivo final da computação autônoma é desenvolver sistemas onde um ser humano determina metas e objetivos a serem satisfeitos pelo software e este se encarrega de se auto-gerenciar a fim de atingir tais objetivos.

2.2 Self-Tuning Databases

Self-tuning database é considerado um ramo da computação autônoma que se preocupa em tornar automática a gerência de desempenho em SGBDs. Os mesmos princípios, requisitos e conceitos da computação autônoma também são válidos para *self-tuning database*, porém, existem atividades que são específicas desse ramo da computação autônoma e, algumas delas, serão discutidas brevemente nesta seção.

2.2.1 Detecção Automática de Problema de Desempenho

A atividade fundamental para *self-tuning database* é a detecção automática de problema de desempenho. É uma atividade primária, pois, a partir dela, outras atividades, tais como gerência de índice e memória podem entrar em ação. A detecção automática de problemas de desempenho em SGBDs envolve, dentre outras coisas, a coleta de estatísticas do SGBD, tais como número de requisições de I/O, tempo médio de resposta das transações, vazão do sistema. Uma forte tendência para esta atividade é o uso de uma única unidade de medida para medir o desempenho dos recursos do SGBD e a hierarquização dos recursos utilizados.

A unificação de uma unidade de medida visa a eliminar a dificuldade em relacionar métricas com unidades de medidas diferentes, tais como, uso de CPU, taxa de ocupação em disco, requisições de I/O por segundo, número de transações por segundo, tempo médio de resposta, dentre outras. Freqüentemente, a unidade de medida unificada é definida em termos de tempo gasto de processamento do SGBD, ou seja, a taxa de ocupação de CPU, o

número de requisições de I/O não são medidos em termos de percentuais ou requisições por segundos, mas sim, em termos de quanto tempo de processamento esses recursos consomem do SGBD. Um exemplo de unidade de medida unificada, denominada *DBTime* [DRS+05], é apresentada na figura 2.2. O *DBtime* representa o tempo total que o SGBD passou processando as transações dos usuários.

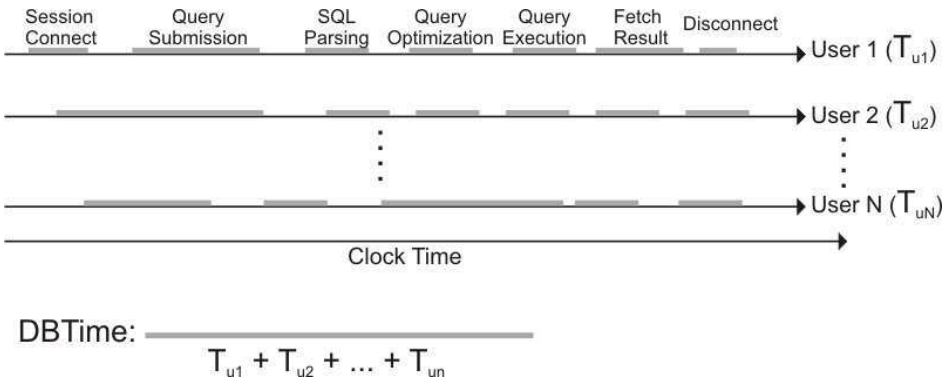

Figura 2.2: *Exemplo de unificação de unidade de medida*

A hierarquização dos recursos utilizados é útil para detectar, de forma rápida e fácil, o recurso que mais está consumindo tempo de processamento do SGBD, ou seja, o gargalo do sistema. Essa hierarquia é criada distribuindo o tempo de processamento total do SGBD entre os diversos recursos utilizados por ele. Cada recurso pode ser subdivido em outros recursos. Em *Dias* et. al. [DRS+05] o *DBTime* é dividido entre as etapas de processamento de comandos SQL. O *DBTime* de cada uma dessas etapas é subdivido em termos do tempo que cada uma delas consome efetuando I/O, usando a CPU e requisitando dados em Memória. A figura 2.3 ilustra um exemplo de uma hierarquia de recursos.

Através do grafo da figura 2.3 é possível detectar qual o gargalo do sistema através de um simples caminhamento na árvore. Em cada nível da hierarquia seleciona-se o nó que mais está consumindo tempo de processamento do SGBD. O processo é repetido com os filhos do nó escolhido no passo anterior e termina quando um nó folha é atingido. Este nó folha será o gargalo do sistema e o recurso por ele representado deve ser ajustado para melhorar o desempenho do SGBD. Na figura 2.3, por exemplo, o nó que representa as atividades de I/O (em destaque) da etapa *Query Execution* seria o gargalo do sistema.

Mais informações sobre o problema de detecção automática de problema de desempenho podem ser encontradas em [Ben05; DRS+05; DD06].

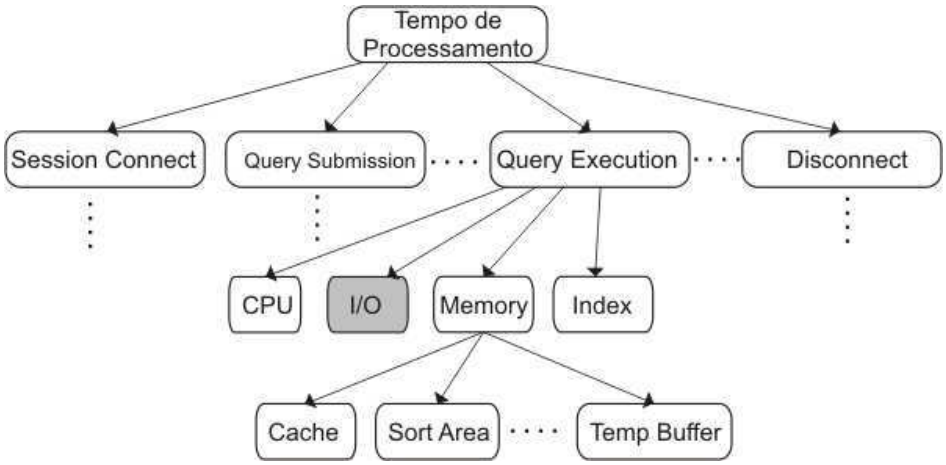

Figura 2.3: *Exemplo de hierarquia de recursos do SGBD*

2.2.2 Gerência Automática de Índices

Com relação à gerência automática de índices, essa atividade também é de vital importância, pois a criação de um índice, possivelmente, é a decisão que ocasiona um maior impacto no desempenho do SGBD [Mul02]. Um índice é uma estrutura de dados que agiliza o acesso a dados em tabelas de banco de dados.

A criação de um índice pode auxiliar na execução de uma consulta submetida ao SGBD diminuindo o número de acessos a disco necessários para buscar os dados utilizados na consulta. O SGBD, em vez de efetuar uma busca seqüencial em disco - que demanda vários acessos a disco, utiliza o índice como um atalho para achar os dados de que necessita.

Por outro lado, um índice pode degradar o desempenho de um comando SQL submetido ao SGBD, pois, se houver a necessidade de remover ou alterar um dado em disco, o conteúdo do índice que referencia esse dado terá que ser reorganizado para refletir a mudança. Esta operação pode afetar o tempo de execução de algumas transações que fazem muita alteração nos dados.

A idéia básica, utilizada na gerência automática de índices, consiste em submeter ao SGBD uma carga de comandos SQL para a qual se deseja ajustar o banco de dados. O gerente automático de índices então analisa as características dos comandos submetidos (predicados utilizados nas consultas, tabelas envolvidas, atributos envolvidos, junções, dentre outros aspectos) e poderá desencadear duas decisões: 1) sugerir a criação ou remoção de índices para que o DBA possa acatar ou não as sugestões, ou 2). criar ou remover, automa-

ticamente sem a intervenção do DBA, os índices sugeridos pelo gerente. Note que, nessa segunda decisão, não existe nenhuma interação entre o gerente de índice e o DBA.

Para averiguar se um determinado índice provocará um ganho de desempenho na execução de um comando SQL, os gerentes de índices utilizam o otimizador de consultas e índices virtuais. O otimizador de consultas, dentre outras coisas, é o responsável por avaliar se um determinado índice deve ou não ser utilizado na execução de uma consulta. O procedimento é executado da seguinte forma: para cada comando SQL da carga submetida ao SGBD, o gerente de índice analisa sua estrutura e seleciona um conjunto de índices, denominado conjunto de índices candidatos, que poderão acarretar um ganho de desempenho. As informações desses índices candidatos são inseridas no catálogo do SGBD, porém sua estrutura de dados não é criada (daí o nome de índices virtuais). Esse procedimento é adotado apenas para que os índices virtuais sejam levados em conta na análise do otimizador de consultas. Caso, durante a geração do plano de execução pelo otimizador de consultas, o uso de um dos índices virtuais esteja presente, o referido índice será sugerido pelo gerente de índices para ser implantado pelo DBA, ou será automaticamente criado.

Segundo *Weikum* [WMHZ02], o problema de gestão automática de índices é um problema de média complexidade, porém, que está bem resolvido devido a grande quantidade de esforços voltados para o tema [Sal04; dCCLdNS05; DDD+04; SAMP06; ACK+04; SGS03; SSG05]. A gestão de índice é, notadamente, um dos temas mais pesquisados em *self-tuning databases*.

2.2.3 Gerência Automática de Memória

A gerência automática de memória é outra atividade fundamental para a gerência automática de desempenho de SGBDs. O problema dessa atividade pode ser formulado como: dada uma carga de comandos SQL submetida ao SGBD, uma quantidade fixa de memória e um conjunto de *buffers*, determinar o tamanho de cada *buffer* de modo a otimizar o processamento da carga submetida.

Buffers de memória são áreas da memória do computador utilizadas para armazenar dados. Os SGBDs utilizam vários *buffers* para, por exemplo, armazenar planos de execução de consultas, fazer *cache* de dados, *cache* de comandos, efetuar ordenação de dados e armazenar dados de sessão de usuários conectados [Mul02].

Um ajuste no tamanho desses *buffers* pode melhorar o desempenho de execução das consultas submetidas ao SGBD de várias maneiras. Por exemplo, aumentando o tamanho do *buffer* de memória utilizado para *cache* de dados, pode-se diminuir o número de leituras de dados efetuadas em disco rígido, que é uma operação muito mais lenta que a leitura de dados efetuada na memória principal do computador [Mul02].

Uma métrica importante para saber se o tamanho do *cache* de dados está adequado é seu *hit rate*. Essa métrica determina o percentual de requisições de acesso a dados que foram atendidas apenas com acesso à memória, não sendo necessário, portanto, acessar o disco rígido. O *miss rate*, por sua vez, determina o percentual de requisições de acesso a dados que foram atendidas acessando o disco rígido. o miss rate e o hit rate possuem uma relação importante definida como $missrate + hitrate = 1$.

Os trabalhos relacionados à gerência automática de memória serão discutidos em detalhes no capítulo 3.

2.3 Arquitetura de SGBDs

Nesta seção é apresentada uma arquitetura genérica de SGBDs. Enfatiza-se, nessa arquitetura, os principais componentes de software que um SGBD utiliza para auxiliá-lo no processamento de transações submetidas por seus clientes.

De maneira geral, os SGBDs existentes utilizam três componentes básicos para processar os comandos SQL que lhe são submetidos. Tais componentes são: um compilador de comandos, um otimizador de consultas e um executor de consultas. A figura 2.4 apresenta uma visão geral da arquitetura genérica de SGBD que será discutida nesta seção.

Quando um comando SQL é submetido a um SGBD, ele precisa passar por um processo de verificação de sintaxe. Esse processo verifica se a escrita do comando está de acordo com a sintaxe da linguagem SQL aceita pelo SGBD. A linguagem SQL (*Structured Query Language*) é a linguagem utilizada pelos SGBDs para expressar consultas às tabelas de seus bancos de dados. Se o comando SQL submetido estiver sintaticamente correto, o compilador, responsável pela verificação sintática, irá gerar uma árvore de consulta *(Query Tree*). Uma *Query Tree* é uma estrutura de dados que armazena informações sobre os objetos do banco de dados que estão envolvidos no comando SQL, incluindo tabelas, seus atributos, operadores

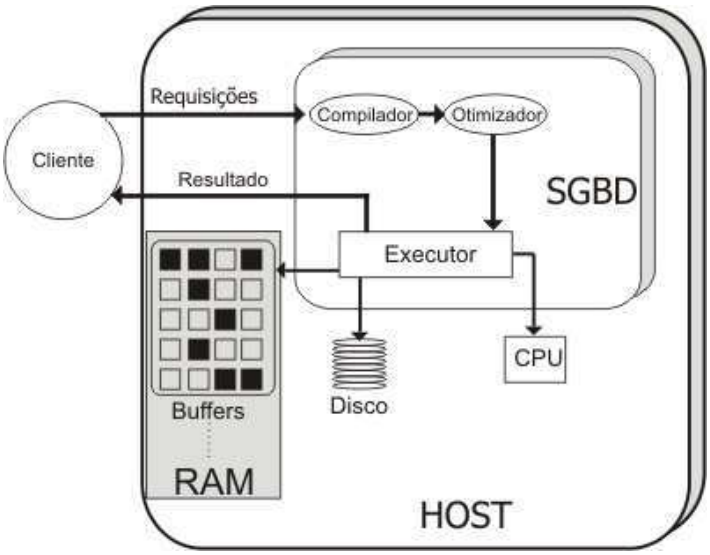

Figura 2.4: *Arquitetura genérica de SGBDs*

lógicos e matemáticos, funções, dentre outros.

A próxima etapa da execução de um comando SQL é a criação do plano de execução (*Query Plan*). O *Query Plan* é criado pelo otimizador de consultas e consiste em uma estrutura de dados que determina uma maneira de recuperar os dados solicitados no comando SQL. Freqüentemente, existem várias alternativas para recuperar os dados solicitados no comando SQL. O otimizador de consultas tem a responsabilidade de escolher uma boa alternativa em um pequeno intervalo de tempo. Dentre as informações contidas em um *Query Plan*, encontram-se decisões tais como: se um determinado conjunto de dados vai ser acessado via índice ou via acesso seqüencial; qual algoritmo de junção será utilizado; se os dados precisarão ser ordenados, dentre outras.

Após a criação do *Query Plan*, este é submetido ao executor de consultas. Esse componente do SGBD é responsável por executar as decisões contidas no *Query Plan*: executar algoritmos de junção, ordenação; buscar dados utilizando índice ou acesso seqüencial, dentre outras. Essa etapa do processamento de comandos SQL, freqüentemente, é a que consome mais recursos do SGBD, tais como memória RAM, disco rígido e CPU. O disco é o local onde estão armazenados os dados procurados; a memória é utilizada, dentre outras coisas, para guardar objetos freqüentemente utilizados pelo SGBD; a CPU efetua todos os algoritmos, tais como ordenação, junções, hashing, dentre outros.

A forma como as arquiteturas dos SGBDs gerenciam esses três recursos de hardware

tem forte impacto no desempenho do SGBD. Geralmente, um banco de dados não cabe inteiramente na memória RAM. O SGBD, sempre que possível, armazena os objetos mais procurados em *buffers* de memória. Como a quantidade de memória RAM quase sempre é bem menor que a quantidade de espaço no disco rígido, o SGBD, eventualmente, precisará trocar um objeto contido na memória RAM por outro armazenado no disco. Essa troca, que necessariamente acessa o disco rígido, é chamada *swap* e deve ser evitada, sempre que possível, pois o tempo de acesso a um dado em disco é muito maior que o tempo de acesso a um dado em memória RAM.

Para reduzir essa atividade de *swap*, os SGBDs implementam diversas técnicas para decidir quais objetos permanecerão na memória e quais serão levados ao disco. De forma geral, os SGBDs usam diferentes *buffers* de memória para diferentes propósitos, por exemplo, um *buffer* para *cache* de disco, outro para *cache* de planos de consultas, outro para ordenação de dados, outro para *cache* de comandos, dentre outros.

2.4 Rede de Filas

Os SGBDs utilizam vários recursos computacionais quando são requisitados pelos seus usuários, incluindo, por exemplo, processadores, discos magnéticos e memória RAM.

Durante a execução de um SGBD, vários clientes, eventualmente, estarão utilizando o sistema de forma simultânea. Nessa ocasião, mais de uma requisição estará solicitando a utilização de um determinado recurso do SGBD ao mesmo tempo, por exemplo, duas transações de banco de dados solicitando acesso a disco. Esses recursos possuem capacidades finitas (um processador, por exemplo, é capaz de realizar apenas X instruções/s) e, eventualmente, ocorrerá a formação de linhas de espera na entrada desses recursos, do mesmo modo que uma fila é formada em um caixa de supermercado. Essas linhas de espera são chamadas filas [MDA04].

Dessa forma, podemos modelar um ambiente de processamento de consultas de SGBD como sendo um conjunto de filas interconectadas, onde, para serem atendidas, as requisições dos clientes passam por várias filas, possivelmente mais de uma vez. Por exemplo, imagine uma transação de banco de dados. Para ser atendida, ela precisará passar por um compilador para verificação de sintaxe; passará por um componente de software responsável por

gerar um plano de execução da transação e então irá para o executor do plano da transação. Durante essa última etapa, a transação irá acessar, possivelmente várias vezes, o disco magnético, em busca de dados, e pelo processador do computador, para manipulação dos dados. Pode-se modelar cada uma das etapas descritas como uma fila onde a transação precisará entrar em uma linha de espera até ser atendida, então ela vai para a próxima fila e assim sucessivamente até que a transação seja concluída. Um modelo de sistema formado por várias filas interconectadas é chamado de rede de filas [MDA04]. Um exemplo de rede de filas para um SGBD genérico é apresentado na seção 2.4.1.

O modelo de fila é caracterizado basicamente por três componentes básicos: a fila (*queue*), onde os clientes esperam para serem atendidos; o recurso (*resource*), onde os clientes são atendidos; e um conjunto de clientes que desejam utilizar o recurso (*customers*). Os clientes chegam ao recurso para serem servidos. Caso o recurso não esteja sendo utilizado, o cliente é atendido imediatamente; caso contrário, o cliente espera um tempo T até ser atendido. Esse tempo de espera T é chamado de *waiting time*. O tempo que o cliente efetivamente gasta sendo atendido em uma passagem pelo recurso é chamado de tempo de serviço (*service time*), ou seja, é o tempo que o cliente leva para ser servido em uma única passagem pelo recurso. A demanda de serviço (*service demand*), por sua vez, é o tempo total gasto pelo cliente sendo servido por um recurso, incluindo nessa soma todas as visitas que o cliente fez ao recurso. A demanda de serviço também pode ser definida como sendo a quantidade de serviço que um cliente demanda de um determinado recurso.

A figura 2.5 apresenta uma representação gráfica do modelo de fila.

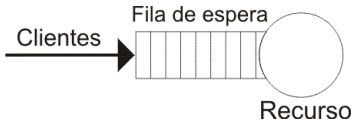

Figura 2.5: *Representação Gráfica do Modelo de Fila*

Os SGBDs, muitas vezes, possuem diferentes classes de clientes, por exemplo, OLTP e OLAP. Essas classes de clientes podem ser modeladas nas redes de filas utilizando-se uma rede de filas multiclasse (*Multiclass Queue Network*). Alguns casos em que se pode utilizar as redes de filas multiclasse são:

- diferentes demandas de serviços - cada classe de clientes apresenta demandas de ser-

viços (*service demand*) significativamente diferentes;

- tipos de cargas diferentes - a exemplo de OLAP e OLTP;
- diferentes objetivos de níveis de serviços - classes de clientes apresentam diferentes níveis aceitáveis de tempo de resposta, por exemplo.

Uma rede de filas ainda pode ser aberta ou fechada. Uma rede de filas é dita aberta quando não se sabe, a priori, o número de clientes que serão atendidos. Nesse caso, diz-se que os clientes chegam à rede de filas a uma determinada taxa de chegada λ (exemplo: 150 transações/segundo). Uma rede de filas é dita fechada quando se sabe, a priori, o número de clientes que serão atendidos. Nesse caso fala-se não em taxa de chegada, mas em população de clientes (exemplo: um processamento *batch* em que se sabe, a priori, quantas transações serão processadas). Em se tratando de uma rede de fila multiclasse, cada classe de clientes pode ter sua própria taxa de chegada (no caso de uma rede de filas aberta) ou população (no caso de uma rede de filas fechada).

Formalmente, uma rede de filas é definida em termos dos seguintes elementos:

K Conjunto de filas que compõem a rede de filas;

R Conjunto de classes de clientes presentes na rede de filas;

$\lambda = \{\lambda_1, ..., \lambda_r\}$ Vetor de taxas de chegadas de uma rede de fila aberta multiclasse, onde $\lambda_r (r = 1, ..., R)$ é a taxa de chegada dos clientes da classe r;

$N = \{N_1, ..., N_R\}$ vetor de população de clientes de uma rede de fila fechada multiclasse, onde $N_r (r = 1, ..., R)$ é o número de clientes da classe r.

$D_{i,r}$ Demanda de serviço dos clientes da classe $r (r = 1, ..., R)$ na fila $i (i = 1, ..., K)$;

$Prior(r)$ Prioridade da classe $r (r = 1, ..., R)$;

$X_{0,r}$ Vazão (*Throughput*) dos clientes da classe r;

$R_{0,r}$ Tempo de Resposta dos clientes da classe r.

Os modelos de fila e de rede de filas têm seus fundamentos no modelo matemático das Cadeias de Markov e são bastante utilizados em análises de desempenho de sistemas [MDA04].

2.4.1 Uma Rede de Filas para SGBDs

Para exemplificar como o modelo de rede de filas pode modelar um sistema, elaborou-se, nesta pesquisa, uma rede de filas genérica para SGBDs, baseada na arquitetura genérica de SGBDs descrita na seção 2.3. O modelo será apresentado de forma incremental para facilitar sua compreensão final.

De maneira simples, um SGBD pode ser visto como um software que alterna seu processamento recuperando dados e manipulando-os com a CPU. Dessa forma, uma rede de filas inicial para representar um SGBD é apresentada na figura 2.6.

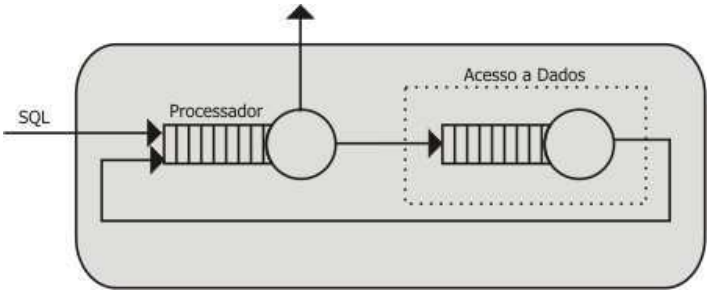

Figura 2.6: *Rede de filas inicial para um SGBD genérico*

As requisições de acesso a dados, no entanto, podem ser atendidas diretamente no disco ou em *cache*, nos *buffers* de memória utilizados pelo SGBD. Dessa forma, a fila de acesso a dados pode ser desmembrada e a nova rede de filas para o SGBD se apresenta como na figura 2.7.

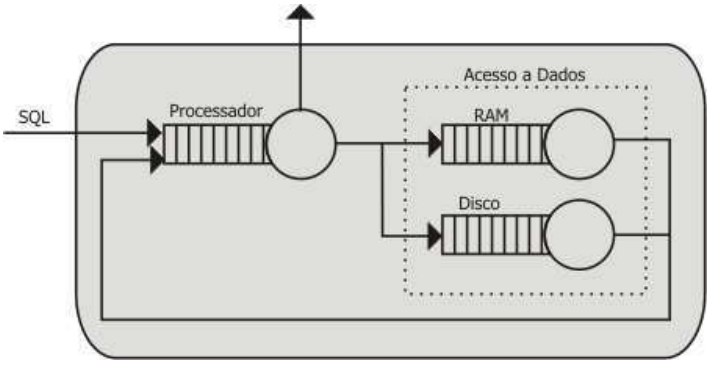

Figura 2.7: *Rede de filas para um SGBD genérico - desmembramento do acesso a dados*

Outra forma de se entender um SGBD é imaginando-o como um software de processamento de transações. As transações chegam ao SGBD e passam por diversas etapas: abertura

de conexão, parsing do SQL, otimização do plano de consulta, execução do plano de consulta e o fechamento da conexão. Essas etapas podem ser modeladas em uma rede de filas semelhante a apresentada na figura 2.8.

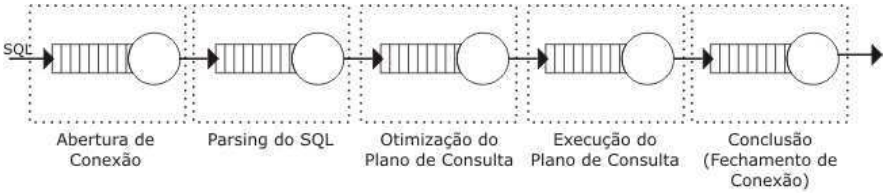

Figura 2.8: *Rede de filas para um SGBD genérico - etapas do processamento de uma transação*

Detalhando as etapas de processamento de uma transação, nota-se que, em cada uma dessas etapas, o SGBD pode usar a CPU, o disco e a memória RAM. Um exemplo de um detalhamento desse processamento é apresentado a seguir.

1. Abertura de Conexão
    1.1 Leitura dos dados do cliente (login, senha, privilégios)
2. Parsing do SQL
    2.1 Acesso ao *Buffer* de Comandos (verificar se a consulta já foi feita anteriormente)
    2.2 Acesso ao Dicionário de Dados (verificar informações de tabelas, atributos, etc)
    2.3 Acesso aos Privilégios de usuários (verificar permissões de acesso aos objetos do comando SQL)
3. Otimização da Árvore de Consulta
    3.1 Acesso ao *Buffer* de Comandos (verificar se já existe um Plano de Execução)
    3.2 Acesso a tabelas de estatísticas
4. Execução da Consulta
    4.1 Acessos a *Buffers* de Dados
    4.2 Acessos a discos (Leitura/Gravação de Dados)

4.3 Espera por recursos (bloqueios)

4.4 Processamento de Dados (ordenações, agregações, junções, etc.)

4.5 Esvaziamento do *Log Buffer* em disco (Gravação de Logs)

4.6 Envio de resultados ao cliente

5. Conclusão

5.1 Fechamento da conexão com usuário.

Em cada passo desse fluxo de execução, o SGBD consome recursos de hardware. Dentre esses recursos, destacam-se:

1. Recursos utilizados da memória

1.1 *Buffer* de Comandos (Passos 2.1 e 3.1)

1.2 *Buffer* de Dicionário de Dados (Passo 2.2)

1.3 *Buffer* de Dados de Usuários (Passos 2.3 e 1.1)

1.4 *Buffer* de Dados (Passo 4.1)

2. Recursos utilizados do disco

2.1 Leitura dos dados do Usuário (Passo 1.1)

2.2 Leitura do Dicionário de Dados (Passo 2.2)

2.3 Leitura dos Privilégios de usuários (Passo 2.3)

2.4 Leitura das Estatísticas das Tabelas envolvidas (Passo 3.2)

2.5 Leitura de Dados (Passo 4.2)

2.6 Gravação em Disco (Passos 4.2 e 4.5)

3. Recursos utilizados da CPU

3.1 Ordenações, agregações e outros Algoritmos (Passo 4.4)

3.2 Processamentos diversos (um pouco em cada passo)

Considerando as etapas do processamento apresentadas e os recursos de hardware consumidos em cada etapa, um SGBD que use a arquitetura genérica da seção 2.3 pode ser modelado através da rede de fila elaborada nesta pesquisa e apresentada na figura 2.9.

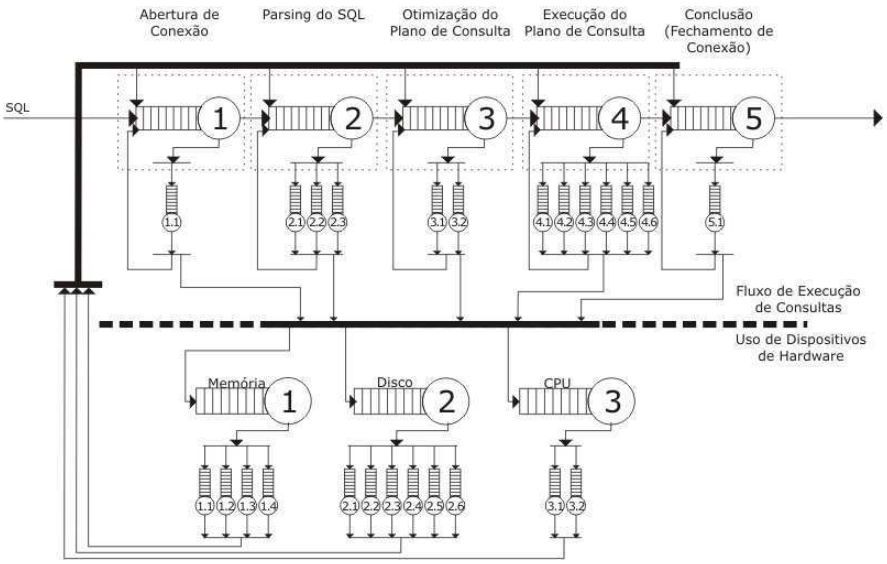

Figura 2.9: *Rede de filas para um SGBD genérico*

As filas da parte superior da figura 2.9 representam o fluxo de execução de consultas (abertura de conexão, compilação do SQL, otimização do plano de consulta, execução do plano de consulta fechamento da conexão), ao passo que as filas da parte inferior representam os recursos de hardware consumidos nesse processo (memória, disco e CPU). Cada fila na figura 2.9 está identificada com sua etapa no processo de execução descrito anteriormente.

2.5 Análise Operacional

A análise operacional é um método quantitativo de análise de desempenho de sistemas baseado no modelo de rede de filas discutido na seção 2.4 [MDA04]. As medidas quantitativas presentes na análise operacional são chamadas de variáveis operacionais. Essas variáveis operacionais são:

T Período de observação do sistema;

K Conjunto de filas do sistema;

B_i Tempo total em que o recurso $i = \{1, .., K\}$ permaneceu ocupado durante o período de observação T;

A_i Total de clientes que chegaram ao recurso $i = \{1, .., K\}$ durante o período de observação T;

A_0 Total de clientes que chegaram ao sistema durante o período T;

C_i Número total de serviços completados (clientes atendidos) no recurso $i = \{1, .., K\}$ durante o período T;

C_0 Número total de serviços completados (clientes atendidos) em todo o Sistema.

Além dessas variáveis operacionais, existem algumas medidas de desempenho derivadas delas, a saber:

S_i Tempo médio de serviço das requisições completadas no recurso i. É definido como

$$S_i = \frac{B_i}{C_i}$$

U_i Taxa de utilização do recurso i. Determina o percentual de tempo que o recurso i permaneceu ocupado durante o intervalo de observação do sistema. É definida como

$$U_i = \frac{B_i}{T}$$

X_i Vazão (requisições/s) do recurso i. Determina a taxa média com que os clientes foram atendidos pelo recurso i durante o intervalo de observação do sistema. É definida como

$$X_i = \frac{C_i}{T}$$

λ_i Taxa de chegada (clientes/s) de clientes no recurso i. Determina a taxa média com que os clientes chegaram ao recurso i durante o intervalo de observação do sistema. É definida como

$$\lambda_i = \frac{A_i}{T}$$

X_0 Vazão do sistema. Determina a taxa média com que os clientes foram atendidos pelo sistema durante o intervalo de observação.

$$X_0 = \frac{C_0}{T}$$

V_i Número médio de visitas (contador de visitas), por requisição, ao recurso i. Determina quantas vezes, em média, uma requisição visita o recurso i durante sua passagem pelo sistema. É definido como

$$V_i = \frac{C_i}{C_0}$$

Tempo de Residência Tempo total gasto por um cliente em um recurso i durante todas as visitas a este recurso. Pode ser definido em função da demanda de serviço como

$$TempoDeResidência = \frac{D_i}{1 - U_i}$$

Tempo de Resposta Tempo total gasto por um cliente dentro do sistema. Pode ser definido como a soma dos tempos de residência desse cliente em cada recurso do sistema.

A análise operacional ainda conta com um conjunto de cinco leis, chamadas Leis Operacionais [MDA04], que definem relações existentes entre as variáveis operacionais apresentadas. Algumas dessas leis são definidas a seguir:

a) Lei da Utilização

Foi definido que, $U_i = \frac{B_i}{T}$. Dividindo-se o numerador e o denominador dessa fração por C_i, obtém-se:

$$U_i = \frac{B_i}{T} = \frac{\frac{B_i}{C_i}}{\frac{T}{C_i}}$$

Como $\frac{B_i}{C_i}$ representa o tempo médio de serviço (Si) das requisições completadas no recurso i, e o termo $\frac{T}{C_i}$ representa o inverso da vazão do recurso i ($\frac{1}{X_i}$), a utilização do recurso i pode ser definida como:

$$U_i = S_i \times X_i$$

Essa relação é chamada **Lei da Utilização**.

Se o número de requisições atendidas (C_i) for igual ao número de requisições que chegaram (A_i), ou seja, $A_i = C_i$, então $X_i = \lambda_i$ e a utilização pode ser definida como:

$$U_i = S_i \times \lambda_i$$

A Lei da Utilização permite calcular a utilização de um recurso a partir de seu tempo médio de serviço S_i e de sua vazão X_i. No caso em que $A_i = C_i$, a utilização do recurso pode ser calculada a partir de seu tempo médio de serviço S_i e de sua taxa de chegada de clientes λ_i.

b) Lei da Demanda de Serviço

A Lei da Demanda de Serviço determina um modo de calcular a demanda de serviço de um recurso a partir de sua utilização e da vazão do sistema. Multiplicando-se a utilização do recurso (U_i) pelo tempo de observação (T), obtém-se o tempo total em que o recurso permaneceu ocupado. Se esse tempo for dividido pelo total de requisições atendidas pelo sistema (C_0), a quantidade média de tempo que o recurso passou ocupado servindo as requisições é derivada. Isso é precisamente a definição de demanda de serviço. Logo, a Lei da demanda de serviço define que:

$$D_i = \frac{U_i \times T}{C_0} = \frac{U_i}{\frac{C_0}{T}} = \frac{U_i}{X_0}$$

ou seja:

$$D_i = \frac{U_i}{X_0}$$

Esta é a **Lei da Demanda de Serviço**.

c) Lei do Fluxo Forçado (Forced Flow Law)

A lei do fluxo forçado oferece uma maneira de descobrir a vazão de um recurso a partir da vazão total do sistema. Por exemplo, imagine que um SGBD possua uma vazão de 3,8 tps e que, em média, cada transação faça dois acessos a disco. Nesse caso, a vazão do disco pode ser inferida e calculada como sendo 7,6 I/O por segundo ($2 \times 3,8$). A equação desta lei é dada por:

$$X_i = V_i \times X_0$$

onde V_i é o número médio de visitas que os clientes fazem ao recurso i, e X_0 é a vazão do sistema.

d) Lei de Little (Little's Law)

A Lei de Little determina uma maneira de se descobrir quantos clientes, em média, estão

sendo servidos em um dado recurso do tipo "caixa preta", ou seja, quantos clientes existem atualmente no interior de um recurso. Um recurso é dito "caixa preta" quando não se sabe detalhes de sua composição interna, por exemplo, uma CPU, a Internet, um SGBD.

Essa informação é derivada de duas outras variáveis, quais sejam: taxa de saída dos clientes do recurso e o tempo médio de permanência dos clientes no recurso. A partir dessas variáveis a Lei de Little afirma que:

O número médio de clientes em um recurso "caixa preta" = (taxa de saída do recurso) X (tempo médio gasto dentro do recurso)

Essa lei, porém, é baseada na hipótese de que nenhum cliente pode ser destruído ou criado dentro do recurso.

As Leis Operacionais permitem determinar os limites de vazão (vazão máxima do sistema) e de tempo de resposta (tempo de resposta mínimo do sistema) para um determinado sistema, baseando-se apenas em suas variáveis operacionais [MDA04].

2.6 Arquitetura do PostgreSQL

Esta seção visa a apresentar uma visão geral da arquitetura do SGBD PostgreSQL. A princípio são apresentados os processos de sistema operacional que são criados quando o PostgreSQL é iniciado e quando um usuário se conecta ao SGBD para submeter-lhe comandos SQL. Em seguida, são descritas, em linhas gerais, as etapas que compõem o processamento dos comandos SQL submetidos ao PostgreSQL. Ao final, são apresentadas as estruturas de memória que auxiliam o SGBD no processo de execução de seus comandos e tecidas algumas considerações sobre o funcionamento e utilidade de cada uma das estruturas, e como a interação entre elas pode afetar o desempenho do PostgreSQL.

2.6.1 Principais Processos

Os principais processos de sistema operacional envolvidos na execução do SGBD PostgreSQL são dois: Postmaster e Postgres [Gro06].

O Postmaster é o principal processo do PostgreSQL. É o processo servidor de banco de dados propriamente dito. É o responsável por gerenciar um único *database cluster*, uma

área de dados do disco rígido que pode conter vários bancos de dados. Para iniciar uma nova conexão, os usuários precisam comunicar-se com esse processo, seja através de uma rede ou localmente via *Unix Sockets*. Ao estabelecer a conexão, o Postmaster cria um processo filho, chamado Postgres, que irá atender às requisições do usuário recém conectado. Para cada cliente conectado ao servidor de banco de dados, existe um processo Postgres que atende a suas requisições.

2.6.2 Processamento de Comandos

O processamento de comandos SQL pelo PostgreSQL é feito em 5 etapas. A primeira delas consiste no estabelecimento de uma conexão entre o cliente e o servidor de banco de dados. Após a conexão estabelecida, o Postmaster determina um processo servidor que irá tratar exclusivamente das requisições do cliente que solicitou a conexão. Como vimos, esse processo é denominado Postgres.

Com a chegada de um comando SQL no processo servidor, dá-se início a uma seqüência de passos que envolve a verificação sintática do comando submetido; a reescrita, se necessária, do comando; a criação de um plano de consulta; e a execução do plano de consulta. Tais passos são denominados, respectivamente, Parser, Rewriter, Planner e Executor [Gro06].

A etapa do Parser consiste em dois passos básicos: 1) verificação da sintaxe do comando e a construção de sua árvore de parser (Parse Tree); 2) Construção da árvore de consulta (Query Tree). No primeiro passo, é verificado a sintaxe do comando. Caso a sintaxe esteja correta, uma árvore de parse (Parse Tree) é criada para o comando. A etapa seguinte consiste em uma interpretação semântica da árvore de parse. Nesta etapa são verificados quais objetos de banco de dados estão envolvidos na consulta submetida. Tais objetos incluem tabelas, índices, operadores, funções. A estrutura de dados que contém tais informações é chamada árvore de consulta (Query Tree) [Gro06].

A etapa que se segue, o Rewriter, recebe a Query Tree da etapa do Parser e a avalia para verificar se alguma alteração em sua estrutura se faz necessária. Essas alterações são baseadas em um sistema de regras do PostgreSQL (PostgreSQL Rule System). Um exemplo de alteração que pode ser feita nessa etapa é a restruturação da Query Tree de uma consulta sobre uma visão em uma Query Tree que envolva as tabelas bases que formam a visão sobre a qual a consulta original foi formulada. Uma análise mais detalhada do sistema de regras

do PostgreSQL está fora do escopo desta dissertação.

A etapa do Planner recebe a árvore de consulta modificada na etapa Rewriter e cria um plano de execução de consulta. Uma mesma árvore de consulta pode ser executada de várias formas. A tarefa do Planner é selecionar a melhor forma de executá-la. O otimizador de plano de consultas implementado pelo PostgreSQL utiliza uma abordagem baseada em algoritmos genéticos para selecionar um plano de execução razoável em um espaço de tempo razoável.

A última etapa do processamento de um comando SQL submetido consiste na execução do plano de consulta gerado pelo Planner. Esta fase, chamada Executor, recebe o plano de consulta do Planner e, recursivamente, a processa para extrair o conjunto de tuplas selecionadas. A execução é baseada essencialmente em um mecanismo de pipeline gatilhado por demanda [Gro06], ou seja, cada nó do plano de execução, quando solicitado, entrega uma tupla de cada vez ou sinaliza que não há mais tuplas para processar. Esta etapa do processamento é a que mais demanda atividades do processador, do disco rígido e da memória do host onde o SGBD está instalado.

2.6.3 Estrutura de Memória do PostgreSQL

A estrutura de memória utilizada pelo PostgreSQL é bastante simples, se comparada com a de outros SGBDs, tais como Oracle e DB2. Ela é formada por quatro áreas de memória principais: Shared Buffer, Work Memory, Temp Buffer e Maintenance Work Memory [Gro06].

O *Shared Buffer Cache* é a área de memória utilizada pelo PostgreSQL para alocar dados de tabelas e de índices [Gro06]. Essa área de memória é compartilhada por todos os clientes que acessam o banco de dados, logo, faz-se necessário o uso de bloqueios para controle de concorrência. O algoritmo de reposição de páginas implementado no PostgreSQL para gerenciar as páginas do Shared Buffer Cache utiliza a política LRU (*Least Recently Used*) [TAN08].

A memória alocada para o Shared Buffer Cache não é paginável [Gro06], ou seja, o sistema operacional não a utiliza para a atividade de swapping. Sendo assim, ela fica reservada exclusivamente para o PostgreSQL e outros processos do S.O não podem utilizá-la. É preciso então ter muito cuidado com o dimensionamento dessa região de memória, pois pode ocor-

rer desperdício de memória para o sistema, caso o Shared Buffer seja superdimensionado [Mom01].

A Work Memory é a memória de trabalho do PostgreSQL. É utilizada para auxiliar a execução de alguns algoritmos durante o processamento de comandos SQL, tais como os algoritmos de ordenação, junção, hashing, dentre outros [Gro06]. Esses algoritmos alocam os dados necessários nesta região de memória até o limite disponível, ocasião em que passam a utilizar a memória secundária do sistema (disco rígido).

Cada cliente conectado ao servidor de banco de dados tem sua própria Work Memory, ou seja, diferente do Shared Buffer Cache, ela não é compartilhada [Gro06]. Outra diferença existente entre a Work Memory e o Shared Buffer Cache é que a primeira, diferente da segunda, não é alocada de forma estática, ou seja, ao conectar ao servidor, o cliente não tem sua quantidade de Work Memory alocada instantaneamente. A Work Memory será alocada de forma dinâmica, a medida em que for sendo requisitada, até o limite configurado no servidor. O Shared Buffer, por sua vez, é alocado estaticamente durante a iniciação do Postmaster e, mesmo que o servidor não receba nenhum comando para processar, essa área de memória não estará disponível para outros processos do sistema operacional [Gro06]. Caso um cliente não esteja utilizando a sua quota de Work Memory, esta poderá ser utilizada pelos processos do sistema operacional.

O dimensionamento da Work Memory também deve ser feito com muita cautela. O parâmetro de configuração do servidor que indica a quantidade máxima de Work Memory que pode ser alocada para cada cliente chama-se work_mem. O valor atribuído a esse parâmetro informa a quantidade máxima de Work Memory disponível para cada cliente, e não a quantidade total (somando as áreas de todos os clientes conectados). Note que, se esse parâmetro for superdimensionado e não levar em conta os outros processos do sistema operacional que coexistem com o servidor de banco de dados, bem como a área de memória já alocada para o Shared Buffer Cache, poderá haver um estouro de memória, caso um cliente solicite mais Work Memory e não haja mais memória disponível no sistema.

As outras áreas de memória utilizadas pelo PostgreSQL visam a agilizar tarefas de manutenção do sistema, tais como VACUUM , ANALYZE, CREATE INDEX, dentre outras (Maintenance Work Memory); e a manipulação de tabelas temporárias (Temp Buffer). Essas duas áreas não serão detalhadas nesta dissertação.

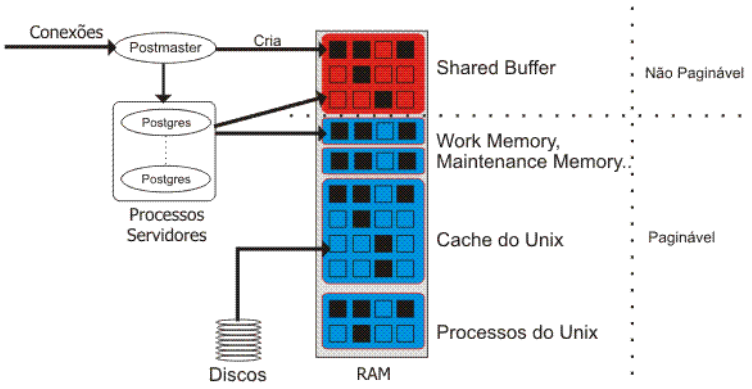

Figura 2.10: *Visão Geral da Arquitetura do PostgreSQL*

A figura 2.10 apresenta uma visão da arquitetura do PostgreSQL em um ambiente Unix, enfatizando seus principais processos e as estruturas de memória utilizadas pelo SGBD.

É importante ressaltar, ainda, a estreita relação que o PostgreSQL mantém com o sistema operacional Unix. Por ter sido, originalmente, projetado e implementado para executar em ambiente Unix, o PostgreSQL ainda possui muitas de suas funcionalidades atreladas a funções e bibliotecas do Unix.

O funcionamento das requisições de I/O necessárias para a execução das consultas do PostgreSQL é estreitamente dependente não só dos *caches* mantidos pelo SGBD, mas também pelo *cache* mantido pelo Unix [Mom01]. Esta relação possui algumas implicações. Ao não achar um dado no Shared Buffer Cache, o PostgreSQL irá solicitar ao Unix, através de uma requisição de I/O, a busca do referido dado em disco. O Unix, por sua vez, de forma transparente para o SGBD, antes de efetuar uma busca pelo dado em disco, verifica se o referido dado está em seu *cache*. Caso o dado se encontre em seu *cache*, ele é automaticamente repassado ao PostgreSQL que o colocará em seu Shared Buffer. Note que, nessa situação, embora tenha ocorrido um *miss* na tentativa de buscar um dado no Shared Buffer, não foi necessária uma leitura física ao disco rígido, pois existe um *cache* intermediário que é o do Unix. Um acesso físico a disco apenas será necessário caso o dado procurado não se encontre nem no Shared Buffer nem no *cache* do Unix.

Vale salientar também que a área de memória alocada ao Shared Buffer é controlada exclusivamente pelo PostgreSQL e não pode ser aumentada ou reduzida dinamicamente (a alocação de memória ocorre uma única vez quando da iniciação do Postmaster). Logo, ela

não está vulnerável a ter seus dados sobrescritos por dados de outros processos do sistema operacional. Por outro lado, o *cache* do Unix, por ser dedicado a todos os processos do sistema operacional, poderá substituir dados que pertençam eventualmente ao banco de dados gerenciado pelo SGBD por dados de outros processos do sistema operacional, tais como páginas web, programas java, vídeos, imagens, dentre outros. A probabilidade de substituição de um dado pertencente ao PostgreSQL por outro que não lhe pertença é bastante reduzida no caso de um servidor de banco de dados dedicado (executando exclusivamente o PostgreSQL).

No entanto, mesmo em um servidor dedicado executando o PostgreSQL, ainda se faz necessário atentar para o fato de que a memória do sistema que está sendo utilizada para *cache* do Unix (a memória livre do host) compete com algumas estruturas de memória utilizada pelos processos servidores do PostgreSQL, por exemplo, quando estes solicitam mais Work Memory. Nessa ocasião, a área de cache do Unix é reduzida (e dados do PostgreSQL são colocados em disco) e a Work Memory é aumentada. De modo inverso, quando a Work Memory deixa de ser utilizada, o Unix poderá utilizá-la para aumentar sua área de *cache*, mantendo mais dados do PostgreSQL em memória e evitando acessos físicos ao disco.

Essas informações são de suma importância para dimensionar cada área de memória utilizada pelo PostgreSQL [Mom01]. Deve-se ter em mente que o *cache* de dados, na prática, é formado por duas estruturas: o Shared Buffer (estrutura estática) e o *cache* do Unix (estrutura dinâmica). Também é preciso saber que a estrutura dinâmica do *cache* de dados compete por memória com outras estruturas, tais como a Work Memory e os processos residentes no host.

Nesse sentido, é preciso muita cautela ao alocar uma grande quantidade de memória ao Shared Buffer Cache, pois assim, restará pouca memória para os outros processos residentes no host e a Work Memory das conexões do SGBD. Por outro lado, se pouca memória for alocada ao Shared Buffer, o cache de dados do PostgreSQL poderá ficar muito reduzido, por exemplo, se muita Work Memory for solicitada pelos clientes do SGBD e sobrescrever o *cache* de dados do Unix. A figura 2.11 apresenta um resumo das situações descritas.

A figura 2.11-A representa o estado da memória principal do sistema logo após o PostgreSQL ser iniciado e antes que qualquer cliente esteja conectado (supondo que o sistema operacional tenha sido recém iniciado e ainda não há formação de *cache* de dados do Unix, ou seu tamanho é desprezível). A figura 2.11-B representa o estado onde a região de

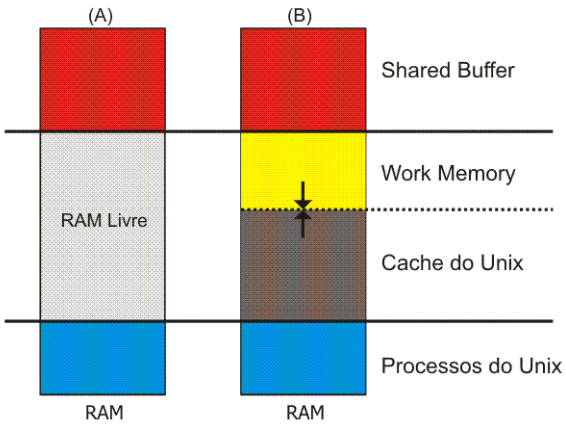

Figura 2.11: *Relação entre as estruturas de memória do PostgreSQL em um ambiente Unix*

memória que estava disponível é gradualmente preenchida pelo *cache* de dados do Unix e pela Work Memory das conexões do PostgreSQL. Ao longo do tempo, com as transações sendo executadas no SGBD, a linha pontilhada tende a oscilar, ora aumentando o cache de dados do Unix, ora reduzindo-o. Ainda há a possibilidade de, caso o parâmetro de configuração da Work Memory seja grande o suficiente, a linha pontilhada descer demais, sobrescrever todo o *cache* de dados do Unix e solicitar mais memória, ocasião em que haverá um estouro de memória. Para evitar esse tipo de situação, deve-se considerar um limite até onde a área da Work Memory poderá sobrescrever o *cache* de dados do Unix.

Capítulo 3

Trabalhos Relacionados

Este capítulo tem como principal objetivo apresentar o estado da arte em gerência automática de memória em SGBDs. São apresentados os principais trabalhos relacionados a esta área, destacando-se as características da abordagem de gerência automática utilizada em cada um deles. Cada trabalho é apresentado em uma seção diferente. A seção 3.8 encerra o capítulo elencando alguns aspectos que os trabalhos relacionados não observaram com a devida importância.

No intuito de automatizar a gerência de memória dos SGBDs, a comunidade de *Self-Tuning Database* vem se esforçando para implementar um componente autônomo que gerencie os buffers de memória de um SGBD de forma automática e com eficiência igual ou superior à de um DBA experiente. Nesse sentido, vários trabalhos vêm contribuindo para a resolução desse problema.

3.1 An analytical model for buffer hit rate prediction

Yongli Xi [XMP01] apresenta um modelo analítico para predição do hit rate do cache de dados de um SGBD. O modelo é baseado em Cadeias de Markov [MDA04] e leva em conta aspectos como o número de partições do banco de dados, tamanho (em páginas) de cada partição existente, probabilidades de acesso às partições, quantidade de memória alocada ao cache de dados e o algoritmo de reposição de páginas utilizado para trocar uma página do cache por uma página em disco. No contexto desse trabalho, uma partição de banco de dados significa um grupo de tabelas que possuem freqüências de acesso semelhantes, por

exemplo, *hot* para uma partição de tabelas muito acessadas, *warm* para uma partição com acessos medianos e *cold* para a partição de tabelas pouco acessadas.

O modelo foi implementado no SGBD IBM DB2 e testado com o benchmark TPC-C [(TP07] (que caracteriza uma carga tipicamente OLTP) e apresentou, segundo os resultados obtidos, boa precisão na estimativa do hit rate de um buffer para uma dada alocação de memória. No entanto, esse trabalho não efetua, de fato, a gerência de memória de um SGBD, apenas serve de apoio para um gerente automático de memória que necessite de uma previsão do *hit rate* de um buffer de memória para tomar suas decisões gerenciais.

3.2 Autonomic Buffer Pool Configuration in PostgreSQL

Powley [WPT05] desenvolveu um componente autônomo para gestão automática dos buffers de memória do SGBD PostgreSQL. A gerência é baseada no ciclo básico da computação autônoma (apresentado no capítulo 2) e visa dimensionar os diversos *buffers* de memória para melhorar o desempenho do SGBD.

O gerente de memória inicia o seu processo de ajuste com o monitoramento dos buffers através da coleta de métricas, tais como número de requisições lógicas de I/O, número de requisições físicas de I/O e o tempo médio de acesso aos dados (Data Access Time − DAT). Para que tais estatísticas fossem coletadas, necessitou-se de uma alteração no coletor de estatísticas do PostgreSQL, uma vez que este apenas coleta estatísticas para um único buffer e não coleta métricas sobre o DAT.

A fase de análise consiste em monitorar o tempo médio de acesso a dados das transações submetidas ao SGBD e, caso este tempo sofra um incremento de 5% em seu valor, o algoritmo de reconfiguração de buffers de memória é executado.

O algoritmo de reconfiguração dos buffers visa reduzir o DAT médio das transações e assume que o DAT é fortemente determinado pelo hit rate de cada buffer. Sendo assim, o algoritmo utiliza um estimador de hit rate, baseado na equação de Belady [Bel66], para prever o DAT médio e assim determinar o ajuste que deverá ser feito nos buffers.

O componente foi acoplado ao PostgreSQL e houve a necessidade de adaptar o código do SGBD para que o gerente automático de memória fosse desenvolvido. Como o PostgreSQL utiliza um único buffer de memória para efetuar cache de disco [Gro06] e a abordagem desen-

volvida no referido trabalho envolve o dimensionamento de múltiplos buffers de memória, necessitou-se de implementar uma mudança na arquitetura de memória do PostgreSQL, de modo que pudessem ser definidos vários buffers para o SGBD. A abordagem implementada assume que os objetos do banco de dados tenham sido previamente alocados aos buffers definidos.

Devido à mudança efetuada na arquitetura do PostgreSQL, a solução apresentada dificilmente poderá ser reusada em outros SGBDs, até mesmo no PostgreSQL em sua versão oficial. Ademais, o critério utilizado para detectar um problema no desempenho do PostgreSQL (incremento de 5% no DAT médio) desconsidera outros recursos utilizados pelo SGBD e que também consomem tempo de processamento, tais como o uso de CPU para execução de algoritmos de ordenação, junção, criação de plano de consulta.

3.3 Fragment Fencing

O Fragment Fencing [BCL93] é uma solução de gerência automática de memória utilizada para atender requisitos de tempo de resposta para diferentes classes de transações submetidas a um SGBD. A abordagem visa traçar um limiar entre a memória utilizada para cache de disco e a memória de trabalho do SGBD (utilizada para hashing, sort, junções). Para alcançar tal objetivo, ela leva em conta a freqüência de acesso aos objetos do banco de dados para agrupar as páginas de dados mais acessadas por cada classe de transação em diferentes fragments. Um fragment, portanto, é definido como sendo um conjunto de páginas de banco de dados que possuem probabilidades de acesso relativamente uniformes.

O algoritmo do Fragment Fencing visa controlar o tempo de resposta de cada classe de transação através do controle individual dos hit rates dos fragments referenciados por essas classes. Ou seja, caso o tempo de resposta de uma determinada classe de transação seja extrapolado, os fragments referenciados pela referida classe de transação são ajustados (acrescentando-se páginas de dados dos fragments ao cache de disco) para que seus hit rates se ajustem de modo a restaurar a meta do tempo de resposta extrapolado.

O número mínimo de páginas de um fragment que deve permanecer em memória para satisfazer o tempo de resposta de uma determinada classe de transação é chamado de Target Residency. Esse número é determinado a partir da freqüência de acesso de cada fragment

(quanto maior a freqüência de acesso, mais favorável é a permanência do fragment em cache) e de uma estimativa da melhora no tempo de resposta que o fragment irá causar, se um maior número de suas páginas de dados ficarem residentes em memória (quanto maior o impacto no tempo de resposta, mais favorável é a permanência do fragment em cache). Um feedback ao algoritmo de troca de páginas dos buffers de memória garante que as páginas pertencentes aos fragments não sejam retiradas de seus buffers.

Para determinar os Target Residencies de cada fragment do banco de dados, o Fragment Fencing se baseia em duas hipóteses: 1) O tempo de resposta das transações é diretamente proporcional ao número de requisições de I/O que elas necessitam; 2) O hit rate de um determinado fragment será igual ao percentual de seus dados que estão residentes em memória. Através dessas duas hipóteses os Target Residencies de cada fragment são determinados em dois passos: 1) Calcula-se a demanda de I/O requerida por uma classe de transação; 2) Configura-se o número de páginas dos fragments que devem permanecer em memória para atender a demanda de I/O calculada no passo anterior. Para cada um desses passos é definida uma função que calcula a informação desejada.

A solução é capaz de alocar mais memória aos fragments quando o tempo de resposta da classe de transações que o referencia está extrapolado e, inversamente, retirar memória dos fragments quando o tempo de resposta da classe de transações que o referencia está muito abaixo da meta.

Nota-se, porém, que o Fragment Fencing está muito acoplado ao SGBD no qual foi implementado (IBM DB2), dificultando o seu reuso em outros SGBDs. Ademais, esta solução se propõe a gerenciar apenas uma parte da memória utilizada pelo SGBD - o cache de dados - deixando de lado outras estruturas importantes, tais como cache de SQL, área de ordenação, dentre outras.

Outro aspecto que deve ser mencionado é o fato de a solução considerar que o tempo de resposta das transações de banco de dados podem ser determinados apenas pelo tempo de I/O, não considerando o tempo gasto com a manipulação dos dados pela CPU, tempo de compilação do comando SQL, geração de seu plano de execução, dentre outras.

3.4 Class Fencing

Brown [BCL96] revisa a abordagem discutida em 3.3. A nova solução proposta, denominada Class Fencing, se baseia nos mesmos componentes básicos do Fragment Fencing: um estimador de tempo de resposta como função de um buffer hit rate; um estimador de buffer hit rate em função de uma alocação de memória para um buffer e um mecanismo de alocação de memória.

O Class Fencing difere da abordagem anterior, principalmente, no tocante ao modelo de estimativa do buffer hit rate. O novo modelo para estimar o hit rate de um buffer baseia-se no Teorema da Concavidade que diz: "*o hit rate, como uma função da alocação de buffer de memória, é uma função côncava, em um ambiente de política de realocação de páginas ótimas*". A abordagem então assume que o algoritmo de realocação de páginas implementado nos SGBDs é próximo ao ótimo. Outra diferença presente no Class Fencing é que este permite o compartilhamento de dados entre diferentes classes, o que não era permitido no Fragment Fencing.

O novo modelo de estimativa do hit rate de um buffer traz algumas melhorias no algoritmo de alocação de memória adotado por esta solução. Uma delas é com relação a convergência do algoritmo de alocação de memória, pois, com o novo estimador de hit rate, o algoritmo alcança a alocação de memória que satisfaz o tempo de resposta de uma classe de transação em menos passos que o Fragment Fencing.

Os passos para a alocação de memória aos buffers são os mesmos do Fragment Fencing: 1) Utilizar o estimador de tempo de resposta para determinar um hit rate que atinja a meta de tempo de resposta definida; 2) Utilizar o estimador de hit rate para determinar uma alocação de memória a um buffer de modo que seu hit rate seja o calculado no passo anterior; 3) Usar o alocador de memória para alocar a quantidade de memória que cada classe de transação necessita para satisfazer seus tempos de respostas.

Nota-se que os mesmos problemas apontados para a solução da seção 3.3 também ocorrem no Class Fencing.

3.5 Quartermaster

O Quartermaster [MPLR02] utiliza a gerência automática de memória para atender a requisitos de QoS em sistemas de comércio eletrônico. O usuário define requisitos de QoS para cada classe de transações do sistema e o Quartermaster é responsável por gerenciar a memória do SGBD e sugerir alterações nos buffers de memória, as quais o DBA pode implementar, a seu critério.

A solução proposta também segue o ciclo básico da computação autônoma. Um monitor coleta informações sobre as classes de transações submetidas ao SGBD. Um analisador verifica se as metas de tempo de resposta estão sendo atendidas por todas as classes de transações. Caso haja alguma violação de tempo de resposta, o analisador sinaliza para o planejador, que elabora uma ou mais estratégias de ajuste para solucionar o problema de desempenho detectado. As estratégias de ajuste consistem em realocação de memória entre os diferentes buffers existentes. O DBA seleciona uma das estratégias proposta que será implementada pelo gerente de memória.

O trabalho foi implementado no SGBD IBM DB2/UDB e utiliza algumas de suas APIs para efetuar o monitoramento do SGBD; que inclui a coleta de métricas, como o número de requisições lógicas de I/O e o número de requisições físicas de I/O, para cada buffer. Também é coletado o tempo de resposta médio de cada classe de transação. O tempo de resposta coletado é comparado com a meta de tempo de resposta de cada classe de transação através de um Achievement Index (AI) que verifica se o tempo de resposta das classes de transações estão sendo extrapolados. Um $AI < 1$ indica que a referida classe de transação não está atendendo a sua meta de tempo de resposta.

Quando o AI de alguma classe de transação indica que seu tempo de resposta está extrapolado, o algoritmo determina uma realocação de páginas de buffers que favorece a classe de transação com o menor valor de AI, ou seja, a classe cujo tempo de resposta está mais distante da meta estabelecida. Este algoritmo é chamado Dynamic Reconfiguration Algorithm (DRF).

O DRF é um algoritmo guloso e iterativo. Ele realoca páginas de um buffer origem para um destino de modo a maximizar o benefício da classe de transação destino. O benefício de uma realocação de páginas para uma classe de transação é estimado em termos do impacto

que a realocação de páginas tem no tempo de resposta da classe de transação destino, ou seja, a que recebe as páginas de memória em seu buffer. O DRF assume que, aumentando o número de páginas de memória de um buffer, o seu hit rate será aumentado e, conseqüentemente, o tempo de resposta da classe de transação que utiliza aquele buffer será reduzido, pois haverá menos acessos físicos a disco.

Para estimar qual será o hit rate resultante do buffer origem e destino, após a realocação de páginas, o DRF utiliza a equação de Belady [Bel66]. A solução assume que o tempo de resposta de uma classe de transação é diretamente proporcional ao tempo médio de acesso a dados utilizados por esta classe. O DRF também utiliza um estimador de tempo de resposta que se baseia nos hit rates dos buffers utilizados pela classe de transação.

Os problemas desta solução são semelhantes aos já apontados para os trabalhos anteriores: forte acoplamento ao SGBD implementado, dificultando o reuso da solução; gerência apenas do cache de dados do SGBD, desprezando outros caches.

3.6 BPCluster

Powley [MPXT06] apresenta uma outra solução de gerência automática de memória, o BPCluster. A abordagem utilizada baseia-se na análise da carga submetida ao SGBD para efetuar um particionamento dos objetos envolvidos e mapeá-los em buffers, levando em conta os seus padrões de acesso e outras similaridades. São utilizadas técnicas de Data Mining para agrupar os objetos similares. Após o mapeamento de objetos similares em buffers, um algoritmo guloso é executado para dimensionar cada buffer de forma a minimizar o custo de uma requisição lógica de I/O.

O processamento do BPCluster é dividido em três etapas: 1) Feature extraction; 2) Clustering; 3) Sizing. A primeira etapa consiste em caracterizar cada objeto do banco de dados em termos de um conjunto de características que são consideradas no mapeamento do objeto em um buffer. Esse conjunto de características é chamado Object's Features Vector, e será utilizado na etapa de Clustering. Dentre as características consideradas estão o tamanho do objeto, tipo (dados / índice), número de acessos ao objeto, número de atualizações (writes) e leituras (reads) do objeto. A etapa de Clustering consiste na execução do algoritmo clássico K-Means para efetuar o mapeamento dos objetos de banco de dados em k buffers. A última

etapa do BPCluster consiste no dimensionamento dos buffers. O algoritmo para dimensionamento dos buffers visa maximizar a vazão do SGBD através da redução do tempo médio de acesso a dados. Para tanto, considera-se o hit rate do buffer e o custo de uma requisição física de I/O. A estimativa do hit rate de cada buffer é feita utilizando a equação de Belady. Para dimensionar os buffers de forma que eles atinjam o hit rate adequado, o algoritmo de dimensionamento usa uma estratégia gulosa baseada no Teorema da Concavidade [BCL96]. O BPCluster foi implementado e testado no SGBD IBM DB2/UDB.

O BPCluster também possui forte acoplamento ao SGBD para o qual foi implementado e ainda apresenta problemas semelhantes aos dos outros trabalhos: assume que o tempo de resposta das transações podem ser determinados apenas pela atividade de I/O das transações, se preocupando apenas com o hit rate do cache de dados do SGBD. Com isso ele não só desconsidera o tempo gasto em outras etapas de execução das transações, como também despreza os outros caches que são utilizados pelo SGBD e que também afetam o desempenho de execução das transações.

3.7 Adaptive Self-Tuning Memory in DB2

Storm [SGAL+06] apresenta uma abordagem bastante complexa e diferente das utilizadas nos trabalhos comentados até agora. O objetivo desse trabalho, implementado no SGBD IBM DB2/UDB, é alocar a quantidade ideal de memória para o SGBD, bem como distribuí-la, também de forma ideal, entre as várias áreas de memória, tais como cache de disco, área de trabalho, cache de SQL, área de lock, dentre outras. A abordagem combina técnicas de teoria de controle, simulação, análise de custo benefício, dentre outras, para atingir seus objetivos.

O componente de gerência de memória, a cada vez que executa uma análise do sistema, avalia a distribuição de memória entre as diversas áreas de memória e verifica se o desempenho do sistema pode ser melhorado com um ajuste nas áreas de memória. Esta avaliação é feita utilizando uma análise de custo benefício e o intervalo de tempo entre uma análise e outra é determinado dinamicamente com base na característica da carga submetida ao SGBD.

A solução proposta simula uma extensão (aumento) dos buffers de memória para calcular o benefício que aquela extensão causaria no desempenho do sistema caso ela fosse de fato

acrescentada ao buffer simulado. O benefício é medido em termos de tempo de processamento, ou seja, é calculado o tempo de processamento que o sistema economizaria caso a extensão simulada de um buffer de memória fosse de fato acrescentada ao referido buffer. Este tempo de processamento pode ser economizado em termos de uma diminuição de acesso a disco (no caso dos caches de disco) ou diminuição de uso do CPU (no caso do SQL cache). Para cada área de memória, é calculada uma unidade de medida que indica a economia de tempo de processamento por unidade de memória. Com esta simulação, é possível detectar quando um buffer não mais se beneficiará com um incremento de memória.

Com as informações de ganho de tempo de processamento de cada área de memória, o objetivo do componente de gerência automática de memória é maximizar a economia no tempo de processamento do sistema, dado que a quantidade total de memória disponível é fixa. Para essa estratégia, é utilizado um algoritmo guloso que retira memória dos buffers que vão ser menos penalizados e aloca para os buffer que vão ser mais beneficiados.

Esta solução, por ter sido implementada para o SGBD IBM DB2/UDB, dificilmente poderá ser reusada em outro SGBD.

3.8 Discussão

A grande maioria das soluções discutidas, embora apresentem bons resultados em seus experimentos, não contemplam, com a devida importância, alguns aspectos relevantes para a gerência automática de memória em SGBDs.

Um primeiro aspecto diz respeito à avaliação do tempo de resposta das transações. Nota-se, claramente, que a maioria dos trabalhos concentra a análise do tempo de resposta de uma transação como função apenas do tempo médio de acesso a dados ou tempo médio de I/O. E, ainda, considera que este DAT médio pode ser determinado como uma função do hit rate do cache de dados.

Esta visão é um pouco restrita, uma vez que o tempo gasto durante o processamento de uma transação de banco de dados não se resume a atividades de I/O que envolvam o disco rígido e o cache de dados. Existem outras etapas do processamento que não são consideradas, a exemplo das etapas de parsing de comando SQL, na qual o comando SQL é compilado, são verificados privilégios de acesso do usuário aos dados envolvidos na consulta; a etapa

do planner, onde é gerado um plano de execução de consulta, são executados algoritmos de otimização de consultas; e, na etapa de execução do comando SQL, não se pode desprezar o tempo gasto com uso de CPU, pois existem transações que fazem uso intensivo do processador para executar algoritmos de ordenação, junção, operações matemáticas e outras funções. Todo esse tempo, desprezado nos trabalhos relacionados, contribui para a composição dos tempos de resposta das transações de bancos de dados.

É importante salientar, também, que, em cada etapa de processamento do comando SQL, podem ser utilizados diferentes tipos de cache, e não só o de dados. Logo, essas outras áreas de memória utilizadas durante o processamento do comando SQL também devem ser levadas em conta na gerência automática de memória, pois com um ajuste nos outros buffers de memória (que não seja o cache de dados) é possível diminuir o tempo gasto com compilação de comandos, acesso a dados de usuários, geração de plano de consulta, otimização de plano de consulta, dentre outros.

É certo que o problema de gerência automática de memória não possui uma solução trivial, que leve em conta todas essas questões levantadas. Porém, uma solução de ajuste automático, mesmo que se proponha a executar ações em um único componente específico do SGBD, por exemplo cache de dados, deve, ao analisar o desempenho do SGBD, avaliar o maior número possível de aspectos que o afetam e verificar se o ajuste sugerido será realmente relevante.

Do ponto de vista de implementação dos trabalhos, nota-se que a grande maioria das soluções possui um forte acoplamento ao SGBD alvo e pouca, ou nenhuma, extensibilidade para outros SGBDs, de forma que dificilmente serão reutilizadas em outros ambientes.

A questão do reuso das soluções propostas é importante pois uma boa abordagem de gerência automática de memória que possa ser reusada, ao menos em parte, em vários outros SGBDs – tais como PostgreSQL, MySQL, dentre outros – seria de grande valia para a comunidade de usuários destes SGBDs.

É evidente que, devido ao fato de cada SGBD existente no mercado possuir sua própria arquitetura e seus próprios componentes, um algoritmo de realocação de buffers de memória utilizado pelo IBM DB2 não irá funcionar no PostgreSQL. Da mesma forma, as informações que alimentam a etapa de análise de uma solução proposta para o IBM DB2 não servirão se a mesma análise for feita no PostgreSQL, pois, certamente, as informações necessárias não

existirão no conjunto de métricas do PostgreSQL.

No entanto, para a etapa de análise, verificou-se ser perfeitamente possível utilizar-se de medidas abstratas que podem ser derivadas de métricas presentes em qualquer SGBD. Um exemplo é a análise de custo benefício utilizada em *Storm* [SGAL+06], onde a unidade de medida poderia ser derivada de outras métricas que não as presentes no IBM DB2.

Dessa forma, é preciso desenvolver soluções com abstrações que permitam o máximo reuso das etapas do ciclo de ajuste automático (Monitorar / Analisar / Planejar e Executar).

Por último, observa-se, nitidamente, que a grande maioria das soluções propostas para gerência automática de memória são voltadas para SGBDs comerciais, deixando de lado a enorme comunidade de usuários e desenvolvedores adeptos a SGBDs livres, tais como PostgreSQL e MySQL.

Levando em conta todos os aspectos discutidos nesta seção, é apresentado, a seguir, um panorama das características dos trabalhos relacionados à gerência automática de memória e alguns requisitos relevantes que esses trabalhos não depositaram a devida importância.

Requisitos

A. **Considerar não só o tempo médio de I/O ou o *hit rate* de *buffers* como fatores determinantes do tempo de resposta das transações em SGBDs**. Existem outras etapas do processamento de transações que também contribuem para o tempo de resposta das transações, tais como compilação, criação do plano de consulta, execução de algoritmos de ordenação, de junção e de funções diversas, tais como funções matemáticas.

B. **Avaliar outros aspectos do SGBD, e não apenas sua memória, para detectar um problema de desempenho**. Durante a fase de avaliação de desempenho do SGBD, é preciso avaliar a situação de outros componentes, tais como o *parser*, o *planner*, a CPU. Afirmar que o SGBD está com um baixo desempenho analisando apenas o *hit rate* de seus *buffers* de memória ou o DAT médio pode levar a uma avaliação imprecisa do desempenho do SGBD.

C. **Extensibilidade**. O gerente automático de memória deve ser extensível para utilização com outros SGBDs. Essa característica é de fundamental importância para que uma boa solução possa ser reusada ou estendida para diferentes SGBDs.

D. **Implementação em SGBDs livres**. A grande maioria dos componentes de gerência de memória são implementados em SGBDs comerciais.

E. **Ajustes em várias estruturas de memória**. O gerente de memória deve ajustar várias estruturas de memória do SGBD, tais como *cache* de dados, área de ordenação e hashing, dentre outras; e não só o *cache* de dados, como faz a maioria dos trabalhos.

Panorama dos trabalhos

Trabalho	**Requisitos Desejados**				
	A	B	C	D	E
[MPXT06]					
[BCL93]					X
[SGAL+06]	X				X
[WPT05]				X	
[BCL96]					
[MPLR02]					

Tabela 3.1: Panorama dos Trabalhos Relacionados

Na tabela 3.1, as células marcadas com um 'X' indicam que o trabalho da linha correspondente se preocupa com o requisito da coluna correspondente. Nota-se que pouquíssimos trabalhos se preocuparam com as questões discutidas nesta seção.

O objetivo deste trabalho de pesquisa é elaborar um framework de gerência automática de desempenho de SGBDs que enfatize os aspectos não contemplados pelos trabalhos discutidos neste capítulo.

Capítulo 4

Arquitetura do Framework DBMS-Analyzer

Este capítulo tem como objetivo principal descrever a arquitetura do *framework* DBMS-Analyzer. A seção 4.1 apresenta uma visão geral do funcionamento e dos principais componentes do framework. A seção 4.2.1 descreve o Gerente de Monitoramento enquanto a seção 4.2.2 detalha o Gerente de Análise. Em seguida, na seção 4.2.3, é descrito o componente responsável por gerar o plano de ajuste - Gerente de Planejamento. Na seção 4.2.4 é apresentado o Gerente de Execução do plano de ajuste e, por fim, a seção 4.3 discute a extensibilidade do *framework*.

Neste capítulo, o leitor não encontrará exemplos práticos de uso do framework DBMS-Analyzer. Um exemplo prático de uso do framework é descrito, em detalhes, no capítulo 5, que descreve a extensão do framework para o SGBD PostgreSQL.

Algumas figuras apresentadas neste capítulo estão no idioma inglês apenas por questões culturais de nomenclatura.

4.1 Visão Geral

O *framework* foi desenvolvido para ser capaz de gerenciar, de forma automática, o desempenho de SGBDs. Suas funções básicas incluem: monitorar o desempenho de SGBDs através da coleta de valores de métricas; alimentar um modelo de rede de filas, a partir dos valores das métricas coletados, através de um mapeamento $métrica \mapsto variáveloperacional$; ana-

lisar o modelo de rede de filas calibrado e verificar se as metas de desempenho do sistema estão sendo satisfeitas; caso as metas de desempenho do sistema não estejam sendo satisfeitas, o DBMS-Analyzer informa quais os componentes do SGBD que estão sobrecarregados (os gargalos) e degradando seu desempenho; por último, o framework deve, automaticamente, gerar um plano de ação para corrigir o problema de desempenho detectado e executar esse plano. Essas funções implementam o ciclo básico de gerenciamento da computação autônoma: monitorar → analisar → planejar → executar, apresentado na seção 2.1.

Todo o processo de gerência é baseado no conceito de **modelo de performance**. Um modelo de performance é o conjunto de informações utilizadas pelo *framework* para gerenciar um SGBD. Essas informações são: rede de filas que modela os componentes do SGBD gerenciado; métricas que serão coletadas do SGBD; definição dos componentes de acesso que irão acessar as métricas durante a coleta; as funções de mapeamento das métricas em variáveis operacionais das filas da rede de filas que representa o SGBD; as metas de desempenho impostas ao SGBD e os componentes de *software* responsáveis por efetuar ajustes nas filas da rede de filas modelada para o SGBD. Cada SGBD gerenciado deve ter um modelo de performance associado.

A rede de filas que modela um SGBD gerenciado deve ser estruturada como uma árvore onde a raiz é o próprio SGBD gerenciado. Cada fila deve representar um recurso ou componente utilizado pelo SGBD e que consuma um tempo considerável de seu processamento (por exemplo, o processador do PC, disco, *parser*, *planner*, *caches*). Se um dado recurso ou componente puder ser dividido em sub-componentes, essa divisão pode ser feita acrescentando-se filas filhas ao componente dividido. Por exemplo, o SGBD PostgreSQL pode ser dividido em três subcomponentes: o *parser*, o *planner* e o *executor*, logo, tem-se a fila PostgreSQL (pai) desmembrada em três filas filhas (*parser*, *planner* e *executor*) formando uma estrutura de árvore. Esse processo pode continuar até satisfazer o nível de análise do usuário. Quanto mais desmembradas forem as filas, mais componentes do SGBD serão analisados pelo *framework*.

Para que a rede de filas definida para um SGBD possa ser utilizada como instrumento de análise pelo *framework*, é preciso que os valores das variáveis operacionais **vazão** e **utilização** de todas as suas filas sejam calculadas. Os valores dessas variáveis operacionais são calculados a partir dos valores das métricas coletados do SGBD ou do ambiente onde ele está

executando. Para calcular o valor das variáveis, o *framework* necessita de duas informações básicas: as métricas a partir das quais a variável operacional será calculada e a função de mapeamento (o cálculo que precisa ser efetuado para se chegar ao valor da variável operacional). Ambas as informações precisam estar definidas no modelo de performance do SGBD gerenciado.

Daqui por diante, por questões de simplificação do discurso, confundiremos métricas com seus valores, assim como variáveis operacionais com seus valores.

Com as variáveis operacionais vazão e utilização das filas calculadas, o *framework* se torna capaz de proceder a uma análise de desempenho do SGBD gerenciado, via a emissão de um relatório de análise. O relatório contém os valores de todas as variáveis operacionais (vazão, utilização, demanda de serviço, tempo de residência e tamanho da fila) de todas as filas, bem como o tempo médio de resposta do SGBD e sua vazão. O tempo de resposta médio é calculado somando-se os tempos de residência das filas folhas da rede de fila elaborada para o SGBD gerenciado. A vazão do SGBD é a vazão da fila raiz da rede de filas.

Calculando-se o tempo médio de resposta do SGBD e sua vazão, e de posse das metas de tempo de resposta e vazão impostas ao SGBD – e definidas em seu modelo de performance – o *framework* é capaz de detectar um problema de desempenho comparando o valor calculado com o valor da meta estabelecida. Se o tempo médio de resposta calculado for superior ao definido como meta para o SGBD ou a vazão calculada for inferior à definida como meta, será indicado um problema de desempenho pelo *framework*. A definição das metas de desempenho é de responsabilidade do usuário do SGBD ou de seu DBA. Nesta dissertação não serão discutidas nenhuma metodologia para definição de metas de desempenho para SGBDs.

Detectado um problema de desempenho, o *framework* irá gerar um plano de ação para que a meta de desempenho extrapolada seja restabelecida. O plano de ação consiste em um conjunto de tuplas do tipo <fila, valor a reduzir da demanda de serviço da respectiva fila>. O plano de ação contém informações de quais filas estão sendo responsáveis por consumir a maior fatia de tempo de processamento do SGBD (gargalos) bem como quanto da demanda de serviço dessas filas precisa ser reduzido para que a meta de tempo de resposta do SGBD seja restabelecida. Esta etapa corresponde à etapa de Planejamento do ciclo básico da computação autônoma.

Criado o plano de ação, este é posto em execução. A execução do plano de ação envolve o último elemento definido no modelo de performance de um SGBD: os componentes de *software* responsáveis por ajustar as filas da rede de filas definida para o SGBD. A execução do plano de ação consiste em, para cada fila contida em suas ações, invocar o ajustador associado a ela. Esta etapa corresponde à etapa de Execução do ciclo básico da computação autônoma.

Cada ajustador de fila desencadeará um processo de ajuste. Um processo de ajuste irá gerar uma seqüência de comandos que deverão ser executados no SGBD. Ao final, o *framework* terá disponível um conjunto de comandos que serão executados no SGBD gerenciado para que suas metas de desempenho sejam restabelecidas, encerrando-se o ciclo de gerência implementado pelo *framework*.

Um resumo do processo de gerência implementado pelo *framework* desenvolvido, bem como as informações utilizadas e geradas durante o processo são ilustrados na figura 4.1.

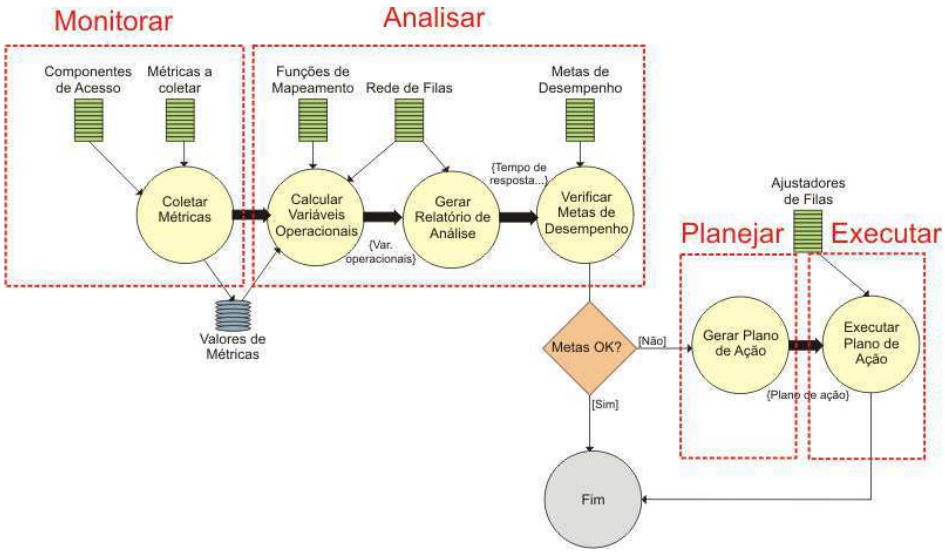

Figura 4.1: *Resumo do processo de gerência usado pelo framework*

Os elementos na cor verde representam as informações contidas em um modelo de performance a ser gerenciado pelo DBMS-Analyzer. Vale salientar que o framework é capaz de gerenciar vários modelos de performance simultaneamente, ou seja, gerenciar vários SGBDs ao mesmo tempo. Para isso, basta que, para cada SGBD a ser gerenciado, defina-se um modelo de performance para ele.

O *framework* armazena em um banco de dados o histórico dos valores coletados das

métricas dos SGBDs gerenciados.

4.2 Arquitetura

A arquitetura do *framework* é composta por quatro componentes básicos: Gerente de Monitoramento; Gerente de Análise; Gerente de Planejamento; Gerente de Execução. Além desses quatro macro componentes, destacam-se ainda dois componentes auxiliares: os componentes de acesso, componentes extensíveis de *software* que auxiliam o Gerente de Monitoramento no processo de coleta de métricas; e os ajustadores de filas, componentes extensíveis de *software* que auxiliam o Gerente de Execução no processo de ajuste das filas da rede de filas dos SGBDs gerenciados. Uma visão geral dessa arquitetura é apresentada na figura 4.2.

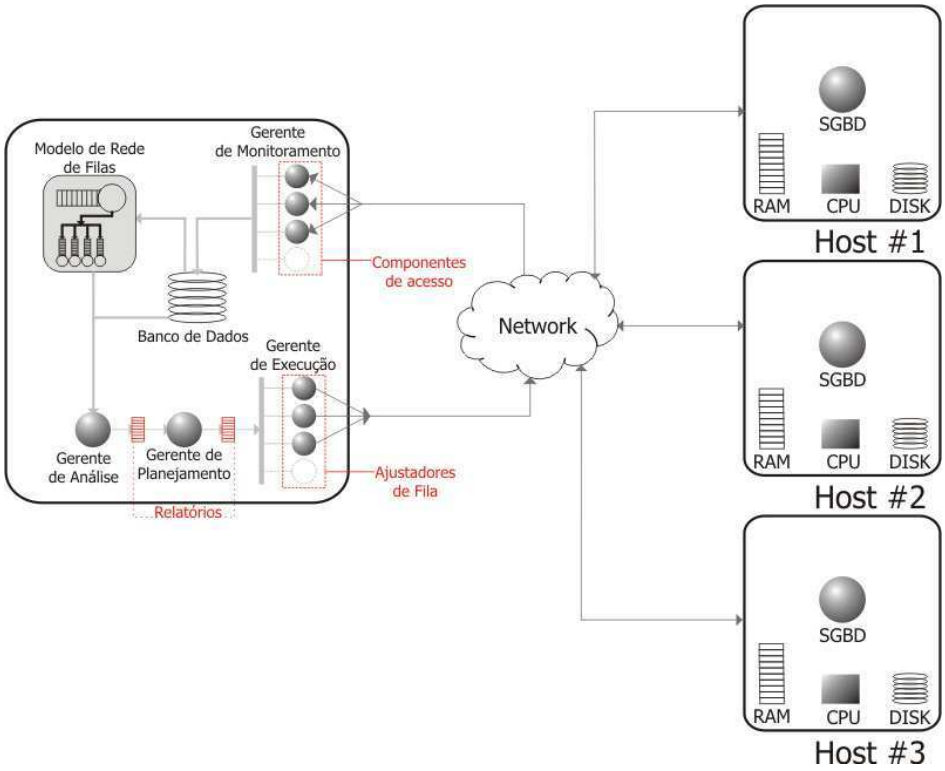

Figura 4.2: *Visão Global da Arquitetura do Framework*

As subseções seguintes descrevem os macro componentes da arquitetura do *framework*.

4.2.1 Gerente de Monitoramento

O gerente de monitoramento é o componente do *framework* que implementa a etapa *Monitorar* do ciclo básico de gerência da computação autônoma. É responsável por gerenciar a coleta de métricas dos SGBDs monitorados utilizando vários coletores de métricas que trabalham em paralelo. Um coletor de métrica coleta as métricas definidas para o modelo de performance de um SGBD. A arquitetura do gerente de monitoramento é apresentada na figura 4.3.

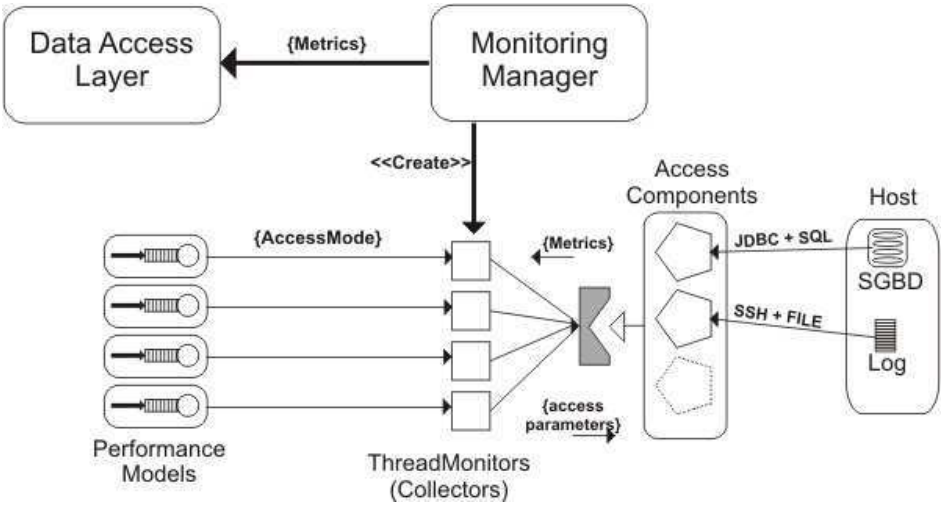

Figura 4.3: *Arquitetura do Gerente de Monitoramento*

Para efetuar a coleta, os coletores de métricas são auxiliados por componentes de acesso. Tais componentes de acesso são as entidades de *software* que implementam os mecanismos pelos quais os coletores acessam os SGBDs monitorados, por exemplo, acessando-os via JDBC, SSH, SNMP. Os componentes de acesso do *framework* podem ser estendidos e reutilizados, o que facilita a monitoração de novos SGBDs.

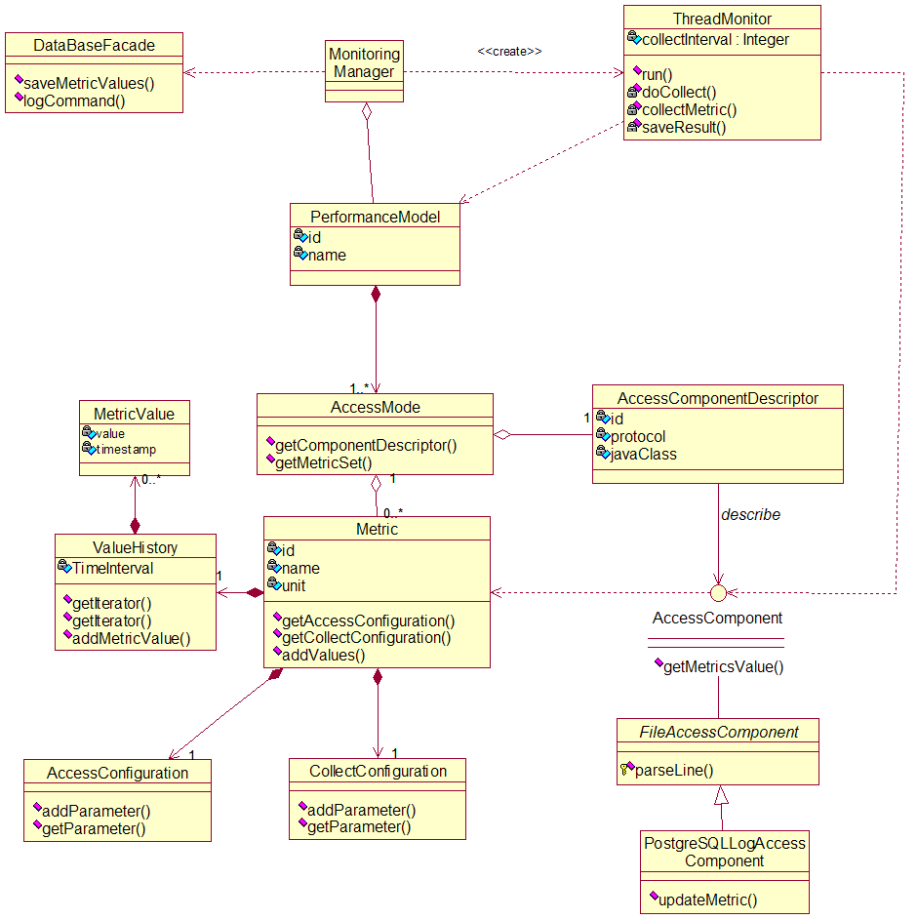

Figura 4.4: *Diagrama de Classe do Gerente de Monitoramento*

A classe MonitoringManager é responsável por gerenciar todo o processo de coleta de todos os modelos de performance definidos no *framework*. Ela é responsável por armazenar esses modelos de performance e gerenciar os coletores de métricas que irão efetuar a coleta das métricas de cada modelo de performance.

A classe PerformanceModel armazena todas as informações relativas a um SGBD monitorado, incluindo a rede de filas que o representa, suas definições de metas de desempenho

do SGBD, as métricas que precisam ser coletadas e os componentes de acessos que serão utilizados na coleta.

Para coletar todas as métricas relacionadas a um SGBD, talvez seja necessário o uso de mais de um componente de acesso. Por exemplo, possivelmente algumas métricas serão extraídas do catálogo de um SGBD, sendo necessário o acesso via JDBC; enquanto outras serão coletadas a partir da análise das estatísticas gravadas em arquivos de log do SGBD, sendo necessário o acesso ao arquivo de log. Para facilitar esse processo, foi criada a classe AccessMode, que associa um conjunto de métricas de um modelo de performance a um componente de acesso comum e que saiba coletá-las.

Cada métrica tem ciência de qual componente de acesso irá coletá-la, sendo assim, elas armazenam os parâmetros de configuração necessários para que o componente de acesso efetue o acesso ao SGBD e a coleta dos valores dessas métricas.

Os coletores de métricas são representados pela classe ThreadMonitor. A classe MonitoringManager cria um ThreadMonitor para cada modelo de performance definido no *framework*. Cada ThreadMonitor efetua a coleta das métricas definidas no modelo de performance associado a ele. Para efetuar essa coleta, os ThreadMonitors recuperam as informações das métricas que precisam ser coletadas, bem como os componentes de acesso que devem ser utilizados para coletá-las. Essas informações estão contidas na classe AccessMode. Os coletores de métricas então usam os parâmetros de configuração definidos na classe Metric para configurar corretamente os componentes de acesso a fim de coletar os valores das métricas com sucesso.

O intervalo entre duas coletas pode ser configurado no framework. O valor padrão do framework é efetuar uma coleta a cada cinco minutos.

O diagrama de classe da figura 4.4 apresenta o projeto do gerente de monitoramento.

A interface AccessComponent, que representa um componente de acesso, possui apenas um método: getMetricsValue(), que recebe um conjunto de métricas e retorna outro conjunto de métricas preenchidas com o conjunto de valores coletados. Esse conjunto é repassado pelos coletores à classe MonitoringManager que se encarrega de enviá-lo para a camada de acesso a dados (representada pela classe DatabaseFacade no diagrama de classe da figura 4.4) para persistência nas tabelas do banco de dados do *framework*. Essa interface representa um ponto de extensão do *framework* que permite a implementação de novos mecanismos de

acesso às métricas dos SGBDs.

O resumo do processo de coleta é apresentado no diagrama de seqüência da figura 4.5.

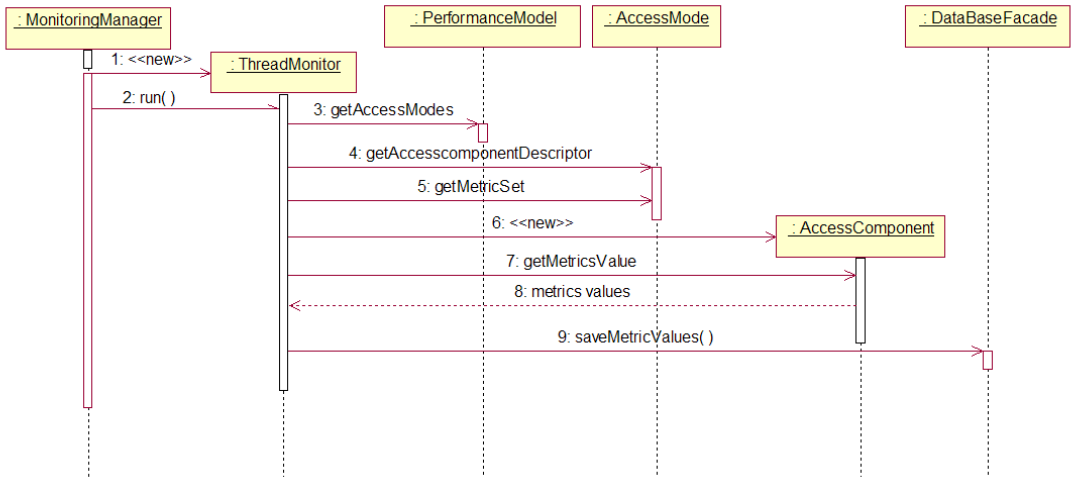

Figura 4.5: *Diagrama de Seqüência do Gerente de Monitoramento*

4.2.2 Gerente de Análise

O Gerente de Análise é o componente do *framework* que implementa a etapa de *Análise* do ciclo básico de gerência da computação autônoma. É o responsável por efetuar o mapeamento das métricas coletadas na etapa de monitoramento nas variáveis operacionais vazão e utilização das filas que compõem o modelo de rede de filas dos SGBDs monitorados. O Gerente de Análise também é o responsável por analisar as redes de filas definidas para os SGBDs monitorados e detectar um eventual problema de desempenho, que, para o *framework*, é caracterizado quando o tempo de resposta calculado para um SGBD extrapola a meta de tempo de resposta definida para ele. Quando um problema de desempenho é detectado, o Gerente de Análise notifica o Gerente de Planejamento, detalhado na próxima seção. O projeto do Gerente de Análise é apresentado no diagrama de classe da figura 4.6.

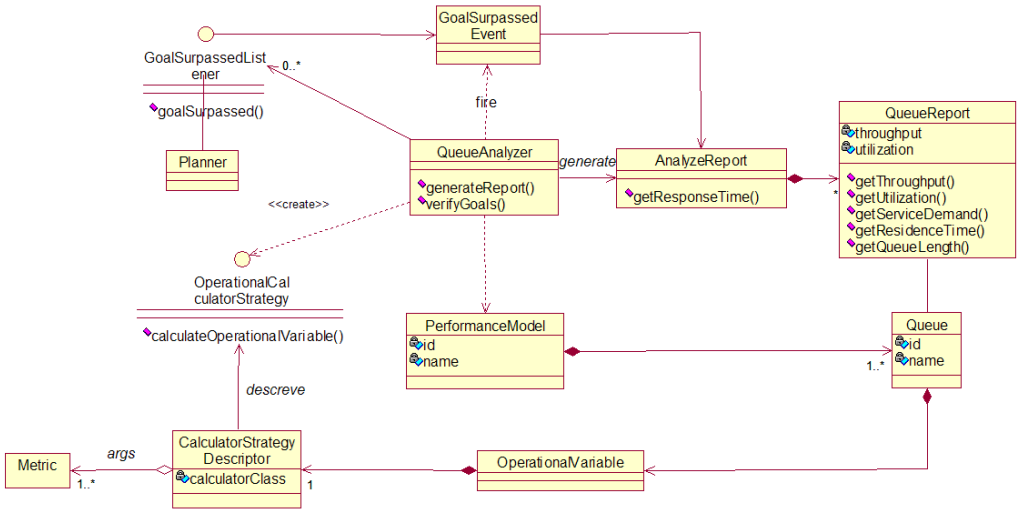

Figura 4.6: *Diagrama de Classe do Gerente de Análise*

A classe principal do projeto do Gerente de Análise é a QueueAnalyzer. Ela é responsável por gerenciar todo o processo de análise. A classe QueueAnalyzer possui dois métodos principais: generateReport() e verifyGoals(). O primeiro tem a função de criar um relatório contendo todas as filas do modelo de rede de filas de um SGBD monitorado, bem como os valores das variáveis operacionais (Vazão, Utilização, Demanda de Serviço, Tempo de Residência e Tamanho da Fila) de cada fila. O segundo método tem a função de verificar se a meta de tempo de resposta definida para o SGBD monitorado está sendo satisfeita.

Para realizar suas funções, a classe QueueAnalyzer precisa de algumas informações básicas recuperadas da classe Queue, que compõe a rede de fila do modelo de performance de um SGBD. Cada Queue contém duas instâncias da classe OperationalVariable. Tais instâncias representam as variáveis operacionais Vazão e Utilização da respectiva fila. Essas duas variáveis operacionais são fundamentais, pois, a partir delas, todas as outras (Demanda de Serviço, Tempo de Residência e Tamanho da Fila) são calculadas.

Uma das etapas da geração do relatório de análise das filas de um modelo de performance envolve o cálculo das variáveis operacionais Vazão e Utilização de cada fila. Os valores dessas variáveis operacionais são calculados a partir dos valores das métricas coletados na etapa de monitoramento.

Para efetuar esses cálculos, a QueueAnalyzer necessita saber quais métricas serão utilizadas no cálculo de cada variável operacional de cada fila do modelo de performance. Além disso, também é preciso saber como as métricas envolvidas serão manipuladas para calcular o valor da variável operacional da fila. Estas informações, métricas envolvidas no cálculo de uma variável operacional de uma fila e a manipulação necessária para efetuar o cálculo, precisam ser definidas pelo usuário. Tais definições são armazenadas na classe CalculatorStrategyDescriptor, que informa o conjunto de métricas necessárias para calcular o valor de uma variável operacional bem como a classe Java responsável por efetuar a manipulação desses valores a fim de calcular o valor da variável operacional em questão.

A classe Java informada pelo CalculatorStrategyDescriptor deve ser uma implementação da interface OperationalCalculatorStrategy, cujo único método é calculateOperationalVariable(). Esse método recebe um conjunto de métricas e uma instância da classe CalculatorStrategyDescriptor e retorna o valor da variável operacional para a qual ele foi criado.

A interface OperationalCalculatorStrategy juntamente com a classe CalculatorStrategyDescriptor formam outro ponto de extensão do *framework*. Através dessa classe e dessa interface, o usuário poderá definir qualquer mapeamento entre um conjunto de métricas em uma variável operacional, permitindo que o *framework* possa analisar qualquer SGBD com a Análise Operacional.

Calculados os valores das variáveis operacionais Vazão e Utilização de cada fila da rede de filas de um modelo de performance, a classe QueueAnalyzer irá gerar, para cada fila, um

relatório de fila, representado no diagrama de classe da figura 4.6 pela classe QueueReport. A classe QueueReport contém informações sobre os valores das seguintes variáveis operacionais de uma dada fila: Vazão, Utilização, Demanda de Serviço, Tempo de Residência e Tamanho da Fila.

Calculados os relatórios das filas, é então gerado o relatório da análise, representado pela classe AnalyzeReport. Essa classe contém os relatórios das filas e o valor do tempo de resposta médio dos comandos SQL submetidos ao SGBD monitorado. O tempo de resposta é calculado somando-se os valores dos Tempos de Residência das filas de mais baixo nível definidas no modelo de rede de filas elaborado para o SGBD monitorado.

É preciso lembrar que o modelo de rede de filas definido para os SGBDs monitorados deve formar uma árvore cuja raiz representa o SGBD monitorado e as filas de um nível inferior da árvore (filhas) representam a decomposição de uma fila de nível superior (o pai). Sendo assim, o tempo de resposta é a soma dos tempos de residência das filas folhas da árvore.

Outra função do Gerente de Análise é verificar se as metas de desempenho dos SGBDs monitorados estão sendo satisfeitas. Para tanto, o QueueAnalyzer apenas efetua uma comparação entre o valor do tempo de resposta calculado no relatório de análise com o valor definido pelo usuário como meta de tempo de resposta para o SGBD monitorado.

Caso o tempo de resposta calculado no relatório de análise seja superior ao tempo de resposta definido como meta de desempenho pelo usuário, o QueueAnalyzer irá gerar um evento do tipo GoalSurpassedEvent contendo o relatório de análise recém criado. Um dos interessados (GoalSurpassedListener) nesse evento será o Gerente de Planejamento, descrito a seguir.

4.2.3 Gerente de Planejamento

Detectado um problema de desempenho, o gerente de planejamento é acionado. Este componente implementa a etapa de Planejamento do ciclo básico de gerência da computação autônoma. Sua função é bem simples: informar ao Gerente de Execução quais filas do SGBD monitorado estão mais sobrecarregadas (gargalos) e determinar quanto da Demanda de Serviço de cada uma delas deve ser reduzido para que a meta de desempenho extrapolada seja restabelecida.

Essas informações são colocadas em um plano de ação. O plano de ação consiste basicamente de um conjunto de pares <Fila, Valor da Demanda de Serviço a reduzir>. As decisões sobre as intervenções no SGBD necessárias para reduzir a demanda de serviço das filas contidas no plano de ação são de responsabilidade do Gerente de Execução, através dos ajustadores de filas, discutidas na próxima seção. A figura 4.7 apresenta o diagrama de classe do Gerente de Planejamento.

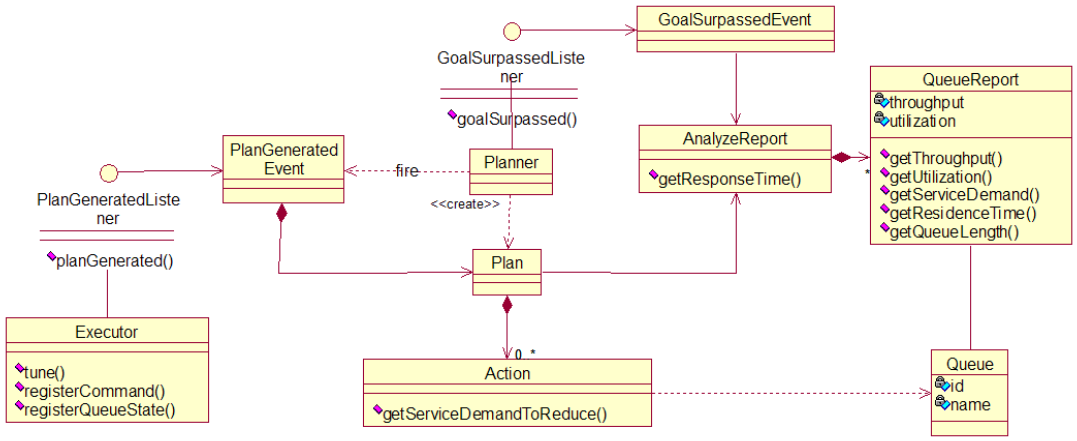

Figura 4.7: *Diagrama de Classe do Gerente de Planejamento*

Todo o processo de geração do plano de ação inicia quando o Gerente de Planejamento, representado pela classe Planner do diagrama de classe da figura 4.7, recebe o evento GoalSurpassedEvent do Gerente de Análise. Esse evento contém o relatório de análise (AnalyzeReport) gerado pelo gerente de análise.

Com base no relatório de análise, que contém informações de desempenho sobre as filas de um modelo de rede de filas, o Planner irá verificar quais filas estão mais sobrecarregadas e determinar quanto do Tempo de Residência dessas filas precisa ser reduzido para que a meta de desempenho do SGBD monitorado seja restabelecida.

Para verificar quais filas estão mais sobrecarregadas, o Planner considera apenas o valor do Tempo de Residência de cada fila. Aquela que possuir maior Tempo de Residência será a fila mais sobrecarregada, e será nela que o *framework* deverá atuar para restabelecer o desempenho do SGBD monitorado. A figura 4.8 apresenta o algoritmo utilizado pelo gerente de planejamento para geração do plano de ação. Esse algoritmo utiliza o algoritmo da figura 4.9 para determinar quanto do Tempo de Residência de cada fila deve ser reduzido para que

a meta de tempo de resposta do SGBD seja restabelecida. Como o Tempo de Residência é derivado da demanda de serviço de uma fila, as reduções também podem ser calculadas em termos de demanda de serviço.

TRCalculado: Tempo de Resposta Calculado no Relatório de Análise

metaTR: Meta de Tempo de Resposta Imposta ao SGBD

redeDeFila: Rede de Filas definida para o SGBD

planoDeAcao: Plano de Ação gerado

```
/* Tempo a ser reduzido para restabelecimento da Meta        */
totalAReduzir ← TRCalculado − metaTR;
while totalAReduzir > 0 do
    g1 ← recuperaGargalo(redeDeFila);
    g2 ← recuperaSegundoGargalo(redeDeFila);
    /* calcula a parcela de tempo que deve ser reduzida do atual
       gargalo (g1)                                              */
    tempoAReduzirDeG1 ← calcReducaoDoGargalo(g1, g2, totalAReduzir);
    proximaAcao ← criaAcao(g1, tempoAReduzirDeG1);
    adicionaAcao(planoDeAcao, proximaAcao);
    /* atualiza o tempo que ainda deve ser reduzido para
       restabelecer a meta                                       */
    totalAReduzir ← totalAReduzir − tempoAReduzirDeG1;
    /* atualiza a situação da rede de filas (pode haver uma
       mudança de gargalo)                                       */
    g1.tempoDeResidencia ← g1.tempoDeResidencia − tempoAReduzirDeG1;
end
/* Resume os ajustes que devem ser feitos nas filas          */
resumirPlanoDeAcao(planoDeAcao);
return planoDeAcao;
```

Figura 4.8: *Algoritmo para Geração do Plano de Ação*

g1: Primeiro Gargalo - Fila Folha com maior tempo de residência da rede de filas

g2: Segundo Gargalo - Fila Folha com segundo maior tempo de residência da rede de filas

totalAReduzir: Tempo restante que precisa ser reduzido para restabelecer a meta do SGBD

reducaoDoGargalo: Tempo de Residência a ser reduzido do atual gargalo (g1)

```
/* A redução máxima de uma fila será de 70% de seu tempo de
   residência (não se pode ZERAR o tempo de residência de uma
   fila)                                                    */
```

1 maxAReduzirDeG1 ← $70\%\times$ g1.tempoDeResidencia;

```
/* calcula o tempo que pode se reduzido de G1 antes que G2 se
   torne o principal gargalo                                */
```

2 gapEntreGargalos ← g1.tempoDeResidencia − g2.tempoDeResidencia;

```
/* usa-se a função mínimo para evitar:                      */
/* 1.  Reduzir mais que 70% do tempo de residencia de uma fila;
   */
/* 2.  Reduzir mais tempo de residência de G1 quando G2 já se
   tornou o principal gargalo;                              */
```

3 reducaoDoGargalo ← `minimo(`gapEntreGargalos, totalAReduzir, maxAReduzirDeG1`);`

```
/* Para não reduzir exatamente a diferença de tempo entre os
   gargalos G1 e G2, acrescenta-se 10% à redução calculada.
   Dessa forma, G2 se tornará o principal gargalo na próxima
   iteração do algoritmo de Geração do Plano de Ação        */
```

4 **if** reducaoDoGargalo = gapEntreGargalos **then**

5 reducaoDoGargalo ← $1.1\times$ reducaoDoGargalo;

6 **end**

7 **return** reducaoDoGargalo;

Figura 4.9: *Algoritmo do Cálculo do Tempo a Reduzir do Gargalo*

O algoritmo de geração do plano de ação (figura 4.8) funciona basicamente da seguinte forma: A linha 1 determina o tempo total que deverá ser reduzido dos tempos de residência das filas para que a meta de tempo de resposta seja restabelecida. O laço das linhas 2 a 10, a cada iteração, insere uma Action no plano de ação. Uma Action contém o tempo de residência que deve ser reduzido de uma determinada fila visando ao restabelecimento da meta de tempo de resposta.

Para criar uma Action o algoritmo necessita das seguintes informações: primeiro gargalo do sistema (fila com maior tempo de residência), segundo gargalo (fila com segundo maior tempo de residência) e o tempo restante que ainda precisa ser reduzido para restabelecer a meta de tempo de resposta. Na linha 5 do algoritmo da figura 4.8 é calculado o tempo de residência que deve ser reduzido do primeiro gargalo. Em seguida, nas linhas 6-9, é criada uma Action contendo a redução recém calculada; atualizado o plano de ação com a nova Action criada; ajustado o tempo total que ainda resta ser reduzido das filas; ajustado o tempo de residência do gargalo do sistema. O ajuste do tempo de residência do gargalo do sistema, a cada iteração, é importante pois poderá haver, na próxima iteração, uma mudança de gargalo, ou seja, outra fila poderá ter o maior tempo de residência.

O algoritmo da figura 4.9 calcula o tempo de residência que deve ser reduzido do gargalo do sistema a cada iteração. O algoritmo recebe os dois maiores gargalos do sistema e o tempo que ainda precisa ser reduzido das filas para restabelecer a meta de tempo de resposta do SGBD. Na linha 1, é calculada a redução máxima que poderá ser feita no tempo de residência do atual gargalo. Essa redução foi limitada a 70% do tempo de residência do gargalo, pois não se pode reduzir em 100% o tempo de residência de uma fila. O valor de 70% foi determinado apenas para impor um limite ao algoritmo, não possuindo nenhum embasamento teórico ou heurístico em sua determinação.

Na linha 2, é calculado o gap existente entre o primeiro e o segundo gargalo, ou seja, a diferença entre seus tempos de residência. A redução do gargalo é feita recuperando-se o valor mínimo entre o gap entre o primeiro e segundo gargalo, o valor máximo a reduzir do primeiro gargalo e o tempo restante a reduzir para restabelecer a meta de tempo de resposta. A aplicação da função mínimo a esses valores evita reduções indesejadas, tais como: reduzir mais que 70% do tempo de residência de uma fila, reduzir tempo de residência de uma fila quando outra fila já passou a ser o gargalo do sistema, reduzir um valor maior que o valor do

tempo de residência de uma fila (deixando-a, em tese, com tempo de residência negativo).

A estrutura condicional das linhas 4-6 do algoritmo da figura 4.9 permite que ocorra uma mudança de gargalo do sistema caso o valor a ser reduzido do primeiro gargalo seja o valor do gap existente entre o tempo de residência do primeiro gargalo e do segundo gargalo. Dessa forma, na próxima iteração do algoritmo da figura 4.8, o primeiro gargalo será o segundo gargalo da atual iteração. Essa alternância poderá ocorrer até que a meta de tempo de resposta seja restabelecida. Note que poderão ser criadas várias Actions para uma mesma fila.

Após criadas todas as Actions, o plano de ação é resumido, agregando, em uma única Action, as Actions que reduzem o tempo de residência de uma mesma fila. Isso é feito na linha 11 do algoritmo da figura 4.8.

Determinadas as filas que estão com os maiores tempos de residência e calculados os valores a serem reduzidos das respectivas filas, o Planner irá criar o plano de ação que será repassado ao Gerente de Execução. Esse plano é representado pela classe Plan e consiste de um conjunto de Actions. Uma Action é um par <Fila, valor a reduzir da demanda de serviço da respectiva fila>.

Criado o plano de ação, o Gerente de Planejamento dispara o evento PlanGeneratedEvent e o repassa ao Gerente de Execução.

É importante salientar que o plano de ação gerado pelo Gerente de Planejamento é formado por ações abstratas que apontam ajustes que devem ser feitos em variáveis operacionais de filas. As ações concretas, ou seja, os ajustes que serão efetivamente implementados na configuração dos SGBDs monitorados, serão determinados pelo gerente de execução, como veremos na próxima seção.

4.2.4 Gerente de Execução

O gerente de execução implementa a etapa de Execução do ciclo básico de gerência da computação autônoma. É o componente responsável por determinar os ajustes que devem ser feitos na configuração do SGBD monitorado para que seu desempenho seja otimizado e a meta de desempenho definida pelo usuário, e que foi extrapolada, seja restabelecida.

Para efetuar tal tarefa, o Gerente de Execução utiliza o conceito de **processo de ajuste**. Um processo de ajuste é definido para cada fila da qual se deseje reduzir a demanda de

serviço. Um processo de ajuste é formado por várias etapas e seu produto final é uma lista de comandos que devem ser executados no SGBD monitorado. Em cada uma dessas etapas, pode ser analisado algum aspecto que afete o desempenho da fila que está sendo otimizada. Por exemplo, em um processo de ajuste de I/O de um SGBD pode-se ter uma etapa para avaliar o *hit rate* do *cache* de dados, outra para avaliar o impacto de uma eventual criação de índice, dentre outras. Com isso, é possível melhorar o processo de ajuste acrescentando-lhe outras etapas para que o ajuste final efetuado leve em conta vários aspectos do SGBD monitorado.

O projeto do Gerente de Execução é apresentado na figura 4.10.

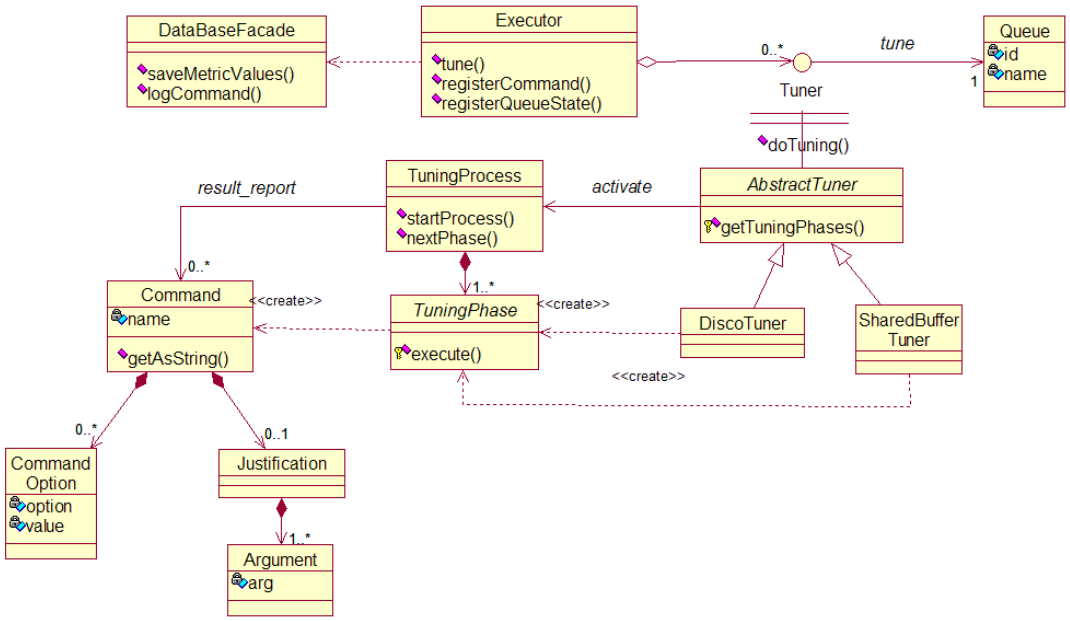

Figura 4.10: *Diagrama de Classe do Gerente de Execução*

A classe Executor é a responsável por gerenciar todo o processo. Ao receber o plano de ação, contendo as filas mais sobrecarregadas do SGBD monitorado, o Executor irá, para cada fila contida no plano de ação: 1. recuperar o ajustador (Tuner) associado à respectiva fila; 2. desencadear o processo de ajuste (TuningProcess) associado; 3. atualizar uma lista de comandos que serão executados ao final de todo o processo.

A interface Tuner, que representa um ajustador de uma fila, contém apenas o método doTuning(). A classe AbstractTuner implementa o código comum a todos os Tuners. Ela gerencia o processo de ajuste de um Tuner. A classe AbstractTuner, juntamente com a classe

TuningPhase, representam um ponto de extensão do *framework*. É através dessas classes que um processo de ajuste pode ser criado ou alterado e novos ajustadores podem ser definidos para novas filas de novos SGBDs que venham a ser monitorados. De outra forma, ajustadores já existentes podem ser reusados se o modelo de performance definido para um SGBD monitorado também for reusado. Por exemplo, ao definir um modelo de performance para o SGBD PostgreSQL, esse modelo pode ser reusado para todas as instâncias do SGBD PostgreSQL e assim, todos os ajustadores definidos podem ser reusados nessas instâncias.

O processo de ajuste é representado pela classe TuningProcess. Essa classe é composta por várias instâncias da classe TuningPhase, que representa uma etapa do processo de ajuste.

O produto de um processo de ajuste é uma lista de comandos. Em cada fase do processo de ajuste (TuningPhase) essa lista é atualizada, até que, ao final do processo, a lista contém os comandos necessários para ajustar uma fila do SGBD monitorado. Os comandos que serão executados são representados pela classe Command e possuem um conjunto de argumentos (CommandOption) e uma justificativa para a sua execução, ou seja, a razão que levou o *framework* a sugerir sua execução.

Quando todos os Tuners encerrarem seus processos, o Executor terá a lista completa de comandos que devem ser executados para restabelecer a meta de desempenho do SGBD monitorado. Cada comando da lista é associado a um componente que sabe como executá-lo. Esses componentes podem ser implementados para executar comandos SQL, shell-scripts, comandos SNMPs, dentre outros.

O plano de execução encerra-se quando todos os comandos recebidos pelo Executor são executados no SGBD monitorado.

4.3 Extensibilidade do Framework

O *framework* desenvolvido pode ser estendido em três pontos básicos. O primeiro ponto diz respeito ao modo como ele irá coletar as métricas do SGBD. Viu-se que, para efetuar a coleta de métricas, o gerente de monitoramento utiliza um componente de acesso, que é o responsável por efetuar o acesso até o local onde as métricas se encontram. Esse acesso pode ser feito utilizando vários protocolos, a saber, JDBC, SNMP, SSH, HTTP, dentre outros. Neste sentido, vários componentes de acesso podem ser implementados e, uma vez

implementados, podem ser reusados na gerência de diferentes SGBDs.

O segundo ponto de extensão do *framework*, reside no modo como é feito o mapeamento das métricas coletadas nas variáveis operacionais vazão e utilização das filas da rede de filas criada para um SGBD. Atualmente, o cálculo de cada uma dessas variáveis é feito por uma classe Java, ou seja, tem-se uma classe Java para efetuar o cálculo de cada variável operacional, vazão e utilização, de cada fila da rede de filas do SGBD gerenciado. Com isso, para gerenciar um novo SGBD, é preciso definir as classes Java que irão efetuar o mapeamento das métricas coletadas nas variáveis operacionais vazão e utilização das filas da rede de filas criada para o referido SGBD. Desse modo, várias estratégias de mapeamento podem ser definidas no *framework*.

O último ponto básico de extensão do *framework* desenvolvido é em relação aos processos de ajustes que podem ser definidos para as filas de uma rede de filas criada para um SGBD. Para cada nova rede de filas criada poderão ser definidos novos processos de ajustes para suas filas. De outro modo, um processo de ajuste já existente para determinada fila poderá ser modificado com acréscimos de novas etapas que poderão analisar outros aspectos para ajuste de uma fila. Por exemplo, um processo de ajuste de uma fila de I/O que analise apenas o tamanho da memória do SGBD, poderá ser acrescido de uma etapa que avalie a possibilidade de se criar um índice em alguma tabela para reduzir I/O.

Além desses três pontos básicos de extensão, o comportamento do *framework* pode ser modificado com alterações em suas configurações. Como foi dito, todas as configurações necessárias para a gerência de um SGBD pelo *framework* precisam ser definidas em um **modelo de performance**. Logo, para adicionar um novo SGBD à gerência do *framework*, o que se precisa fazer é definir um novo modelo de performance. Para isso, será preciso:

1. **Definir a rede de filas que irá modelar o novo SGBD**. Cada fila da rede de filas deve modelar um componente de *software* ou recurso utilizado pelo SGBD e que consuma algum tempo de processamento, tais como *parser*, *planner*, CPU, disco, dentre outros. A rede de filas não possui uma quantidade máxima de filas que podem ser adicionadas à rede, essa quantidade irá depender do nível de detalhes que o usuário quiser. Quanto maior o número de filas, mais componentes do SGBD serão analisados pelo *framework*. Vale lembrar também que a rede de fila deve formar uma árvore com a fila raiz modelando o próprio SGBD monitorado. A fila raiz então pode ser desmembrada

em várias filas que representem os componentes de mais alto nível do SGBD. Esses componentes ainda podem ser desmembrados em filas filhas e assim por diante.

2. **Definir o conjunto de métricas que serão coletadas do ambiente de execução do SGBD**. As métricas devem ser selecionadas de modo que permitam o cálculo das variáveis operacionais vazão e utilização de cada fila da rede de filas definida no passo anterior.

3. **Definir os componentes de acesso que irão auxiliar na coleta das métricas definidas no passo anterior**. Caso já exista um componente de acesso capaz de acessar as métricas definidas no passo anterior, basta que o usuário informe o identificador desse componente de acesso. De posse desse identificador, o *framework* é capaz de recuperar a classe Java que implementa o protocolo de acesso às métricas. Caso não exista nenhum componente de acesso capaz de acessar as métricas definidas no passo anterior, o *framework* terá de ser estendido com a implementação de um novo componente de acesso. Esse novo componente de acesso deverá ser registrado nas configurações do *framework* e seu identificador informado para que o *framework* possa recuperar a nova classe Java implementada.

4. **Definir as funções de mapeamento das métricas coletadas nas variáveis operacionais vazão e utilização de cada fila da rede de filas definida para o novo SGBD gerenciado**. Para definir uma função de mapeamento, é preciso implementar uma nova classe Java, que estenda a interface OperationalCalculatorStrategy do Gerente de Análise, e que efetue o cálculo da variável operacional. Cada fila da rede de filas terá duas funções de mapeamento, uma para calcular a vazão da fila e outra para calcular sua utilização. Após implementadas, as funções de mapeamento precisam ser informadas no novo modelo de performance, bem como o conjunto de métricas que participam do cálculo da função.

5. **Definir as metas de desempenho para o novo SGBD gerenciado**. As metas de desempenho podem ser de tempo de resposta ou vazão do SGBD e serão utilizadas pelo Gerente de Análise para verificar se o desempenho do novo SGBD está satisfatório ou necessita de um ajuste.

6. **Definir os ajustadores das filas do modelo de rede de filas definido para o novo SGBD**. Ajustadores de filas só precisam ser definidos para as filas folhas da rede de filas e somente para aquelas que se deseje ajustar automaticamente. Ou seja, não é obrigatório que cada fila folha da rede de fila possua um ajustador de fila associado, porém, para que uma fila folha seja ajustada automaticamente, é obrigatória a definição de um ajustador para ela. Caso não existam ajustadores de filas que possam ser reusados, um novo ajustador deverá ser implementado. Isso equivale a implementar uma nova classe Java que estenda a classe TuningProcess do Gerente de Execução. Caso já exista um ajustador para determinada fila, este pode ser reusado totalmente ou alterado com outras etapas de ajuste. Isto equivale a implementar novas classes que estendam a classe TuningPhase do Gerente de Execução e informar que estas novas etapas devem ser acrescidas ao ajustador reusado.

É importante salientar que, uma vez definido, um modelo de performance pode ser reusado completamente. Isto é, uma vez definido um modelo de performance para um SGBD, este modelo pode ser disseminado para que outras pessoas possam reusá-lo e assim gerenciar automaticamente qualquer instância desse SGBD. Logo, a definição de um bom modelo de performance será de grande valia para os usuários de SGBDs que precisam de uma ferramenta para gerenciar esse *software* automaticamente.

No capítulo seguinte é descrito o modelo de performance que foi definido para o SGBD PostgreSQL.

4.4 Conclusão

Neste capítulo foi apresentada a arquitetura do DBMS-Analyzer. O framework foi projetado para implementar o ciclo básico da computação autônoma de maneira a possibilitar sua extensão para o gerenciamento de desempenho de diferentes SGBDs. Seus quatro componentes básicos são Gerente de Monitoramento, Gerente de Análise, Gerente de Planejamento e Gerente de Execução. O gerente de monitoramento é auxiliado por componentes de acesso que efetuam o acesso até o ambiente onde o SGBD está executando a fim de proceder a coleta das métricas necessárias. O gerente de execução, por sua vez, utiliza ajustadores de filas para decidir quais ajustes devem ser feitos na configuração do SGBD para que este restabeleça a

meta de desempenho que lhe foi imposta.

Com essa arquitetura extensível, o DBMS-Analyzer torna-se capaz de gerenciar o desempenho de qualquer SGBD utilizando o ciclo básico de gerência da computação autônoma.

Foram detalhados os passos que o desenvolvedor deve seguir para montar uma aplicação do DBMS-Analyzer para gerenciar o desempenho de um SGBD específico. Os passos descritos permitem definir um modelo de performance para qualquer SGBD.

Capítulo 5

Extensão do Framework para o PostgreSQL

Este capítulo tem como finalidade apresentar a extensão do framework apresentado no Capítulo 4. Essa extensão foi implementada, seguindo a metodologia descrita na seção 4.3 do capítulo 4, para monitorar o desempenho do SGBD PostgreSQL e gerenciar sua estrutura de memória. A seção 5.1 apresenta o modelo de rede de filas definido para representar os principais componentes de processamento de consultas do PostgreSQL. Na seção 5.2 é apresentado o conjunto de métricas utilizadas no processo de coleta e como as variáveis operacionais das filas do modelo de rede de filas elaborado são calculadas. A seção 5.3 encerra o capítulo apresentando o ajustador implementado e sua estratégia de ajuste de memória utilizada para gerenciar o desempenho do PostgreSQL.

5.1 Definição da Rede de Filas para o PostgreSQL

Levando em conta as etapas que compõem o processo de execução de consultas implementado pelo PostgreSQL (vide Capítulo 2), os componentes de software que compõem o processo e os recursos de hardware utilizados, foi definida a rede de filas utilizada nesta aplicação.

A rede de filas da aplicação, que modela o processamento de consultas do SGBD PostgreSQL, é composta de oito filas estruturadas em uma árvore de quatro níveis, como visto na figura 5.1.

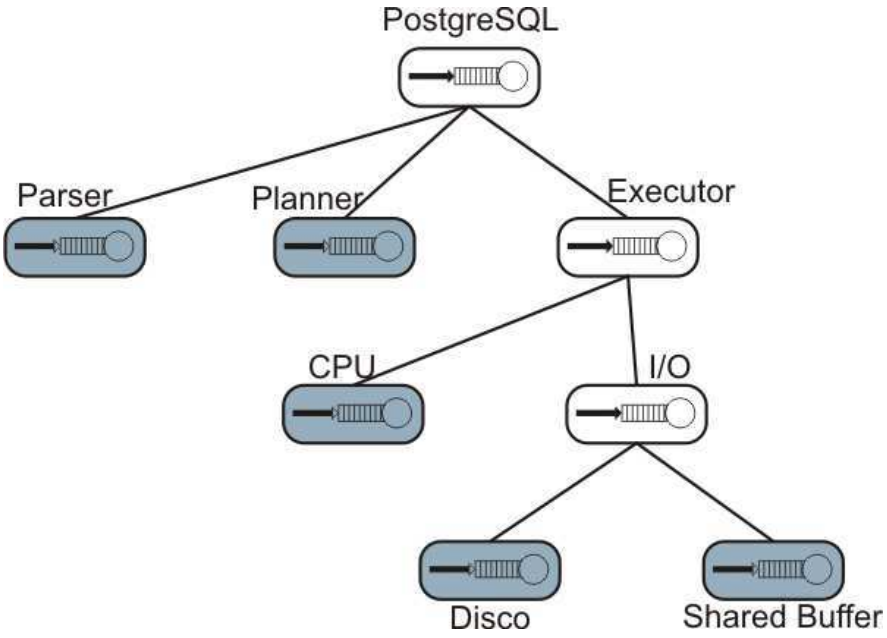

Figura 5.1: *Hierarquia da rede de filas elaborada para o PostgreSQL*

No topo da árvore, ou seja, na raiz da árvore, se encontra a fila que representa o próprio PostgreSQL. Essa fila é desmembrada em três outras filas: Parser, Planner e Executor, que modelam, respectivamente o Parser o Planner e o Executor da arquitetura do PostgreSQL. O componente Rewriter não foi modelado na rede de fila desta aplicação para não torná-la muito complexa.

A fila Executor é subdividida em duas filas que modelam os recursos de hardware consumidos durante essa etapa do processamento. A primeira fila modela o consumo de CPU; enquanto a segunda, as requisições de I/O.

A fila das requisições de I/O ainda foi desmembrada em duas outras filas para modelar as requisições que utilizam o disco magnético (envolvendo acesso físico ao disco) e as que utilizam o Shared Buffer do PostgreSQL.

As filas em destaque representam as folhas da árvore criada. Elas são utilizadas para calcular o tempo médio de resposta do SGBD, o qual é calculado através da soma dos Tempos de Residência das filas folhas de um modelo de rede de filas.

Vale salientar que a relação pai-filho presente na estrutura da rede de fila representa apenas que a fila pai é composta por suas filas filhas, ou seja, os recursos que as filas filhas modelam fazem parte de um recurso de mais alto nível, modelado pelo pai delas. Essa composição não necessita ser completa, ou seja, uma fila pai não necessita ser decomposta em

tantas filhas quanto forem suas subestruturas. Por exemplo, sabemos que a fila PostgreSQL é composta de muito mais recursos que apenas Parser, Planner e Executor. Por exemplo, podemos citar recursos tais como o logger, o coletor de estatísticas e locks utilizados em bloqueios de tabelas. No entanto, optou-se por não representar todos esses recursos no modelo de rede de filas definido devido ao complexo modelo de filas que resultaria, caso todas essas estruturas fossem modeladas.

Esta etapa implementa o passo 1 da metodologia apresentada na seção 4.3 do capítulo 4.

5.2 Métricas Coletadas e Funções de Mapeamento

Para monitorar o desempenho do PostgreSQL é preciso coletar algumas informações a respeito de cada fila definida no modelo de rede de filas elaborado. Essas informações serão utilizadas para calibrar as filas do modelo, ou seja, para calcular suas variáveis operacionais vazão e utilização. Esta seção apresenta as métricas utilizadas no processo de monitoramento do PostgreSQL desta aplicação.

Porém, antes de apresentar as métricas utilizadas no processo de coleta para a gerência de memória do SGBD PostgreSQL, faz-se necessário uma pequena explicação da estrutura do arquivo de log gerado pelo PostgreSQL.

5.2.1 Estrutura do Arquivo de log do PostgreSQL

Para efetuar a coleta das métricas necessárias ao funcionamento do framework, é preciso ativar o coletor de estatísticas do PostgreSQL. As estatísticas coletadas por esse SGBD são gravadas em seu arquivo de log. É desse arquivo que o framework coleta as métricas necessárias para calibrar o modelo de rede de filas definido e efetuar toda a gerência de desempenho do PostgreSQL.

Quando ativado, o coletor de estatísticas do PostgreSQL coleta algumas informações sobre todo comando SQL que é submetido ao SGBD. Tais informações são referentes a cada etapa de processamento do comando SQL — que inclui Parser, Rewriter, Planner e Executor, responsáveis, respectivamente, por verificar a sintaxe do comando; reescrever, se necessário, o comando submetido; gerar um plano de execução para ele e executá-lo.

Em cada uma dessas etapas, o PostgreSQL efetua requisições de I/O para recuperar dados

e faz uso do processador para manipular esses dados. Dentre as estatísticas coletadas, em cada etapa de processamento de um comando SQL, destacam-se:

elapsed tempo total gasto com o processador;

user tempo de uso do processador no modo usuário;

system tempo de uso do processador no modo kernel;

shared blocks reads número de blocos de discos requisitados para leitura ao Shared Buffer e que não foram encontrados, sendo necessária uma requisição de I/O ao disco;

shared blocks written número de blocos de discos requisitados ao Shared Buffer para gravação e que não foram encontrados, sendo necessária uma requisição de I/O ao disco;

duration tempo total decorrido em uma etapa do processamento

Um exemplo de coleta de estatísticas gravada no arquivo de Log do PostgreSQL é apresentado na figura 5.2. A coleta refere-se a etapa EXECUTOR de um comando SQL.

```
2008-01-08 11:20:29.192 BRT [15106] LOG:  EXECUTOR STATISTICS
2008-01-08 11:20:29.192 BRT [15106] DETAIL:  ! system usage stats:
        !       0.002564 elapsed 0.001999 user 0.000000 system sec
        !       [0.302953 user 0.038994 sys total]
        !       0/0 [0/0] filesystem blocks in/out
        !       0/0 [0/1631] page faults/reclaims, 0 [0] swaps
        !       0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
        !       0/0 [306/12] voluntary/involuntary context switches
        ! buffer usage stats:
        !       Shared blocks:      3 read, 0 written, buffer hit rate = 89.00%
        !       Local  blocks:      0 read, 0 written, buffer hit rate = 0.00%
        !       Direct blocks:      0 read, 0 written
2008-01-08 11:20:29.192 BRT [15106] LOG:  duration: 1.661 ms  execute S_3: SELECT COUNT(DISTINCT
(s_i_id)) AS stock_count FROM order_line, stock WHERE ol_w_id = $1 AND ol_d_id = $2 AND ol_o_id < $3
AND ol_o_id >= $4 - 20 AND s_w_id = $5 AND s_i_id = ol_i_id AND s_quantity < $6
2008-01-08 11:20:29.192 BRT [15106] DETAIL:  parameters: $1 = '26', $2 = '8', $3 = '3026', $4 = '3026',
$5 = '26', $6 = '19'
```

Figura 5.2: *Exemplo de seção do log do PostgreSQL*

Uma coleta de estatística como a da figura 5.2 é gerada em cada etapa de processamento de cada comando SQL submetido ao PostgreSQL.

A partir do arquivo de log do PostgreSQL, um componente de acesso implementado para a aplicação efetua a coleta das métricas necessárias para calibrar o modelo de rede de filas

definido. Esse componente de acesso ler o arquivo de log linha a linha e retira os valores de cada métrica utilizada nesta aplicação. A implementação desse componente de acesso conclui o passo 3 da metodologia descrita na seção 4.3 do capítulo 4.

As métricas utilizadas nesta extensão do DBMS-Analyzer para o PostgreSQL, bem como o mapeamento de seus valores em variáveis operacionais são descritos a seguir.

5.2.2 Cálculo das Variáveis Operacionais

Após a definição do modelo de rede de filas que irá representar o SGBD PostgreSQL, é preciso definir como as variáveis operacionais referentes a cada fila do modelo serão calculadas.

Como mencionado anteriormente, as variáveis operacionais são calculadas a partir de um conjunto de métricas que são coletadas do SGBD monitorado. Para esta aplicação, as métricas serão coletadas a partir das estatísticas gravadas no arquivo de log do PostgreSQL. A seguir, é detalhado, para cada fila do modelo de rede de filas definido, como é feito o cálculo de suas variáveis operacionais Vazão e Utilização. Vale salientar que essas duas variáveis operacionais são utilizadas nos cálculos das outras variáveis operacionais: Demanda de Serviço, Tempo de Residência e Tamanho da Fila. A Vazão do Sistema (Vazão da fila raiz do modelo de rede de filas) também é de fundamental importância para o cálculo da Demanda de Serviço das filas restantes. A Utilização do sistema, ou seja, da fila PostgreSQL, não foi calculada pois não foi utilizada em nenhuma etapa da gerência do PostgreSQL pela aplicação.

Esta etapa da aplicação implementa os passos 2 e 4 da metodologia descrita na seção 4.3 do capítulo 4.

Fila PostgreSQL

Para o cálculo da vazão do sistema (fila PostgreSQL) é necessário apenas o número de comandos SQL executados em um intervalo de tempo considerado.

Para calcular o número de comandos SQL executados, o framework conta o número de vezes em que a estatística $duration$ das etapas do Parser aparece no log do PostgreSQL. Isso deve-se ao fato que, durante a execução de um comando SQL, este passa apenas uma única vez pela etapa de Parsing. Sendo assim, para cada comando SQL executado, tem-se uma

etapa de Parsing efetuada, logo, contando-se o número de etapas de Parsing efetuada, tem-se o número de comandos SQL executados.

O intervalo de tempo de monitoramento (T) é derivado a partir dos $timestamps$ das métricas que aparecem ao longo do arquivo de log do PostgreSQL. Efetuando-se a diferença entre a ocorrência do último $timestamp$ e o primeiro, tem-se o intervalo de tempo em que os comandos SQL foram executados.

De posse dessas medidas, a Vazão da fila PostgreSQL (X_0) é calculada com a seguinte fórmula:

$$X_0 = \frac{count(parser.duration)}{T}$$

A função $count()$ conta o número de ocorrências de um conjunto de métricas. A notação $parser.duration$ representa a métrica $duration$ das etapas Parser do arquivo de log do SGBD. Esta notação $< etapa > . < métrica >$ será utilizada ao longo desta seção.

Fila Parser

A Vazão da fila Parser é calculada da mesma forma que a Vazão do Sistema (fila PostgreSQL), ou seja, derivada da métrica $parser.duration$, pois contando-se o número de ocorrências dessa métrica, sabe-se o número de comandos SQL que passaram pelo Parser.

Dessa forma, a Vazão do Parser é calculada utilizando a fórmula:

$$X_{parser} = \frac{count(parser.duration)}{T}$$

A Utilização da fila Parser também é derivada da métrica $parser.duration$. Em vez de contar o número de ocorrências dessa métrica no arquivo de log do PostgreSQL, o framework soma os valores coletados dessa métrica e divide o resultado pelo intervalo de tempo considerado. Nesse caso, a fórmula utilizada para calcular a Utilização da fila Parser é:

$$U_{parser} = \frac{\sum(parser.duration)}{T}$$

Fila Planner

Para o cálculo da Vazão e Utilização da fila Planner são utilizadas as mesmas métricas utilizadas na fila Parser, com o detalhe que o valor da métrica utilizado é aquele gerado para a etapa do Planner, ou seja, é utilizada a métrica $planner.duration$.

A Vazão da fila Planner é calculada da seguinte forma:

$$X_{planner} = \frac{count(planner.duration)}{T}$$

Como se vê, a Vazão do Planner é calculada contando-se o número de ocorrências da métrica $planner.duration$ e dividindo-se o resultado pelo intervalo de tempo considerado.

A Utilização da fila Planner também é calculada de forma semelhante a fila Parser, pois o cálculo envolve a soma dos valores da métrica $planner.duration$ e a divisão desse resultado pelo intervalo de tempo considerado. A fórmula utilizada é:

$$U_{planner} = \frac{\sum(planner.duration)}{T}$$

Fila Executor

Também de forma análoga às filas anteriores, a Vazão e Utilização da fila Executor é calculada utilizando a métrica $duration$ das etapas Executor ($executor.duration$) do log do PostgreSQL.

A fórmula utilizada para o cálculo da Vazão da fila Executor é definida como:

$$X_{executor} = \frac{count(executor.duration)}{T}$$

Contando-se o número de ocorrências da métrica $executor.duration$ deriva-se o número de comandos SQL que passaram nesta etapa.

Para o cálculo da utilização da fila Executor a mesma métrica $executor.duration$ é utilizada. A fórmula para o cálculo da utilização é:

$$U_{executor} = \frac{\sum(executor.duration)}{T}$$

Fila Executor.I/O

Esta fila representa uma das filhas da fila Executor e modela as requisições de I/O solicitadas durante a etapa de execução do plano de consulta (etapa Executor). Para calcular a Vazão da fila Executor.I/O o framework precisa recuperar a quantidade de requisições de I/O que foram efetuadas na etapa de execução do comando SQL. Para tanto, o framework utiliza três métricas básicas:

sharedBlocksReads número de blocos de discos requisitados para leitura ao Shared Buffer e que não foram encontrados, sendo necessária uma requisição de I/O ao disco;

sharedBlocksWritten número de blocos de discos requisitados ao Shared Buffer para gravação e que não foram encontrados, sendo necessária uma requisição de I/O ao disco;

shared blocks reads hit rate (sharedHitRate) fração das requisições de I/O que encontraram seus dados no Shared Buffer do PostgreSQL.

O cálculo da Vazão da fila Executor.I/O é um pouco mais complexo do que o das filas anteriores. Primeiramente, precisa-se calcular o número de requisições lógicas de leitura ($sharedBufferLogicalReads$) que foram solicitadas na etapa do Executor. Para isso, utiliza-se as métricas $sharedBlocksReads$ e $sharedHitRate$ e a seguinte fórmula:

$$sharedBufferLogicalReads = \frac{sharedBlocksReads}{(1 - sharedHitRate)}$$

com $0 \leq sharedHitRate < 1$.

De posse do número de leituras lógicas, basta acrescentar o número de escritas solicitadas na etapa do Executor ($sharedBlocksWritten$) e tem-se o total de requisições lógicas de I/O. Esse total, quando dividido pelo intervalo de tempo de observação, resulta na Vazão da fila Executor.I/O. Esse cálculo é apresentado na fórmula abaixo:

$$X_{i/o} = \frac{\sum(sharedBufferLogicalReads) + \sum(sharedBlocksWritten)}{T}$$

Todas as métricas envolvidas no cálculo da Vazão dessa fila se encontram nas estatísticas das seções referentes ao Executor do arquivo de log do PostgreSQL.

Para calcular a Utilização da fila Executor.I/O, o framework necessita das métricas $executor.duration$, $executor.user$ e $executor.system$ que fornecem, respectivamente, o tempo total decorrido na etapa Executor, o tempo que o SGBD gastou utilizando a CPU em modo usuário e o tempo que o SGBD gastou utilizando a CPU em modo kernel.

A estratégia para calcular o tempo que o SGBD gastou efetuando I/O baseia-se na seguinte hipótese: ao executar um plano de consulta, assume-se que o Executor ou está usando CPU ou está efetuando I/O, ou seja, o tempo total gasto pelo Executor é igual a soma do tempo gasto com CPU e o tempo gasto com I/O. Essa hipótese pode ser formulada pela seguinte relação entre essas medidas:

$$executor.duration = (executor.user + executor.system) + TempoGastoComI/O$$

Através dessa hipótese, fica fácil recuperar o tempo que o executor gastou efetuando requisições de I/O e o cálculo da Utilização da fila Executor.I/O é definido como:

$$U_{i/o} = \frac{\sum(executor.duration) - [\sum(executor.user) + \sum(executor.system)]}{T}$$

Fila Executor.CPU

A Vazão da fila Executor.CPU, que representa o uso de CPU dos comandos SQL durante a etapa Executor, é muito semelhante a Vazão de outras filas, tais como PostgreSQL e Parser. O interesse nessa medida é saber quantos comandos SQL passaram pela CPU. Essa medida é calculada a partir da métrica $parser.duration$ e envolve apenas a contagem do número de ocorrência dessa métrica para saber o número de comandos que foram executados. Como todo comando executado obrigatoriamente irá passar pela CPU, o cálculo da Vazão da fila Executor.CPU utiliza tal métrica e é definido segundo a fórmula abaixo:

$$Xcpu = \frac{count(parser.duration)}{T}$$

O cálculo da Utilização desta fila baseia-se na hipótese definida para a fila Executor.I/O. Neste caso, faz-se necessário recuperar o tempo total que o Executor gastou utilizando a CPU. Desta forma, e usando a hipótese assumida para a fila Executor.I/O, o tempo que o Executor gastou utilizando a CPU é definido como:

$$TempoGastoComCPU = \sum(executor.user) + \sum(executor.system)$$

e a Utilização da fila Executor.CPU é definida como:

$$U_{cpu} = \frac{\sum(executor.user) + \sum(executor.system)}{T}$$

Fila Executor.I/O.Disco

Esta fila representa a fração de requisições lógicas de I/O que foram solicitadas ao disco magnético. Para calcular sua Vazão, o framework necessita do número de blocos de disco

requisitados ao disco. Essa informação pode ser encontrada facilmente nas estatísticas $sharedBlocksReads$ e $sharedBlocksWritten$ das seções Executor do arquivo de log do PostgreSQL.

Sabe-se que a métrica $sharedBlocksReads$ informa o número de blocos de disco requisitados para leitura no disco magnético, enquanto a métrica $sharedBlocksWritten$ informa o número de blocos de disco requisitados para gravação no disco magnético.

De posse dessas duas métricas, pode-se calcular a Vazão da fila Executor.I/O.Disco através da seguinte forma:

$$X_{i/o.disco} = \frac{\sum(sharedBlocksReads) + \sum(sharedBlocksWritten)}{T}$$

O cálculo da Utilização da fila Executor.I/O.Disco é o mais complexo dentre os cálculos vistos até agora. A fórmula envolve algumas suposições e deduções que são explicadas a seguir.

Sejam:

B_{disk} Número de blocos de disco requisitados ao disco magnético;

B_{mem} Número de blocos de disco acessados na memória cache (Shared Buffer);

T_{total} Tempo total gasto com leituras de blocos de disco (tanto fisicamente em disco, como em memória);

T_{disk} Tempo gasto pelo PostgreSQL somente com I/O físico, ou seja, acesso a disco;

T_{mem} Tempo gasto pelo PostgreSQL somente com I/O em memória, ou seja, acesso ao Shared Buffer;

td Tempo médio de 1 acesso físico a um bloco de disco;

tm Tempo médio de 1 acesso a um bloco de disco em memória;

Supõe-se então as seguintes relações

1. $T_{total} = T_{disk} + T_{mem}$
2. $\frac{td}{tm} = K$, onde K é uma constante

3. $T_{disk} = td \times B_{disk}$

4. $T_{mem} = tm \times B_{mem}$

Dessas relações pode-se inferir que $tm = \frac{td}{K}$.

Somando-se as relações 3 e 4 tem-se

$$T_{disk} + T_{mem} = (td \times B_{disk}) + (tm \times B_{mem})$$

Substituindo $tm = \frac{td}{K}$ na equação acima e utilizando a relação 1, tem-se:

$T_{total} = (td \times B_{disk}) + \frac{(td \times B_{mem})}{K}$
$\Rightarrow T_{total} = \frac{[(K \times td \times B_{disk}) + (td \times B_{mem})]}{K}$
$\Rightarrow K \times T_{total} = td \times [(K \times B_{disk}) + B_{mem}]$
$\Rightarrow td = \frac{K \times T_{total}}{[(K \times B_{disk}) + B_{mem}]}$

Atribuindo-se à constante K um valor muito grande, pode-se determinar que na relação $\frac{td}{tm} = K$ o tempo de acesso físico a um bloco de disco é muito maior que o tempo de acesso a um bloco de disco na memória, o que torna a fórmula coerente com a realidade. O valor de K utilizado no cálculo da Utilização é $K = 100000$, ou seja, assume-se que o tempo de acesso físico a um bloco de disco é 100000 vezes maior que o tempo de acesso a um bloco de disco na memória.

Com essas hipóteses pode-se calcular o tempo médio de acesso físico a um bloco de disco através da fórmula

$$td = \frac{K \times T_{total}}{[(K \times B_{disk}) + B_{mem}]}$$

e o tempo total gasto pelo Executor com requisições de I/O físico (T_{disk}) através da fórmula $T_{disk} = td \times B_{disk}$.

A Utilização da fila Executor.I/O.Disco é então calculada através da fórmula

$$U_{i/i.disco} = \frac{T_{disk}}{T}$$

que determina a razão entre o tempo em que o Executor gastou efetuando atividades de I/O físico e o intervalo de tempo no qual os comandos SQL foram executados.

Resta agora encontrar o valor de todas as variáveis, presentes nas duas fórmulas acima, a partir das métricas coletadas no log do PostgreSQL.

Para calcular o valor de T_{total}, são utilizadas as métricas $executor.user$, $executor.system$ e $executor.duration$. A partir dessas métricas, e utilizando-se de um cálculo semelhante ao cálculo da Vazão da fila Executor.I/O, explicado anteriormente, encontra-se T_{total} com a seguinte fórmula:

$$T_{total} = \sum(executor.duration) - [\sum(executor.user) + \sum(executor.system)]$$

O valor de B_{disk} é calculado de modo semelhante ao cálculo da Vazão desta fila, ou seja, somando-se os valores das métricas $sharedblocksreads$ e $sharedblockswritten$. Sendo assim, a fórmula para o cálculo de B_{disk} é

$$B_{disk} = \sum(sharedBlocksReads) + \sum(sharedBlocksWritten)$$

O valor de B_{mem} precisa de uma fórmula utilizada no cálculo da Vazão da fila Executor.I/O, explicado anteriormente e repetida aqui.

$$sharedBufferLogicalReads = \frac{sharedBlocksReads}{(1 - sharedHitRate)}$$

com $0 \leq sharedHitRate < 1$.

onde $sharedBufferLogicalReads$ é o número de requisições de I/O (tanto físicas como em memória cache) feitas pelo Executor, e $sharedHitRate$ é o percentual dessas requisições que foram atendidas com leituras apenas ao Shared Buffer.

Finalmente, com a fórmula para calcular o número total de requisições de I/O, achamos o valor de B_{mem} fazendo

$$B_{mem} = \sum(sharedBufferLogicalReads) - \sum(sharedBlocksReads)$$

que recupera apenas os blocos de disco que foram acessados no Shared Buffer.

Fila Executor.I/O.SharedBuffer

A fila Executor.I/O.SharedBuffer é uma espécie de complemento da fila Executor.I/O.Disco, uma vez que esta representa as atividades de I/O físicos e aquela, as atividades de I/O executadas no Shared Buffer do PostgreSQL. Portanto, os cálculos das variáveis operacionais da fila Executor.I/O.SharedBuffer são bem parecidos com os da fila Executor.I/O.Disco.

Para o cálculo da Vazão, são utilizadas as métricas $sharedBufferLogicalReads$ e $sharedBlocksReads$. O cálculo da Vazão é feito segundo a fórmula

$$X_{i/o.sharedBuffer} = \frac{\sum(sharedBufferLogicalReads) - \sum(sharedBlocksReads)}{T}$$

O cálculo da Utilização da fila Executor.I/O.SharedBuffer é baseado nas mesmas hipóteses e relações que as da fila Executor.I/O.Disco, anteriormente explicadas. Porém, para o cálculo da Utilização, precisa-se descobrir o valor do tempo médio de acesso a 1 bloco de disco em memória (tm). A dedução desta informação é explicada a seguir.

Assumindo-se as mesmas hipóteses do cálculo da Utilização da fila Executor.I/O.Disco, tem-se:

B_{disk} Número de blocos de disco requisitados ao disco magnético;

B_{mem} Número de blocos de disco acessados na memória cache (Shared Buffer);

T_{total} Tempo total gasto com leituras de blocos de disco (tanto fisicamente em disco, como em memória);

T_{disk} Tempo gasto pelo PostgreSQL somente com I/O físico, ou seja, acesso a disco;

T_{mem} Tempo gasto pelo PostgreSQL somente com I/O em memória, ou seja, acesso ao Shared Buffer;

td Tempo médio de 1 acesso físico a um bloco de disco;

tm Tempo médio de 1 acesso a um bloco de disco em memória.

e as seguintes relações:

1. $T_{total} = T_{disk} + T_{mem}$
2. $\frac{td}{tm} = K$, onde K é uma constante
3. $T_{disk} = td \times B_{disk}$
4. $T_{mem} = tm \times B_{mem}$

Somando-se as relações 3 e 4 tem-se:

$$T_{disk} + T_{mem} = (td \times B_{disk}) + (tm \times B_{mem})$$

Substituindo $tm = \frac{td}{K}$ e utilizando a relação 1, obtém-se:

$$T_{total} = (tm \times K \times B_{disk}) + (tm \times B_{mem})$$
$$\Rightarrow T_{total} = tm \times [(K \times B_{disk}) + B_{mem}]$$
$$\Rightarrow tm = \frac{T_{total}}{[(K \times B_{disk}) + B_{mem}]}$$

Essa é a fórmula para estimar o tempo médio de acesso a 1 bloco de disco no Shared Buffer do PostgreSQL.

De posse do tempo médio de acesso a 1 bloco de disco na memória, pode-se estimar o tempo total gasto pelo Executor com requisições de I/O no Shared Buffer (T_{mem}) através da fórmula $T_{mem} = tm \times B_{mem}$ e a Utilização da fila Executor.I/O.SharedBuffer através da fórmula:

$$U_{i/o.sharedBuffer} = \frac{T_{mem}}{T}$$

Os valores dos termos necessários para o cálculo da Utilização são recuperados de forma análoga aos valores da fórmula de Utilização da fila Executor.I/O.Disco. A fim de manter consistência entre as fórmulas, o valor da constante K continua sendo 100000.

Para calcular o valor de T_{total}, são utilizadas as métricas $executor.user$, $executor.system$ e $executor.duration$ e a fórmula:

$$T_{total} = \sum(executor.duration) - [\sum(executor.user) + \sum(executor.system)]$$

O valor de B_{disk} é calculado através da fórmula:

$$B_{disk} = \sum(sharedBlocksReads) + \sum(sharedBlocksWritten)$$

O valor de B_{mem} precisa de uma fórmula utilizada no cálculo da Vazão da fila Executor.I/O, explicada anteriormente.

$$sharedBufferLogicalReads = \frac{sharedBlocksReads}{(1 - sharedHitRate)}$$

com $0 \leq sharedHitRate < 1$.

Finalmente, com a fórmula para calcular o número total de requisições de I/O, achamos o valor de B_{mem} fazendo

$$B_{mem} = \sum(sharedBufferLogicalReads) - \sum(sharedBlocksReads)$$

que recupera apenas os blocos de disco que foram acessados no Shared Buffer.

Todas as fórmulas vistas até agora determinam como a aplicação desenvolvida para o SGBD PostgreSQL calcula as variáveis operacionais das filas do modelo de rede de filas definido, a partir de métricas coletadas pelo coletor de estatísticas do PostgreSQL e gravadas no arquivo de log desse SGBD.

Com essas fórmulas, o framework já é capaz de gerar um relatório de análise de desempenho utilizando a análise operacional; detectar um problema de desempenho automaticamente e informar quais as filas que mais estão contribuindo para a degradação do desempenho do SGBD PostgreSQL.

Com o modelo de rede de filas calibrado com os valores das variáveis operacionais de suas filas, o framework também se torna capaz de gerar um plano de ação e determinar quanto da Demanda de Serviço de cada fila que esta contribuindo para a degradação do desempenho do PostgreSQL precisa ser reduzida de modo a restabelecer a meta de desempenho definida pelo usuário para o SGBD.

A seção seguinte discorre acerca do ajustador implementado para reduzir a demanda de serviço da fila Executor.I/O.Disco e sua estratégia de ajuste de memória que foi desenvolvida para implementar o plano de ação gerado pelo framework para restabelecimento das metas de desempenho definidas para o PostgreSQL.

5.3 Ajustadores da Fila Executor.I/O.Disco

Para essa extensão do *framework* foi implementado um ajustador (*Tuner*) para uma única fila: a fila Executor.I/O.Disco, ou seja, para a extensão desenvolvida para o PostgreSQL, o *framework* irá desencadear um processo de ajuste apenas quando o problema de desempenho detectado pelo framework estiver na fila Executor.I/O.Disco, porém, outros ajustadores podem ser definidos para cada fila do modelo de performance elaborado para o PostgreSQL.

O *Tuner* desenvolvido tem como objetivo reduzir a Demanda de Serviço da fila Executor.I/O.Disco com ajustes feitos apenas na memória do PostgreSQL. A estratégia de ajuste de memória desenvolvida para o PostgreSQL almeja alocar memória para o Shared Buffer e a Work Memory de modo a melhorar o desempenho do SGBD. Vale salientar que o ajustador pode ser melhorado com outros ajustes, por exemplo, com a criação automática de índices. Porém, essa abordagem não é o foco deste trabalho.

O processo de ajuste desenvolvido é composto de duas etapas. A primeira tem o objetivo de determinar qual o espaço de memória que deve ser alocado para o Shared Buffer, enquanto a segunda tem o objetivo de determinar qual o espaço que deve ser alocado para a Work Memory do PostgreSQL.

Ambas as etapas levam em conta a carga de comandos SQL submetida ao PostgreSQL bem como as freqüências com que cada tabela envolvida nos comandos é utilizada. A seguir, as duas etapas de ajuste são descritas com detalhes.

A implementação do ajustador conclui o passo 6 da metodologia descrita na seção 4.3 do capítulo 4.

5.3.1 Estratégia de Ajuste do Shared Buffer

A estratégia para determinar qual o espaço de memória que deve ser alocado para o Shared Buffer de modo a melhorar o desempenho do PostgreSQL é baseada em um problema de otimização clássico: o Problema da Mochila (Knapsack Problem) [CLRS01]. O Problema da Mochila descreve uma situação em que se deve preencher uma mochila com vários objetos de diferentes pesos e valores. A mochila possui uma capacidade máxima de peso que ela consegue suportar. O objetivo é preencher a mochila com os referidos objetos de modo a maximizar o valor total carregado na mochila.

A aplicação do Problema da Mochila como estratégia de determinação do tamanho do Shared Buffer para o PostgreSQL é feita da seguinte maneira: a entidade que faz o papel da mochila, na aplicação, é o próprio Shared Buffer — local onde os objetos são colocados. Os objetos que terão que ser alocados na mochila são tabelas e índices de tabelas. Cada tabela e índice de tabela possui um peso e um valor. Para esta aplicação, o peso de um objeto (tabela ou índice) corresponde ao espaço ocupado por ele na memória (em Mb). Quanto mais espaço um objeto ocupar na memória, mais "pesado" será esse objeto. O valor de cada

objeto é atribuído com base na freqüência com que ele é utilizado.

A coleta de informações sobre os objetos inicia ainda na etapa do monitoramento, quando o framework está analisando o arquivo de log do PostgreSQL. Aproveitando que o componente de monitoramento desenvolvido para analisar as estatísticas do log do PostgreSQL está lendo o arquivo de log do SGBD, o próprio componente de monitoramento se encarrega de salvar, em um arquivo, todos os comandos SQL que foram executados em um intervalo de tempo considerado.

O arquivo gerado com a coleção de comandos SQL é lido, durante o processo de ajuste, pelo Tuner desenvolvido. Nessa fase, um componente auxiliar do Tuner efetua uma análise dos comandos contidos na carga de comandos executada e coleta algumas estatísticas, tais como:

- freqüência de aparição de cada tabela nos comandos SQL;
- tamanho, em Mb, de cada tabela contida nos comandos SQL;
- índices definidos em cada tabela;
- tamanho dos índices definidos em cada tabela.

Essas informações são utilizadas na etapa de ajuste do Shared Buffer e servem para alimentar a entrada do algoritmo da mochila.

A etapa de ajuste do Shared Buffer inicia-se com a recuperação de algumas informações básicas entradas no processo. Tais informações incluem o tamanho da memória RAM do sistema onde o PostgreSQL está executando. Essa informação é importante para o Tuner calcular um tamanho máximo de alocação de memória para o Shared Buffer, que será de 15% do valor da memória RAM do sistema. Esse tamanho máximo será a capacidade máxima da mochila (L), no algoritmo da mochila. O percentual escolhido de 15% é sugerido por experts na administração do PostgreSQL [Mom01].

O algoritmo da mochila é executado duas vezes. Na primeira execução, o algoritmo aloca espaço para as tabelas envolvidas na carga de comando SQL. Para tanto, é preciso atribuir pesos e valores para as tabelas que serão alocadas no Shared Buffer. O peso das tabelas será o espaço ocupado por elas (em Mb) na memória. O valor de uma tabela é dado pela freqüência com que ela aparece na carga de comandos SQL, ou seja, quanto mais uma tabela

é consultada em comandos SQL, mais valiosa ela será. De posse dos objetos (tabelas), seus valores (freqüências), seus pesos (espaço ocupado em memória) e da capacidade máxima da mochila (15% do valor total da RAM do sistema), o algoritmo da mochila é executado. O retorno desse algoritmo é um *array* que informa a quantidade de cada objeto (tabela) alocado na mochila (shared buffer).

A versão do algoritmo utilizada é a Mochila Binária [CLRS01] e tem-se apenas 1 elemento de cada tipo de objeto. Isso significa que uma tabela só terá espaço alocado no Shared Buffer se ela couber completamente nele (não é possível alocar uma fração da tabela) e uma tabela só poderá ser alocada uma única vez no Shared Buffer. Esta restrição na estratégia de ajuste pode levar a crer que não será alocada memória suficiente caso as tabelas acessadas sejam muito grandes, no entanto, considerando as questões expostas na seção 2.6.3 do capítulo 2, o ajustador está ciente de que, caso não seja alocada memória ao Shared Buffer para as referidas tabelas, o próprio S.O. irá se encarregar de efetuar tal tarefa através de seu *cache* de disco.

Uma vez retornados os objetos (tabelas) com as respectivas quantidades que devem ser alocadas na mochila (Shared Buffer), o passo seguinte é determinar o espaço total ocupado pelos objetos que foram alocados, o que é feito somando-se os tamanhos das tabelas que foram alocadas ao Shared Buffer.

Terminado esse passo, ainda resta alocar os índices que utilizam as tabelas, pois, durante o processamento de uma consulta SQL, faz-se necessário ter em memória os dados de tabelas e os índices. Os índices das tabelas são alocados de maneira semelhante às tabelas, exceto por uma pequena mudança na determinação do valor de um índice. O valor de um índice é determinado com base em duas regras:

1. se a tabela na qual o índice é definido teve espaço alocado no Shared Buffer no passo anterior, o valor do respectivo índice será 1 (valor máximo que pode ser atribuído). Isso aumenta as chances de uma tabela e seus respectivos índices ficarem juntos no Shared Buffer.

2. caso contrário, o Tuner irá analisar as freqüências dos predicados nos quais a tabela onde o índice está definido aparece. Cada predicado, em que a tabela onde o índice está definido aparece, tem sua freqüência de aparição nas consultas calculada. Em

seguida, verifica-se, para cada índice definido, se ele possui algum atributo em comum com os atributos envolvidos nos predicados analisados. Através dessa verificação, a freqüência de um índice é calculada. Cada vez que um índice possuir um atributo em comum com um atributo envolvido em um predicado, a freqüência do índice será incrementada com a freqüência de aparição do respectivo predicado. Ao final, quando todos os índices de todas as tabelas forem analisados, os valores dos índices serão atribuídos com o valor dessas freqüências acumuladas.

Determinados os valores e tamanhos dos índices, o algoritmo da mochila é executado novamente. Desta vez, a capacidade máxima da mochila será o espaço que ainda pode ser alocado ao Shared Buffer quando retirado o espaço já alocado às tabelas durante a primeira execução do algoritmo. Sendo assim, a nova capacidade máxima da mochila (L_1) na segunda execução do algoritmo é dada por

$$L_1 = (0,15 \times SystemRAM) - EspacoAlocado\grave{A}sTabelas$$

Encerrada a execução do algoritmo, é determinado o tamanho da segunda mochila, agora com a alocação de índices, somando-se os tamanhos dos índices alocados para o Shared Buffer.

Por fim, o tamanho final do Shared Buffer será a soma dos espaços alocados para os índices e dos espaços alocados para as tabelas.

Dessa forma, o Tuner determina a quantidade de memória que deve ser alocada ao Shared Buffer de modo a comportar o conjunto de objetos mais valioso e que caiba em, no máximo, 15% da RAM total do Sistema.

Vale salientar que o algoritmo da mochila não aloca efetivamente as tabelas ao Shared Buffer. Quem decide qual tabela entra no Shared Buffer ou sai dele é o próprio PostgreSQL, utilizando uma política de alocação LRU [Gro06]. O que o Tuner desenvolvido faz é apenas determinar um tamanho para o Shared Buffer de modo que um conjunto de objetos valiosos caiba inteiramente, e exatamente, no espaço alocado para o Shared Buffer.

5.3.2 Estratégia de Ajuste do Work Memory

A próxima etapa do processo de ajuste da memória do PostgreSQL tem o objetivo de determinar uma alocação de memória adequada para a área de ordenação e hashing do SGBD.

Essa fase também utiliza as estatísticas da carga de comandos SQL geradas na fase anterior. Na fase de ajuste do Work Memory, o Tuner desenvolvido leva em conta as freqüências dos comandos SQL que necessitam de ordenação. Tais comandos são os que possuem cláusulas como ORDER BY, DISTINCT e GROUP BY. Essa informação é importante para saber o percentual de comandos da carga submetida ao SGBD que necessitam efetuar ordenação de dados e, conseqüentemente, precisam de mais Work Memory.

Também são analisados, nesta etapa, o espaço de memória alocado para o Shared Buffer, durante a fase anterior; o número máximo de conexões simultâneas permitidas ao SGBD; uma estimativa do espaço de memória ocupado pelos processos do sistema operacional e o espaço de memória alocado para cache de dados do sistema operacional, que na realidade também é utilizado para fazer o cache de dados do PostgreSQL.

A estratégia desenvolvida para a fase de ajuste do Work Memory inicia-se com uma decisão básica que deve ser tomada levando em conta a freqüência de comandos SQL que precisam de ordenação:

Se a freqüência de comandos SQL que precisam de ordenação for menor que 35%, o Tuner irá ajustar o tamanho do Work Memory utilizando a seguinte fórmula:

$$workMemory = min(16, maxWorkMemoryPerConnection)$$

onde $maxWorkMemPerConnection$ é a quantidade máxima de Work Memory que o framework poderá alocar para o PostgreSQL. O número 16 corresponde ao valor máximo que o framework irá alocar, de fato, para a Work Memory, caso a freqüência de comandos SQL que precisam de ordenação for menor que 35%. Ou seja, o framework poderá calcular um valor para $maxWorkMemPerConnection$ igual a 32Mb, por exemplo. No entanto, se a freqüência de comandos SQL que precisam de ordenação for menor que 35%, o framework não acha necessário alocar toda essa quantidade de memória para a Work Memory. Sendo assim, no máximo, ela irá alocar 16 Mb, caso a freqüência de comandos SQL que precisam de ordenação for menor que 35%. O percentual de 35% foi estabelecido considerando apenas algumas heurísticas, não sendo utilizado nenhum estudo mais específico para determinação deste valor.

O parâmetro $maxWorkMemPerConnection$ é definido em função de três parâmetros básicos: número máximo de conexões simultâneas permitidas ao SGBD (n); tama-

nho do Shared Buffer ($sharedBuffer$); total de memória RAM disponível no Sistema ($SystemRAM$). O valor do parâmetro é calculado da seguinte maneira

$$maxWorkMemoryPerConnection = \frac{[0, 2 \times (SystemRAM - sharedBuffer)]}{n}$$

Em outras palavras, o valor do parâmetro $maxWorkMemPerConnection$ será de 20% da memória restante do sistema, quando retirado o espaço alocado para o Shared Buffer, dividido pelo número máximo de conexões simultâneas permitidas pelo SGBD.

Essa decisão se baseia na hipótese de que, se poucos comandos necessitam de ordenação, não seria bom alocar muita memória para ordenação e penalizar, por exemplo, o cache de dados do sistema operacional, que é utilizado pela maioria dos comandos SQL.

Se a freqüência de comandos SQL que precisam de ordenação for maior que 35%, a decisão do framework também é bem simples: o valor do Work Memory será definido através da fórmula

$$workMemory = max(16, maxWorkMemoryPerConnection)$$

Essa decisão eleva o valor máximo que poderá ser alocado, de fato, ao Work Memory para $maxWorkMemPerConnection$, e o mínimo para o valor de 16Mb. A lógica da decisão se baseia na hipótese de que, se uma quantidade considerável de comandos SQL precisam de ordenação, é prudente penalizar um pouco o cache de dados do sistema operacional para permitir que a ordenação de dados seja feita de forma eficiente, permitindo que os comandos que precisem de ordenação sejam executados mais rapidamente.

Vale salientar que o espaço máximo de memória que será consumido para ordenação, mesmo que todas as conexões permitidas estejam ativas e executando um comando que precise de ordenação, será igual a $[0, 2 \times (SystemRAM - sharedBuffer)]$, ou seja 20% da memória RAM disponível, quando retirado o espaço alocado ao Shared Buffer. Os outros 80% de RAM disponível serão utilizados para cache de dados do sistema operacional e os processos do próprio sistema operacional.

5.4 Conclusão

Neste capítulo foi apresentada a extensão do *framework* para o SGBD PostgreSQL. Foi definida uma rede de filas para modelar os principais recursos utilizados pelo SGBD (Parser,

Planner, Executor, I/O e uso de CPU); um conjunto de métricas do arquivo de log do SGBD foi escolhido para possibilitar os cálculos das variáveis operacionais vazão e utilização das filas definidas; as funções de mapeamento das métricas nas variáveis operacionais vazão e utilização de cada fila também foram definidas e um ajustador foi implementado para ajustar a fila Executor.I/O.Disco através de intervenções na estrutura de memória do PostgreSQL.

Nota-se que, com a rede de filas definida para o SGBD, o *framework* será capaz de analisar o desempenho do PostgreSQL avaliando não só o tempo médio de I/O ou o *hit rate* do *cache* de dados, mas o tempo que as várias etapas de processamento de um comando SQL estão consumindo do SGBD. Com essa avaliação holística, o *framework* é capaz de identificar qual a fila que está consumindo o maior tempo de processamento do SGBD e então identificar onde está o real gargalo do sistema.

Viu-se também que, embora a rede de filas definida permita uma avaliação holística de desempenho do PostgreSQL, os ajustes efetivos no SGBD só serão feitos em suas estruturas de memória, pois, por questões de tempo e complexidade, apenas um ajustador foi implementado nesta extensão do *framework*. Porém, este ajustador poderá ser alvo de trabalhos futuros que amplie sua capacidade de ajustar a fila Executor.I/O.Disco da rede de filas definida, acrescentando-lhe a capacidade de, por exemplo, sugerir a criação de índices para reduzir a atividade de I/O em disco.

Em resumo, a extensão implementada para o PostgreSQL permitirá ao *framework* elaborar um diagnóstico de problema de desempenho utilizando de uma análise holística do SGBD, porém, apenas irá efetuar algum ajuste caso o problema detectado esteja na fila Executor.I/O.Disco e tal ajuste estará restrito a configurar as estruturas de memória do PostgreSQL.

Capítulo 6

Resultados Obtidos

Este capítulo tem como objetivo apresentar os resultados obtidos com os testes de avaliação executados para a extensão do *framework* desenvolvida para o PostgreSQL. A seção 6.1 descreve o ambiente em que os testes foram executados, incluindo detalhes de configurações do host, do SGBD utilizado, e do TPC [(TP07]. Em seguida, a seção 6.2 descreve a metodologia de execução dos testes. A seção 6.3 apresenta os casos de testes aos quais o *framework* foi submetido e os resultados obtidos. Por fim, a seção 6.4 apresenta uma rápida discussão acerca dos resultados alcançados pelo *framework*.

6.1 Ambiente de Teste

Todos os testes especificados foram executados em um PC equipado com um processador AMD Athlon 64 3000+ e 1GB de memória RAM. Nele foi instalado o sistema operacional Red Hat Linux EL e o SGBD PostgreSQL versão 8.2.5 e criado um banco de dados com aproximadamente 10GB de espaço ocupado em disco rígido, seguindo o esquema relacional do TPC-C [(TP07]. O espaço ocupado em disco pelo banco de dados criado é determinado pelo número de registros na tabela warehouse do TPC. No banco de dados utilizado nos testes, foram utilizados 100 warehouses.

As tabelas e índices envolvidos no banco de dados com seus respectivos tamanhos estão descritos no apêndice A.

Os testes foram executados utilizando o software BenchmarkSQL [Lus06], que implementa a especificação TPC-C. Algumas transações contidas em um carga típica do TPC são

detalhadas no apêndice A.

O software BenchmarkSQL permite que sejam configurados os percentuais de participação de cada transação na carga de comandos SQL submetida. Os valores padrões dos percentuais de cada transação são apresentados na tabela 6.1.

Transação	**Percentual default de participação na carga do TPC-C**
Payment	44%
Order-Status	4%
Delivery	4%
Stock-Level	4%

Tabela 6.1: Valores padrões dos percentuais de participação das transações na carga do TPC-C

O percentual restante (caso os 100% não sejam completados) é preenchido com uma nova transação denominada new_order [(TP07]. O software também possibilita a configuração do número de clientes simulados, os quais submetem as transações ao SGBD. Na execução dos testes foram utilizados 10 clientes simultâneos.

Um resumo esquemático do ambiente padrão de testes é apresentado na tabela 6.2 seguir:

Configuração do Host	
Processador	AMD Atlhon 65 3000+
Memória RAM	1 GB
SGBD	PostgreSQL v8.2.5
Sistema Operacional	Red Hat Linux EL
Tamanho do Banco de Dados	10 GB
Configuração do TPC	
Transação	**Percentual de participação na carga do TPC-C**
Payment	44%
Order-Status	4%
Delivery	4%
Stock-Level	4%
Clientes Simultâneos	10
Warehouses utilizados	100
Configuração do PostgreSQL	
Shared Buffer Size	30MB
Work Memory Size	1 MB
Máximo de Conexões Permitidas	10

Tabela 6.2: Ambiente Padrão de Teste

Este ambiente, com essas configurações, é o ambiente padrão utilizado nos testes. Qualquer alteração feita nesse ambiente para a execução de algum caso de teste será informada.

6.2 Metodologia

Nesta seção, é apresentada a metodologia utilizada para a execução dos casos de testes definidos para avaliar a eficácia do *framework* desenvolvido. Cada caso de teste é especificado através de um conjunto de parâmetros que fazem parte da configuração do ambiente de teste, tais como percentual de transações utilizado; valor dos parâmetros shared_buffer e work_memory do arquivo de configuração do PostgreSQL; dentre outros.

Para verificar se o *framework* apresenta boa eficácia, foram desenvolvidos quatro casos de testes que descrevem quatro tipos de situações distintas para as quais o *framework* deve sugerir ajustes nos parâmetros shared_buffer e work_mem do SGBD PostgreSQL. Essas situações simulam quatro tipos de carga de transações SQL: a carga de transações padrão do TPC-C; um tipo de carga cujos comandos SQL acessam, predominantemente, tabelas pequenas; um tipo de carga cujos comandos SQL acessam, predominantemente, tabelas muito grandes e, por fim, uma carga cujos comandos SQL exijam muita ordenação do SGBD. Além disso, é imposta uma meta de tempo de resposta para cada caso de teste para que o *framework* possa reagir e tentar restabelecer a meta de tempo de resposta quando ela for extrapolada.

Cada tipo de carga descrita é determinado pelo percentual de participação de cada tipo de transação do TPC-C na carga simulada.

Para cada caso de teste a avaliação é feita da seguinte forma: coleta-se informações do desempenho do PostgreSQL executando com a configuração padrão dos parâmetros shared_buffer (valor default = 30Mb) e work_mem (valor default = 1Mb), bem como as outras configurações padrões do ambiente de teste (vide seção 6.1). Em seguida, os valores dos parâmetros shared_buffer e work_mem são alterados para os valores sugeridos pelo *framework* e então, procede-se uma nova coleta de informações do desempenho do PostgreSQL. As informações de desempenho, antes e após acatadas as sugestões do *framework*, são comparadas para medir o impacto causado no desempenho do SGBD pelos ajustes do *framework*. Esse impacto será medido em termos de redução no tempo médio de resposta dos comandos SQL submetidos e do impacto nos valores das variáveis operacionais das filas da rede de filas

elaborada para o PostgreSQL. Também será verificado se o ajuste sugerido pelo *framework* foi capaz de restabelecer a meta de tempo de resposta imposta em cada caso de teste.

Para coletar as informações de desempenho, várias execuções do TPC são submetidas ao PostgreSQL em uma mesma configuração. Cada execução consiste em 10 clientes submetendo transações ao PostgreSQL, de acordo com os percentuais definidos, durante um intervalo de tempo de 30 segundos. Ao final das execuções, é gerado um relatório de desempenho contendo as médias dos valores coletados nas várias execuções.

O software utilizado para simular clientes acessando o PostgreSQL foi o BenchmarkSQL [Lus06], que implementa o padrão TPC-C [(TP07]. Este software permite ao usuário determinar os percentuais das transações que irão compor a carga a ser submetida ao SGBD. As transações que compõem uma carga típica do TPC-C estão definidas no apêndice A. O BenchmarkSQL também permite definir o número de clientes que serão simulados, assim como o tempo que esses clientes passarão submetendo transações ao SGBD. Durante esse intervalo de tempo, o BenchmarkSQL se encarrega de, a cada nova transação a ser executada, sortear, conforme o percentual definido, a próxima transação que será executada, bem como gerar valores aleatórios para as constantes que fazem parte dos predicados dos comandos SQL da transação. Portanto, não se sabe, a priori, qual o tipo da próxima transação que será executada, nem os valores das constantes utilizadas.

A descrição detalhada de cada caso de teste e os resultados obtidos estão na próxima seção.

6.3 Resultados dos Testes Executados

Nesta seção são apresentados os casos de testes e os resultados obtidos após suas execuções. Cada caso de teste é especificado em termos de parâmetros do ambiente de teste descrito na seção 6.1 e acompanhado de uma pequena discussão acerca dos resultados alcançados. É informado, em cada caso de teste, o percentual das transações que irão compor a carga submetida no teste, os valores dos parâmetros shared_buffer e work_mem do PostgreSQL, a meta de tempo de resposta imposta ao SGBD para a situação do caso de teste, dentre outros. Quando esses valores não forem informados, assume-se que tais parâmetros são configurados com os valores padrões informados na seção 6.1.

6.3.1 Avaliação da eficácia do ajuste no ambiente padrão de teste

Este caso de teste tem, como principal objetivo, verificar o impacto do ajuste do *framework* quando o PostgreSQL está submetido no ambiente padrão de teste. O ambiente padrão de teste, descrito na seção 6.1, é relembrado aqui:

Configurações do TPC

- composição da carga submetida

- Payment: 44%
- Order-Status: 4%
- Delivery: 4%
- Stock-Level: 4%

N° de clientes simultâneos: 10

N° de warehouses utilizados: 100

Configuração do PostgreSQL

- Tamanho do Shared Buffer: 30Mb
- Tamanho do Work Memory: 1 Mb
- Número máximo de conexões permitidas: 10

A meta de tempo de resposta imposta para este caso de teste foi de 0,8ms. Esse tempo determina o tempo médio de resposta que cada comando SQL submetido ao SGBD deve satisfazer. Note que este tempo não é o tempo da transação, e sim dos comandos SQL que a compõe. Vale salientar que, apesar de a meta de tempo de resposta determinada ser baixa, o SGBD está submetido em um ambiente com vários clientes executando várias transações simultaneamente. Os tempos de resposta de todos esses comandos submetidos, quando somados, podem afetar de forma negativa o desempenho do SGBD.

Resultado Obtido

Após a execução do caso de teste, verificou-se que o PostgreSQL apresentava-se na situação de desempenho descrita pela figura 6.1.

Fila	Utilizacao	Vazao	Demanda de Servico	Tempo de Residencia	Tamanho Fila
Executor.CPU	0,012021	19,626686	0,000613	0,000621	0,012173
Executor.IO	0,924566	363,728778	0,048679	0,869729	16,340689
IO.Disco	0,924543	104,287962	0,048678	0,869193	16,330942
IO.SharedBuffer	0,000230	259,440817	0,000012	0,000012	0,000230
PostgreSQL	0,000000	19,626686	0,000000	0,000000	0,000000
PostgreSQL.Executor	0,925762	67,461751	0,048740	0,895705	16,840641
PostgreSQL.Parser	0,009897	19,627503	0,000502	0,000508	0,010006
PostgreSQL.Planner	0,010878	67,393006	0,000551	0,000569	0,011261
Tempo de Resposta estimado	0,870902 ms				

Figura 6.1: *Situação de desempenho do PostgreSQL no ambiente padrão de teste*

Diante desta situação, o *framework* sugeriu as seguintes configurações para os parâmetros shared_buffer e work_mem do SGBD:

- shared_buffer = 145 Mb
- work_mem = 16 Mb

Após acatadas todas as sugestões do *framework*, a situação de desempenho do PostgreSQL alterou-se para a indicada na figura 6.2.

Fila	Utilizacao	Vazao	Demanda de Servico	Tempo de Residencia	Tamanho Fila
Executor.CPU	0,012310	20,021072	0,000624	0,000632	0,012473
Executor.IO	0,901234	307,756325	0,048733	0,729090	13,128080
IO.Disco	0,901206	73,953088	0,048732	0,728676	13,120431
IO.SharedBuffer	0,000283	233,803237	0,000015	0,000015	0,000283
PostgreSQL	0,000000	20,021072	0,000000	0,000000	0,000000
PostgreSQL.Executor	0,902432	74,071132	0,048794	0,746390	13,458626
PostgreSQL.Parser	0,009095	20,021072	0,000457	0,000461	0,009183
PostgreSQL.Planner	0,006514	73,999802	0,000327	0,000329	0,006559
Tempo de Resposta estimado	0,730113 ms				

Figura 6.2: *Situação de desempenho do PostgreSQL no ambiente padrão de teste, após ajustes sugeridos*

O impacto, em termos percentuais, dos ajustes sugeridos pelo *framework* nas filas do modelo de rede de filas elaborado para o PostgreSQL é apresentado na figura 6.3.

Fila	Utilização	Vazão	Demanda de Serviço	Tempo de Residência	Tamanho da Fila
Executor.CPU	2,40%	2,01%	1,79%	1,77%	2,46%
Executor.IO	-2,52%	-15,39%	0,11%	-16,17%	-19,66%
IO.Disco	-2,52%	-29,09%	0,11%	-16,17%	-19,66%
IO.SharedBuffer	23,04%	-9,88%	25,00%	25,00%	23,04%
PostgreSQL	0,00%	2,01%	0,00%	0,00%	0,00%
PostgreSQL.Executor	-2,52%	9,80%	0,11%	-16,67%	-20,08%
PostgreSQL.Parser	-8,10%	2,01%	-8,96%	-9,25%	-8,23%
PostgreSQL.Planner	-40,12%	9,80%	-40,65%	-42,18%	-41,75%
Tempo de Resposta Estimado	**-16,17%ms**				

Figura 6.3: *Comparação Antes - Após ajustes, no ambiente padrão de teste*

Como se pode observar na figura 6.3, houve uma diminuição de 16,17% no tempo de resposta dos comandos submetidos ao SGBD, após implementados os ajustes sugeridos pelo *framework*. Vale ressaltar, também, a diminuição da vazão do disco (menos blocos de discos lidos fisicamente) e o aumento da utilização do Shared Buffer Cache do PostgreSQL (maior número de leituras a blocos de discos atendidas na memória).

O impacto causado pelo ajuste do *framework* nas variáveis operacionais das filas são indicados nos gráficos das figuras 6.4, 6.5, 6.6, 6.7 e 6.8.

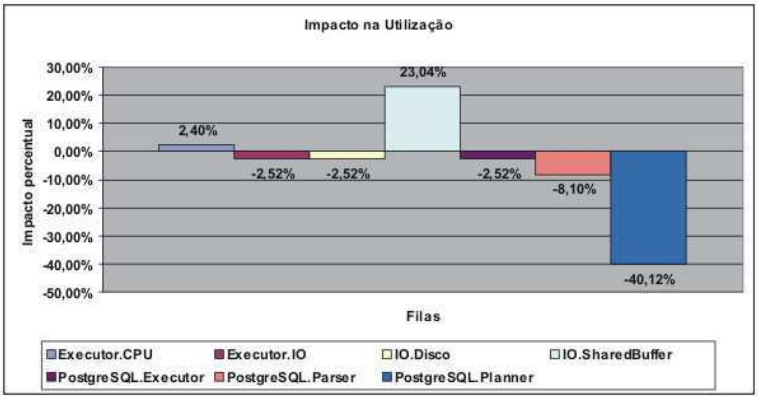

Figura 6.4: *Impacto do ajuste do framework na utilização das filas, no ambiente padrão de teste*

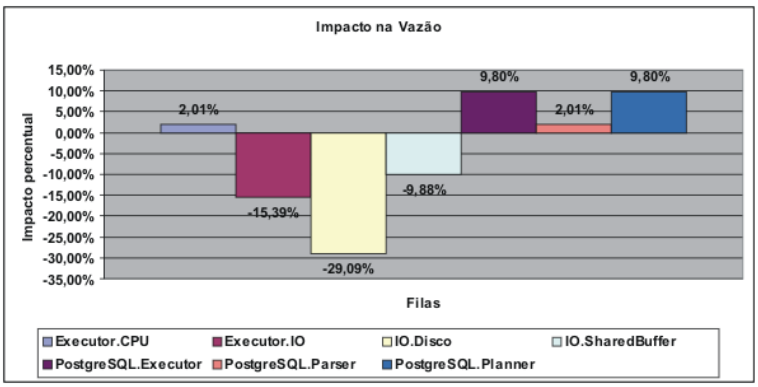

Figura 6.5: *Impacto do ajuste do framework na vazão das filas, no ambiente padrão de teste*

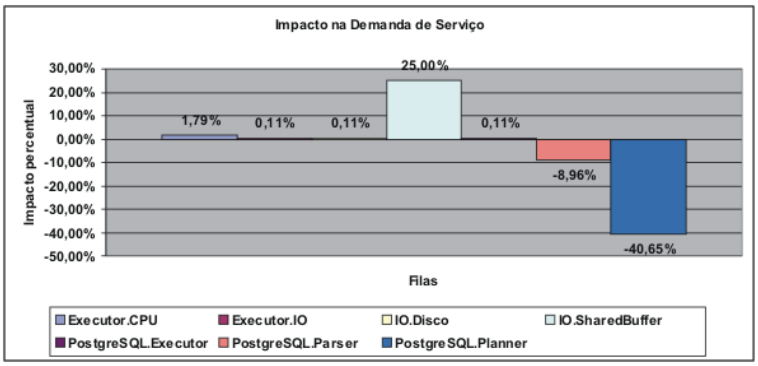

Figura 6.6: *Impacto do ajuste do framework na demanda de serviço das filas, no ambiente padrão de teste*

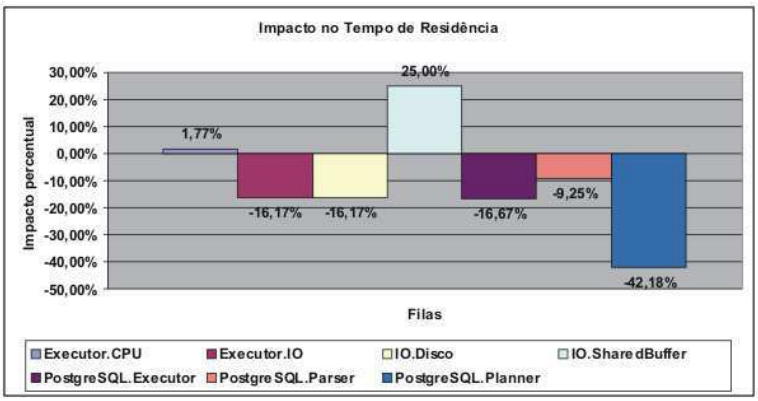

Figura 6.7: *Impacto do ajuste do framework no tempo de residência das filas, no ambiente padrão de teste*

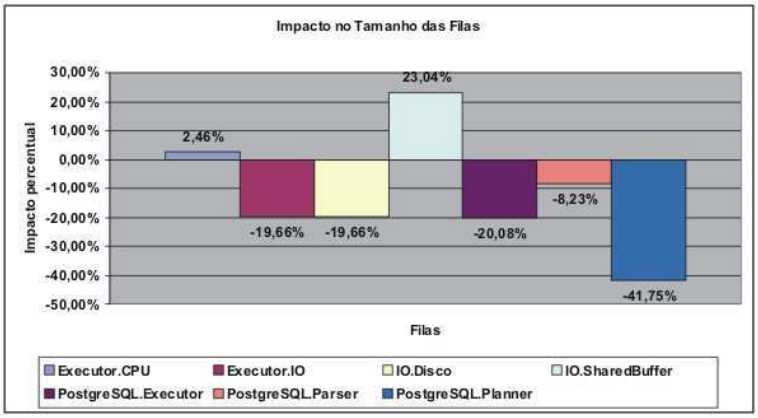

Figura 6.8: *Impacto do ajuste do framework no tamanho das filas, no ambiente padrão de teste*

Através dos gráficos das figuras 6.4, 6.5, 6.6, 6.7 e 6.8, é possível destacar os impactos causados pelo ajuste do *framework* nas variáveis operacionais das filas IO.SharedBuffer, IO.Disco e PostgreSQL.Planner. Um avaliação rápida permite concluir que houve uma redução na atividade de disco compensada com um aumento da atividade de I/O no shared buffer cache do PostgreSQL, ou seja, o disco passou a ser menos procurado pelas requisições de I/O. Vale destacar também a redução da atividade do Planner, que contribuiu bastante para redução no tempo de resposta médio dos comandos executados.

6.3.2 Avaliação da eficácia do ajuste em um ambiente que acesse predominantemente tabelas muito pequenas

Este caso de teste tem, como principal objetivo, verificar o impacto do ajuste do *framework* quando o PostgreSQL está executando em um ambiente cuja carga contém comandos SQL que acessam, predominantemente, tabelas pequenas, tais como **warehouse**, **district**, **item**, **new_order** (ver apêndice A.

Esta situação de teste reflete o inverso da situação descrita na seção 6.3.4, ou seja, verificar qual a sugestão de configuração de memória proposta pelo *framework* quando as tabelas mais acessadas pelos comandos SQL da carga submetida ao SGBD ocupam pouco espaço na memória do sistema.

A configuração do ambiente do teste é resumida a seguir:

Configuração do TPC

- composição da carga submetida

- Payment: 100%
- Order-Status: 0%
- Delivery: 0%
- Stock-Level: 0%

Configuração do PostgreSQL

- padrão

A meta de tempo de resposta para este caso de teste é de 0,6ms.

Resultado Obtido

Após a execução do caso de teste, verificou-se que o PostgreSQL apresentava-se na situação de desempenho descrita pela figura 6.9.

Diante desta situação, o *framework* sugeriu as seguintes configurações para os parâmetros shared_buffer e work_mem do SGBD:

- shared_buffer = 88 Mb

Fila	Utilizacao	Vazao	Demanda de Servico	Tempo de Residencia	Tamanho Fila
Executor.CPU	0,028980	18,467745	0,001584	0,001632	0,029876
Executor.IO	0,912039	540,726364	0,050646	0,665880	11,801187
IO.Disco	0,912003	109,968172	0,050644	0,665512	11,794765
IO.SharedBuffer	0,000357	430,758192	0,000020	0,000020	0,000357
PostgreSQL	0,000000	18,467745	0,000000	0,000000	0,000000
PostgreSQL.Executor	0,914918	341,725346	0,050803	0,694750	12,311127
PostgreSQL.Parser	0,009382	18,467745	0,000508	0,000513	0,009474
PostgreSQL.Planner	0,061706	341,715998	0,003478	0,011595	0,194350
Tempo de Resposta estimado	0,663326 ms				

Figura 6.9: *Situação de desempenho do PostgreSQL no ambiente acessando tabelas pequenas*

- work_mem = 16 Mb

Após acatadas todas as sugestões do *framework*, a situação de desempenho do PostgreSQL alterou-se para a indicada na figura 6.10.

Fila	Utilizacao	Vazao	Demanda de Servico	Tempo de Residencia	Tamanho Fila
Executor.CPU	0,030892	20,660854	0,001525	0,001575	0,031929
Executor.IO	0,859907	541,165238	0,043204	0,406382	7,943317
IO.Disco	0,859863	88,088525	0,043202	0,406129	7,938523
IO.SharedBuffer	0,000442	453,076713	0,000022	0,000022	0,000442
PostgreSQL	0,000000	20,660854	0,000000	0,000000	0,000000
PostgreSQL.Executor	0,862942	428,588825	0,043354	0,421140	8,232036
PostgreSQL.Parser	0,080556	20,662358	0,004747	0,128865	2,053143
PostgreSQL.Planner	0,094457	428,561377	0,005454	0,009283	0,157362
Tempo de Resposta estimado	0,507882 ms				

Figura 6.10: *Situação de desempenho do PostgreSQL, após ajustes sugeridos, no ambiente acessando tabelas pequenas*

O impacto, em termos percentuais, dos ajustes sugeridos pelo *framework* nas filas do modelo de rede de filas elaborado para o PostgreSQL é apresentado na figura 6.11.

Fila	Utilização	Vazão	Demanda de Serviço	Tempo de Residência	Tamanho da Fila
Executor.CPU	6,60%	11,88%	-3,72%	-3,49%	6,87%
Executor.IO	-5,72%	0,08%	-14,69%	-38,97%	-32,69%
IO.Disco	-5,72%	-19,90%	-14,69%	-38,97%	-32,69%
IO.SharedBuffer	23,81%	5,18%	10,00%	10,00%	23,81%
PostgreSQL	0,00%	11,88%	0,00%	0,00%	0,00%
PostgreSQL.Executor	-5,68%	25,42%	-14,66%	-39,38%	-33,13%
PostgreSQL.Parser	758,62%	11,88%	834,45%	25019,88%	21571,34%
PostgreSQL.Planner	53,08%	25,41%	56,81%	-19,94%	-19,03%
Tempo de Resposta Estimado	**-23,43%ms**				

Figura 6.11: *Comparação Antes - Após ajustes, no ambiente acessando tabelas pequenas*

De acordo com a figura 6.11, observa-se que o ajuste sugerido pelo *framework* provocou uma diminuição do tempo de resposta médio dos comandos executados no SGBD de 23,43%. Nota-se também uma grande diminuição na atividade de I/O física do SGBD, o que pode

ser comprovado pela redução no tempo de residência e da vazão da fila IO.Disco, a qual representa a atividade de disco. O Shared Buffer Cache do PostgreSQL, por sua vez, passou a ser mais utilizado pelo SGBD, como pode se observar no incremento da utilização da fila IO.SharedBuffer, que representa a atividade de I/O no Shared Buffer do PostgreSQL.

O impacto do ajuste do *framework* nas variáveis operacionais das filas são indicados nos gráficos das figuras 6.12, 6.13, 6.14, 6.15 e 6.16.

Destaca-se, nos gráficos, o impacto causado na fila PostgreSQL.Parser, que teve um aumento extraordinário em sua utilização (figura 6.12), causando um aumentando considerável em seu tempo de residência. No entanto, como o valor absoluto do tempo de residência da fila PostgreSQL.Parser, após os ajustes do *framework*, foi de 0,128865ms, o grande aumento percentual ocorrido não foi capaz de afetar negativamente o tempo de resposta do sistema, que diminuiu 23,43% (figura 6.11). Essa redução no tempo de resposta pode ser explicada pela redução ocorrida nos tempos de residência da maioria das filas (figura 6.15), principalmente na fila I/O.Disco, que obteve uma redução de 38,97% em seu tempo de residência.

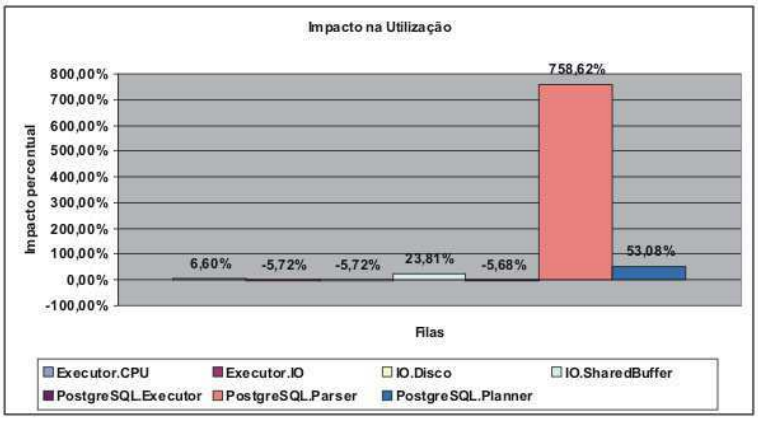

Figura 6.12: *Impacto do ajuste do framework na utilização das filas, no ambiente acessando tabelas pequenas*

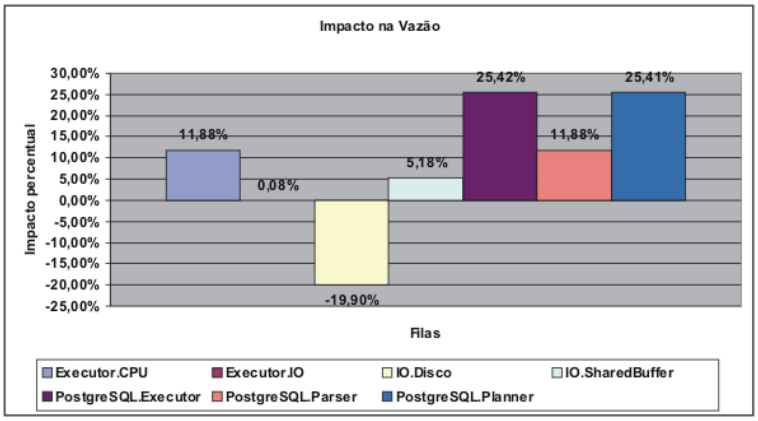

Figura 6.13: *Impacto do ajuste do framework na vazão das filas, no ambiente acessando tabelas pequenas*

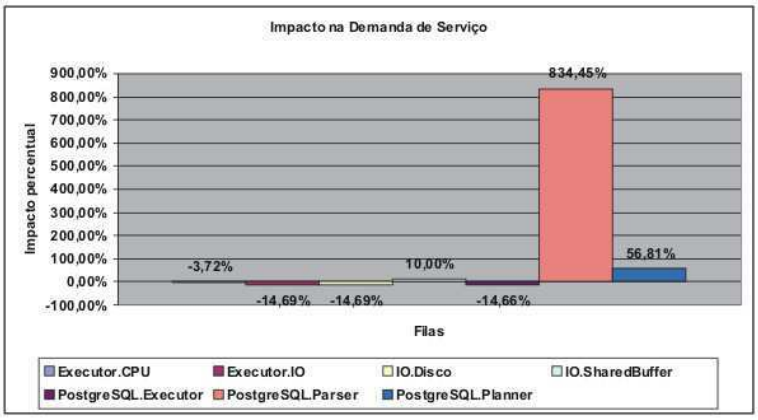

Figura 6.14: *Impacto do ajuste do framework na demanda de serviço das filas, no ambiente acessando tabelas pequenas*

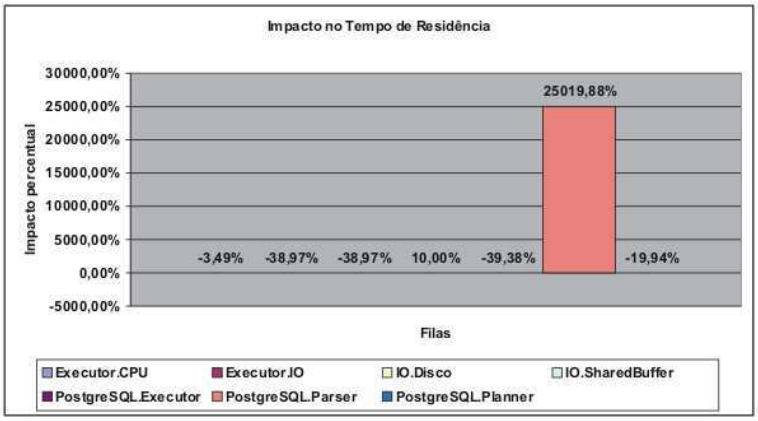

Figura 6.15: *Impacto do ajuste do framework no tempo de residência das filas, no ambiente acessando tabelas pequenas*

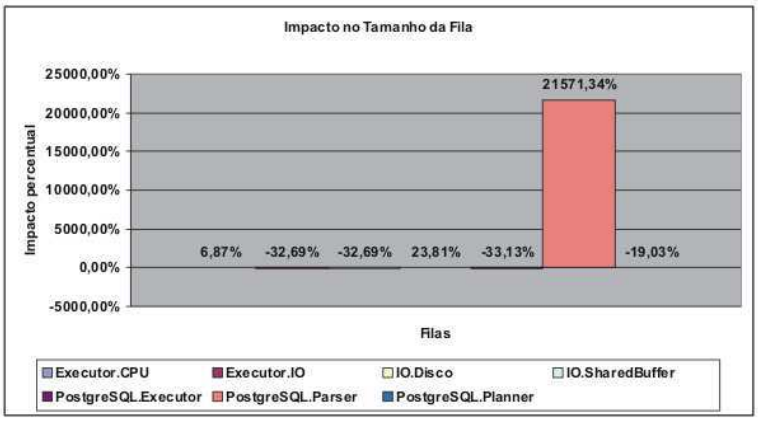

Figura 6.16: *Impacto do ajuste do framework no tamanho das filas, no ambiente acessando tabelas pequenas*

6.3.3 Avaliação da eficácia do ajuste em um ambiente que exija muita ordenação

Este caso de teste tem, como principal objetivo, verificar o impacto do ajuste do *framework* quando o PostgreSQL está executando em um ambiente cuja carga contém, predominantemente, comandos SQL que exijam ordenação.

Esta situação é interessante para demonstrar qual a configuração de memória o *framework* sugere quando o SGBD precisa utilizar suas principais estruturas de memória: Shared Buffer e Work Memory.

A configuração do ambiente do teste é resumida a seguir:

Configuração do TPC

- composição da carga submetida

- Payment: 0%
- Order-Status: 0%
- Delivery: 0%
- Stock-Level: 100%

Configuração do PostgreSQL

- padrão

A meta de tempo de resposta determinada para este caso de teste é de 1ms.

Resultado Obtido

Após a execução do caso de teste, verificou-se que o PostgreSQL apresentava-se na situação de desempenho descrita pela figura 6.17.

Fila	Utilizacao	Vazao	Demanda de Servico	Tempo de Residencia	Tamanho Fila
Executor.CPU	0,407619	3,643474	0,111437	0,198028	0,733268
Executor.IO	0,703900	493,675705	0,208540	1,051967	3,163045
IO.Disco	0,703881	132,489757	0,208535	1,051795	3,162576
IO.SharedBuffer	0,000191	361,185948	0,000056	0,000056	0,000191
PostgreSQL	0,000000	3,643474	0,000000	0,000000	0,000000
PostgreSQL.Executor	0,737854	440,922389	0,217713	1,232693	3,730926
PostgreSQL.Parser	0,008816	3,643474	0,002190	0,002414	0,009723
PostgreSQL.Planner	0,066249	440,893424	0,018399	0,019918	0,071783
Tempo de Resposta estimado	1,272211 ms				

Figura 6.17: *Situação de desempenho do PostgreSQL no ambiente exigindo muita ordenação*

Diante desta situação, o *framework* sugeriu as seguintes configurações para os parâmetros shared_buffer e work_mem do SGBD:

- shared_buffer = 49 Mb
- work_mem = 19 Mb

Após ajustado apenas o parâmetro work_mem, a situação de desempenho do PostgreSQL alterou-se para a indicada na figura 6.18.

Fila	Utilizacao	Vazao	Demanda de Servico	Tempo de Residencia	Tamanho Fila
Executor.CPU	0,429225	4,192312	0,104610	0,188913	0,786500
Executor.IO	0,614575	573,076377	0,165918	0,696864	2,221492
IO.Disco	0,614557	143,696786	0,165913	0,696748	2,221169
IO.SharedBuffer	0,000180	429,379591	0,000048	0,000048	0,000180
PostgreSQL	0,000000	4,192312	0,000000	0,000000	0,000000
PostgreSQL.Executor	0,654018	581,069941	0,175404	0,834116	2,667562
PostgreSQL.Parser	0,008846	4,192312	0,002056	0,002155	0,009258
PostgreSQL.Planner	0,084203	581,046087	0,020196	0,022624	0,095249
Tempo de Resposta estimado	0,910488 ms				

Figura 6.18: *Situação de desempenho do PostgreSQL em ambiente exigindo ordenação, após ajuste do work_mem*

Fila	Utilização	Vazão	Demanda de Serviço	Tempo de Residência	Tamanho da Fila
Executor.CPU	5,30%	15,06%	-6,13%	-4,60%	7,26%
Executor.IO	-12,69%	16,08%	-20,44%	-33,76%	-29,77%
IO.Disco	-12,69%	8,46%	-20,44%	-33,76%	-29,77%
IO.SharedBuffer	-5,76%	18,88%	-14,29%	-14,29%	-5,76%
PostgreSQL	0,00%	15,06%	0,00%	0,00%	0,00%
PostgreSQL.Executor	-11,36%	31,79%	-19,43%	-32,33%	-28,50%
PostgreSQL.Parser	0,34%	15,06%	-6,12%	-10,73%	-4,78%
PostgreSQL.Planner	27,10%	31,79%	9,77%	13,59%	32,69%
Tempo de Resposta Estimado	-28,43%ms				

Figura 6.19: *Comparação Antes - Após ajuste do work_mem, no ambiente exigindo muita ordenação*

O impacto, em termos percentuais, do ajuste do work_mem sugerido pelo *framework* nas filas do modelo de rede de filas elaborado para o PostgreSQL é apresentado na figura 6.19.

O impacto causada apenas pelo ajuste do parâmetro work_mem nas variáveis operacionais das filas são indicados nos gráficos das figuras 6.20, 6.21, 6.22, 6.23 e 6.24.

Analisando esses gráficos, pode-se notar o nítido impacto que o ajuste apenas do parâmetro work_mem causou na utilização e na vazão das filas do sistema. A utilização das filas Executor.I/O, I/O.Disco e PostgreSQL.Executor obtiveram reduções de mais de 10% e a fila I/O.SharedBuffer, de 5,76% (figura 6.20). A vazão de todas as filas do sistema subiram em pelo menos 8% (figura 6.21). Essa nova situação das filas indica que o SGBD passou a efetuar mais trabalho (maior vazão) em menos tempo (menor utilização), o que ocasionou a redução de 28,43% no tempo de resposta do sistema.

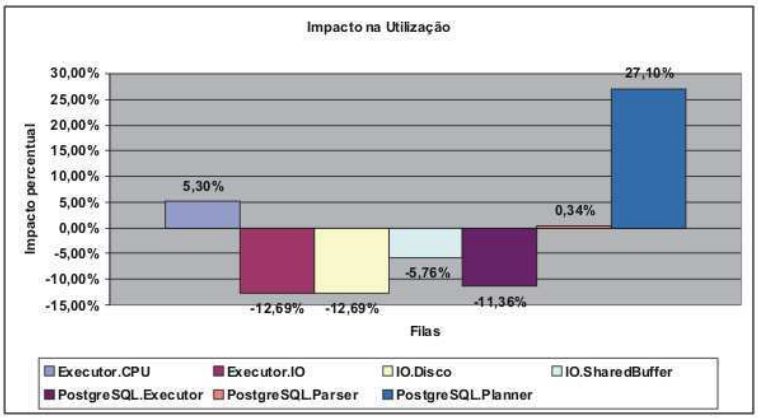

Figura 6.20: *Impacto do ajuste do framework na utilização das filas, no ambiente exigindo ordenação (com ajuste apenas do work_mem)*

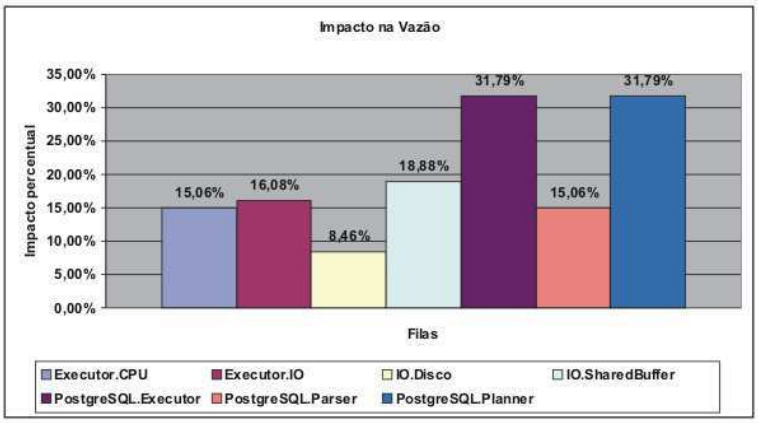

Figura 6.21: *Impacto do ajuste do framework na vazão das filas, no ambiente exigindo ordenação (com ajuste apenas do work_mem)*

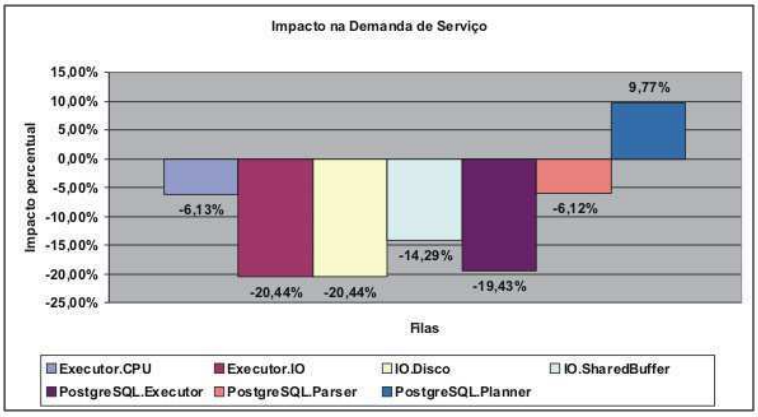

Figura 6.22: *Impacto do ajuste do framework na demanda de serviço das filas, no ambiente exigindo ordenação (com ajuste apenas do work_mem)*

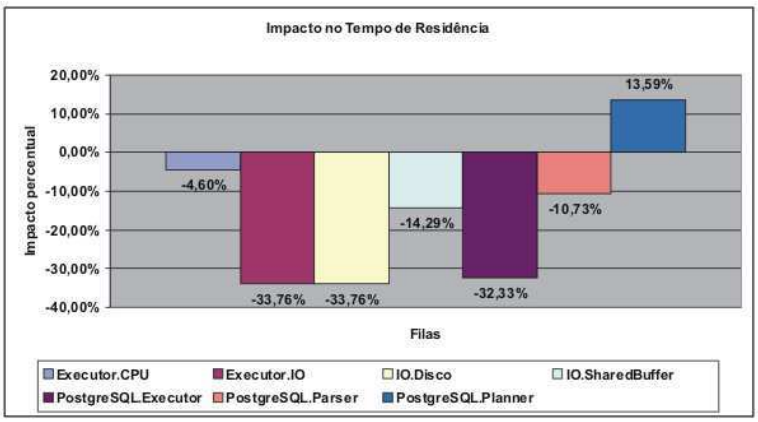

Figura 6.23: *Impacto do ajuste do framework no tempo de residência das filas, no ambiente exigindo ordenação (com ajuste apenas do work_mem)*

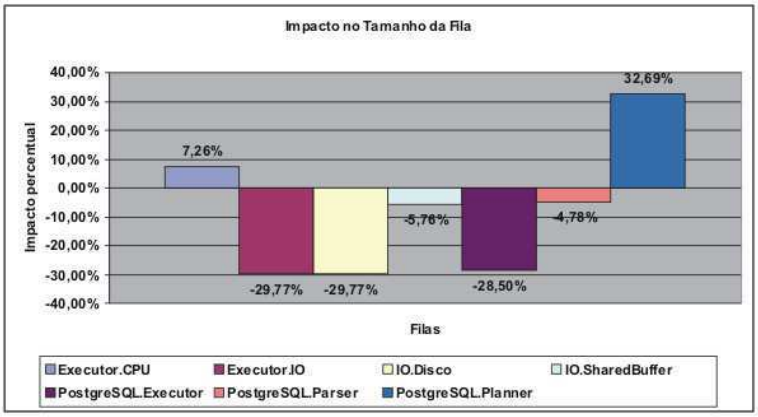

Figura 6.24: *Impacto do ajuste do framework no tamanho das filas, no ambiente exigindo ordenação (com ajuste apenas do work_mem)*

Outra situação de desempenho do SGBD é apresentada quando foram acatadas todas as sugestões do *framework*, ou seja, ajustando-se ambos os parâmetros work_mem e shared_buffer. Com as novas configurações, a situação do PostgreSQL alterou-se para a indicada na figura 6.25.

Fila	Utilizacao	Vazao	Demanda de Servico	Tempo de Residencia	Tamanho Fila
Executor.CPU	0,415745	3,870481	0,109934	0,192407	0,736482
Executor.IO	0,720701	551,924526	0,201267	0,951371	3,155240
IO.Disco	0,720680	139,356265	0,201261	0,951232	3,154793
IO.SharedBuffer	0,000210	412,568261	0,000057	0,000057	0,000210
PostgreSQL	0,000000	3,870481	0,000000	0,000000	0,000000
PostgreSQL.Executor	0,753938	442,164254	0,209908	1,127831	3,772340
PostgreSQL.Parser	0,008098	3,870481	0,002014	0,002096	0,008446
PostgreSQL.Planner	0,077500	442,135313	0,019768	0,021850	0,086205
Tempo de Resposta estimado	1,167642 ms				

Figura 6.25: *Situação de desempenho do PostgreSQL em ambiente exigindo ordenação, após todos os ajustes*

O impacto, em termos percentuais, de ambos os ajustes sugerido pelo *framework* nas filas do modelo de rede de filas elaborado para o PostgreSQL é apresentado na figura 6.26.

Os relatórios ilustrados nas figuras 6.19 e 6.26 apresentam melhorias no desempenho do PostgreSQL. É importante observar, porém, que o impacto causado apenas com o ajuste do parâmetro work_mem (figura 6.19) foi bem mais significativo que o impacto causado acatando-se todas as sugestões do *framework* (figura 6.26).

Este resultado ressalta a importância de se avaliar o tipo de carga a que o SGBD está submetido. Uma carga com prevalência de comandos que exijam ordenação necessita de recursos que auxiliam o processo de ordenação, no caso do PostgreSQL, uma maior quan-

Fila	Utilização	Vazão	Demanda de Serviço	Tempo de Residência	Tamanho da Fila
Executor.CPU	1,99%	6,23%	-1,35%	-2,84%	0,44%
Executor.IO	2,39%	11,80%	-3,49%	-9,56%	-0,25%
IO.Disco	2,39%	5,18%	-3,49%	-9,56%	-0,25%
IO.SharedBuffer	9,95%	14,23%	1,79%	1,79%	9,95%
PostgreSQL	0,00%	6,23%	0,00%	0,00%	0,00%
PostgreSQL.Executor	2,18%	0,28%	-3,58%	-8,51%	1,11%
PostgreSQL.Parser	-8,14%	6,23%	-8,04%	-13,17%	-13,13%
PostgreSQL.Planner	16,98%	0,28%	7,44%	9,70%	20,09%
Tempo de Resposta Estimado	-8,22%	ms			

Figura 6.26: *Comparação Antes - Após todos os ajustes sugeridos, no ambiente exigindo muita ordenação*

tidade de Work Memory. Também se reforça a idéia de que as ferramentas de gerência automática de memória não devem se preocupar apenas com uma única estrutura de memória de um SGBD, por exemplo, o cache de dados, que é o único foco da maioria dos trabalhos relacionados (vide Capítulo 3).

O impacto causada pelo ajuste de ambos os parâmetros work_mem e shared_buffer nas variáveis operacionais das filas são indicados nos gráficos das figuras 6.27, 6.28, 6.29, 6.30 e 6.31.

Através do gráfico da figura 6.27, nota-se que a utilização da maioria das filas do SGBD obteve um aumento pouco expressivo (em torno de 2%), com exceção das filas I/O.SharedBuffer (aumento de quase 10%), PostgreSQL.Parser (redução de 8,14%) e PostgreSQL.Planner (aumento de 16,98%).

A vazão de todas as filas aumentou, principalmente a fila I/O.SharedBuffer que trata das requisições de leituras no shared buffer cache do PostgreSQL. Essa nova configuração das filas acarretou uma redução em seus tempos de residência que contribuíram para a redução de 8,22% no tempo de resposta do sistema.

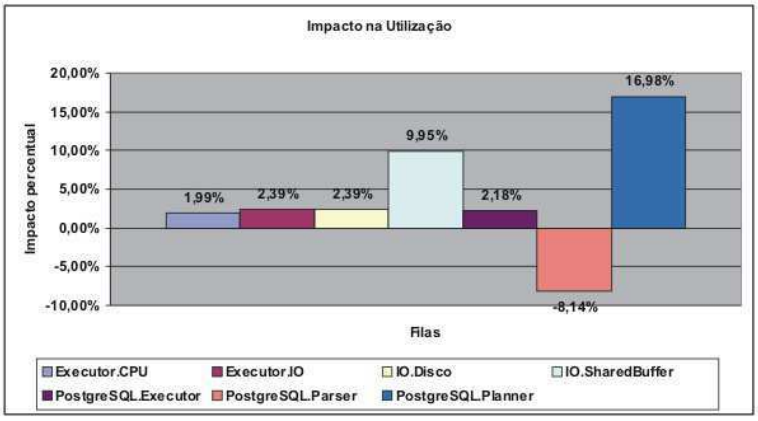

Figura 6.27: *Impacto do ajuste do framework na utilização das filas, no ambiente exigindo ordenação*

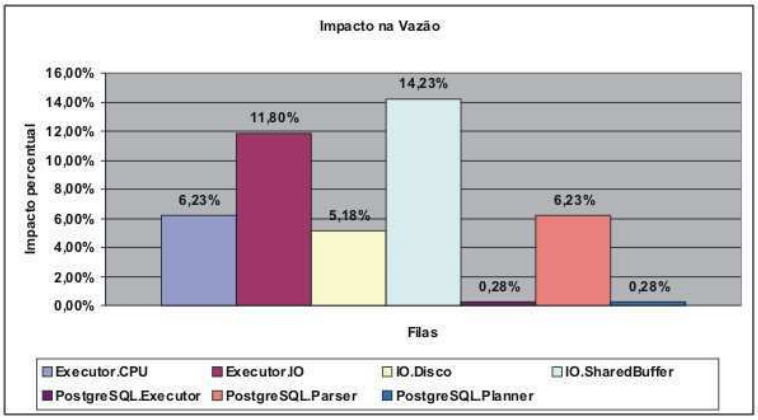

Figura 6.28: *Impacto do ajuste do framework na vazão das filas, no ambiente exigindo ordenação*

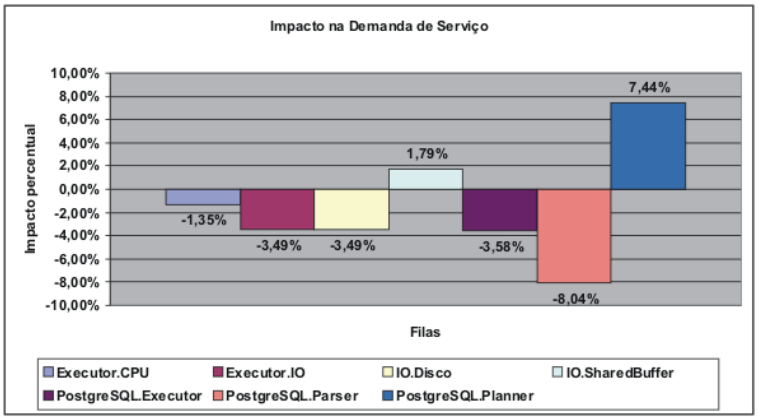

Figura 6.29: *Impacto do ajuste do framework na demanda de serviço das filas, no ambiente exigindo ordenação*

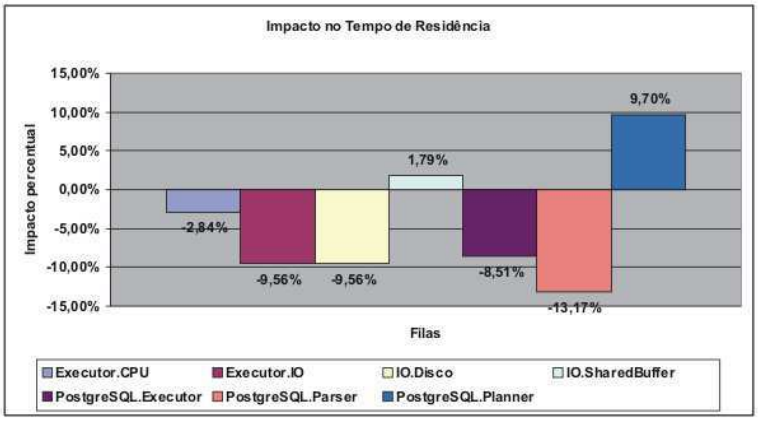

Figura 6.30: *Impacto do ajuste do framework no tempo de residência das filas, no ambiente exigindo ordenação*

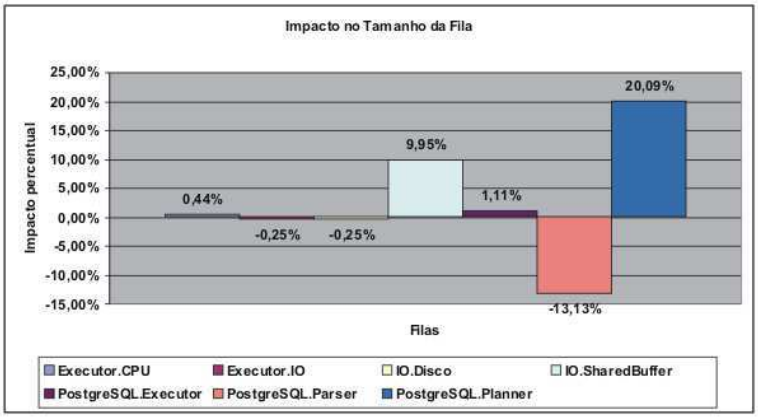

Figura 6.31: *Impacto do ajuste do framework no tamanho das filas, no ambiente exigindo ordenação*

6.3.4 Avaliação da eficácia do ajuste em um ambiente que acesse predominantemente tabelas muito grandes

Este caso de teste tem, como principal objetivo, verificar o impacto do ajuste do *framework* quando o PostgreSQL está executando em um ambiente cuja carga contém comandos SQL que acessam, predominantemente, tabelas muito grandes, tais como **stock**, **order_line**, **customer**, **history** e **oorder**.

Nenhuma dessas tabelas cabem inteiramente na memória do PC utilizado nos testes, pois todas ocupam espaços superiores a 1GB. É interessante, então, avaliar qual a configuração de memória que o *framework* sugere quando as tabelas mais acessadas na carga de comandos SQL submetida ao SGBD não cabem na memória do sistema.

A configuração do ambiente do teste é resumida a seguir:

Configuração do TPC

- composição da carga submetida

- Payment: 0%
- Order-Status: 50%
- Delivery: 50%
- Stock-Level: 0%

Configuração do PostgreSQL

- padrão

A meta de tempo de resposta utilizada para este caso de teste é de 0,9ms.

Resultado Obtido

Após a execução do caso de teste, verificou-se que o PostgreSQL apresentava-se na situação de desempenho descrita pela figura 6.32.

Fila	Utilizacao	Vazao	Demanda de Servico	Tempo de Residencia	Tamanho Fila
Executor.CPU	0,030025	16,279772	0,001848	0,001906	0,030970
Executor.IO	0,915420	1086,336152	0,056442	0,985578	15,979781
IO.Disco	0,915375	186,172930	0,056439	0,984387	15,960412
IO.SharedBuffer	0,000443	900,163222	0,000027	0,000027	0,000443
PostgreSQL	0,000000	16,279772	0,000000	0,000000	0,000000
PostgreSQL.Executor	0,918404	87,342608	0,056625	1,071966	17,388262
PostgreSQL.Parser	0,027755	16,281505	0,001700	0,002444	0,039933
PostgreSQL.Planner	0,024624	87,298684	0,001524	0,001767	0,028603
Tempo de Resposta estimado	0,990530 ms				

Figura 6.32: *Situação de desempenho do PostgreSQL no ambiente acessando tabelas grandes*

Diante desta situação, o *framework* sugeriu as seguintes configurações para os parâmetros shared_buffer e work_mem do SGBD:

- shared_buffer = 30 Mb
- work_mem = 16 Mb

Após acatadas todas as sugestões do *framework*, a situação de desempenho do PostgreSQL alterou-se para a indicada na figura 6.33.

Fila	Utilizacao	Vazao	Demanda de Servico	Tempo de Residencia	Tamanho Fila
Executor.CPU	0,026534	16,050621	0,001653	0,001699	0,027289
Executor.IO	0,938305	934,089920	0,058972	1,211847	19,439917
IO.Disco	0,938262	165,336058	0,058970	1,210368	19,416156
IO.SharedBuffer	0,000437	768,753862	0,000028	0,000028	0,000437
PostgreSQL	0,000000	16,050621	0,000000	0,000000	0,000000
PostgreSQL.Executor	0,940939	75,293586	0,059136	1,310249	21,046260
PostgreSQL.Parser	0,021281	16,051573	0,001352	0,001575	0,024710
PostgreSQL.Planner	0,026771	75,265165	0,001686	0,002663	0,042129
Tempo de Resposta estimado	1,216332 ms				

Figura 6.33: *Situação de desempenho do PostgreSQL no ambiente acessando tabelas grandes, após ajustes sugeridos*

O impacto, em termos percentuais, dos ajustes sugeridos pelo *framework* nas filas do modelo de rede de filas elaborado para o PostgreSQL é apresentado na figura 6.34.

Observa-se na figura 6.34 que houve um aumento de 22,8% no tempo de resposta do PostgreSQL após o ajuste sugerido pelo *framework*. Destaca-se aqui o fato de o *framework*

Fila	Utilização	Vazão	Demanda de Serviço	Tempo de Residência	Tamanho da Fila
Executor.CPU	-11,63%	-1,41%	-10,55%	-10,86%	-11,89%
Executor.IO	2,50%	-14,01%	4,48%	22,96%	21,65%
IO.Disco	2,50%	-11,19%	4,48%	22,96%	21,65%
IO.SharedBuffer	-1,35%	-14,60%	3,70%	3,70%	-1,35%
PostgreSQL	0,00%	-1,41%	0,00%	0,00%	0,00%
PostgreSQL.Executor	2,45%	-13,80%	4,43%	22,23%	21,04%
PostgreSQL.Parser	0,00%	-1,41%	0,00%	0,00%	0,00%
PostgreSQL.Planner	8,72%	-13,78%	10,63%	50,71%	47,29%
Tempo de Resposta Estimado	22,80%ms				

Figura 6.34: *Comparação Antes - Após ajustes, no ambiente acessando tabelas grandes*

ter sugerido uma alteração apenas no valor do parâmetro work_mem, mantendo intacto o parâmetro shared_buffer.

Este resultado evidencia a natureza holística da atividade de ajuste de desempenho em SGBDs e reforça a discussão do capítulo 3, que o tempo gasto com atividade de I/O não pode ser reduzido apenas com ajustes nas estruturas de memória de um SGBD. Tampouco, o tempo de resposta das transações submetidas ao SGBD pode ser determinado apenas pelo tempo de atividade de I/O.

O impacto do ajuste do *framework* nas variáveis operacionais das filas são indicados nos gráficos das figuras 6.35, 6.36, 6.37, 6.38 e 6.39.

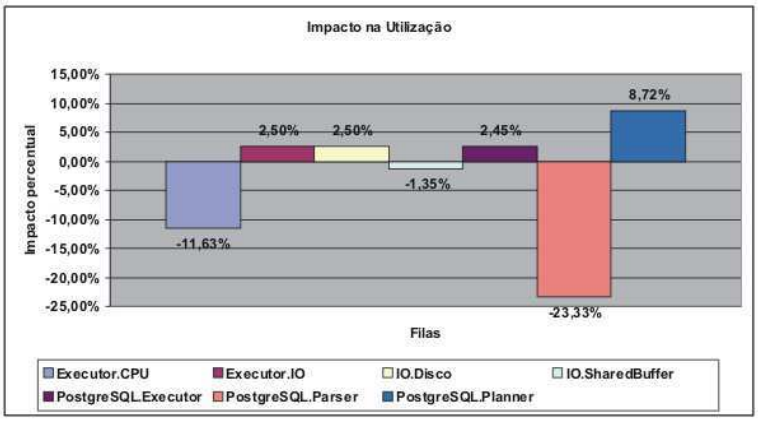

Figura 6.35: *Impacto do ajuste do framework na utilização das filas, no ambiente acessando tabelas grandes*

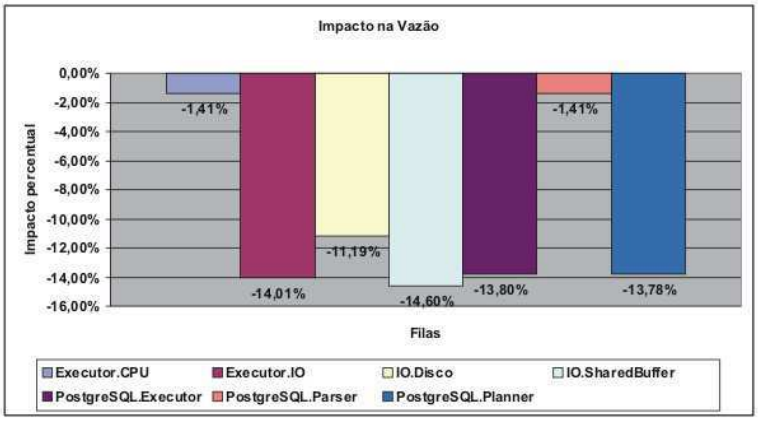

Figura 6.36: *Impacto do ajuste do framework na vazão das filas, no ambiente acessando tabelas grandes*

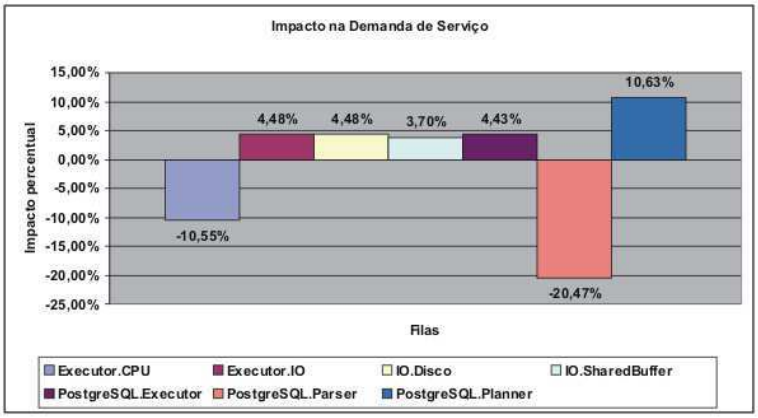

Figura 6.37: *Impacto do ajuste do framework na demanda de serviço das filas, no ambiente acessando tabelas grandes*

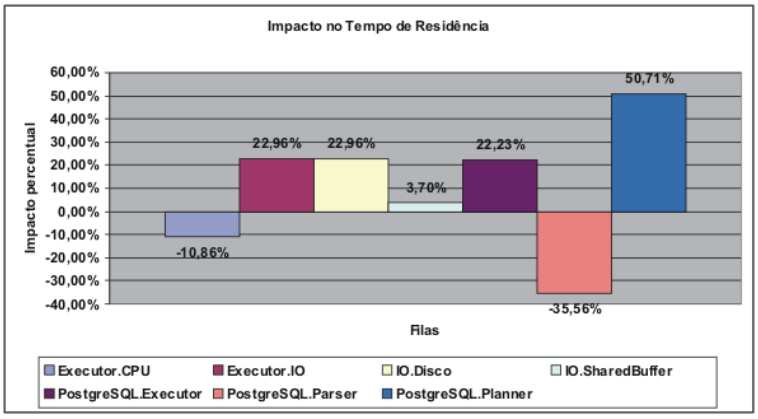

Figura 6.38: *Impacto do ajuste do framework no tempo de residência das filas, no ambiente acessando tabelas grandes*

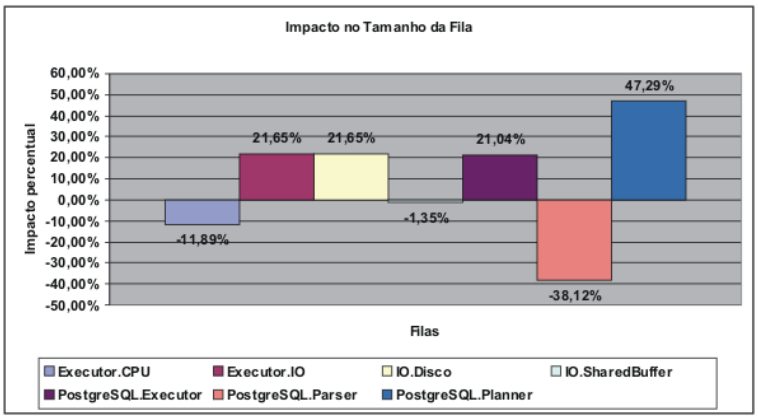

Figura 6.39: *Impacto do ajuste do framework no tamanho das filas, no ambiente acessando tabelas grandes*

Destaca-se, nos gráficos das figuras 6.35, 6.36, 6.37, 6.38 e 6.39, o fato de as vazões de todas as filas terem diminuído, no entanto, a utilização das filas se mantiveram praticamente com o mesmo valor (com exceção das filas PostgreSQL.Parser, PostgreSQL.Planner e Executor.CPU). Pode-se interpretar este fato como se o SGBD estivesse gastando o mesmo tempo (mesma utilização) para fazer menos trabalho (menor vazão). Como conseqüência, o tempo médio de resposta dos comandos aumentou.

Para verificar se um aumento maior do Shared Buffer Cache acarretaria um impacto positivo no desempenho do PostgreSQL, foi elaborada uma situação especial: retirou-se o limite máximo de 15% do total da memória RAM que o *framework* pode alocar ao Shared

Buffer (vide seção 5.3.1) e uma nova análise do *framework* foi executada. Os novos ajustes sugeridos pelo *framework* foram:

- shared_buffer = 697 Mb
- work_mem = 6 Mb

Avaliou-se a performance do sistema com a nova configuração sugerida e o resultado obtido pode ser visto na figura 6.40. O impacto, em termos percentuais, da nova configuração no desempenho do PostgreSQL pode ser visto na figura 6.41.

Fila	Utilizacao	Vazao	Demanda de Servico	Tempo de Residencia	Tamanho Fila
Executor.CPU	0,031720	16,466886	0,001891	0,001958	0,032879
Executor.IO	0,934188	732,434833	0,057802	1,313789	21,738148
IO.Disco	0,934093	67,315204	0,057796	1,310104	21,675418
IO.SharedBuffer	0,000950	665,119628	0,000058	0,000058	0,000951
PostgreSQL	0,000000	16,466886	0,000000	0,000000	0,000000
PostgreSQL.Executor	0,937343	91,079606	0,057990	1,451972	24,126633
PostgreSQL.Parser	0,009615	16,466886	0,000582	0,000592	0,009785
PostgreSQL.Planner	0,097757	91,050892	0,006145	0,003427	0,054610
Tempo de Resposta estimado	1,310754 ms				

Figura 6.40: *Situação de desempenho do PostgreSQL no ambiente acessando tabelas grandes, após ajustes sugeridos para a situação especial*

Fila	Utilização	Vazão	Demanda de Serviço	Tempo de Residência	Tamanho da Fila
Executor.CPU	5,65%	1,15%	2,33%	2,73%	6,16%
Executor.IO	2,05%	-32,58%	2,41%	33,30%	36,04%
IO.Disco	2,04%	-63,84%	2,40%	33,09%	35,81%
IO.SharedBuffer	114,45%	-26,11%	114,81%	114,81%	114,67%
PostgreSQL	0,00%	1,15%	0,00%	0,00%	0,00%
PostgreSQL.Executor	2,06%	4,28%	2,41%	35,45%	38,75%
PostgreSQL.Parser	-65,36%	1,14%	-65,76%	-75,78%	-75,50%
PostgreSQL.Planner	297,00%	4,30%	303,22%	93,94%	90,92%
Tempo de Resposta Estimado	**32,33%ms**				

Figura 6.41: *Comparação Antes - Após ajustes, na situação especial elaborada para o ambiente acessando tabelas grandes*

Observa-se que, de acordo com as figuras 6.40 e 6.41, o aumento do Shared Buffer Cache de 30Mb para 697Mb manteve o aumento no tempo de resposta dos comandos do SGBD. Isso reforça a idéia de que, nem sempre, um ajuste na memória do SGBD pode acarretar um ganho de desempenho. Logo, a análise de outros aspectos do SGBD se faz necessário. Lembramos, também, que o foco desta dissertação é avaliar a influência de diversos componentes do ambiente do SGBD PostgreSQL (parser, planner, executor, CPU, I/O) em seu desempenho, porém, com intervenções apenas na memória. A intervenção em

outros componentes do SGBD será parte de trabalhos futuros e poderá ser acrescentada ao processo de ajuste implementado nesta dissertação.

O novo impacto do ajuste do *framework* nas variáveis operacionais das filas são indicados nos gráficos das figuras 6.42, 6.43, 6.44, 6.45 e 6.46.

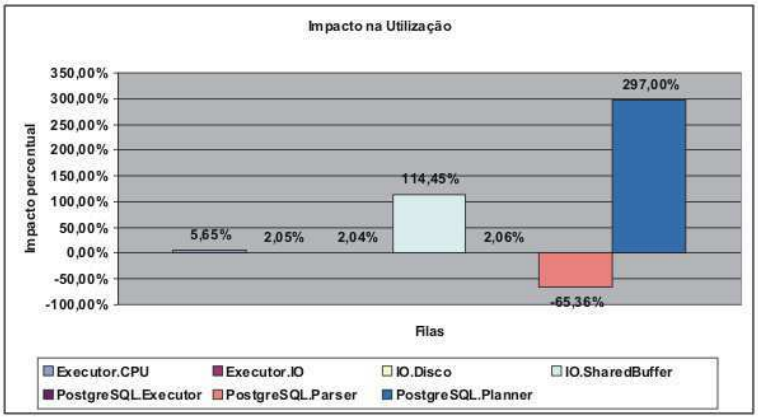

Figura 6.42: *Impacto do ajuste do framework na utilização das filas, na situação especial com ambiente acessando tabelas grandes*

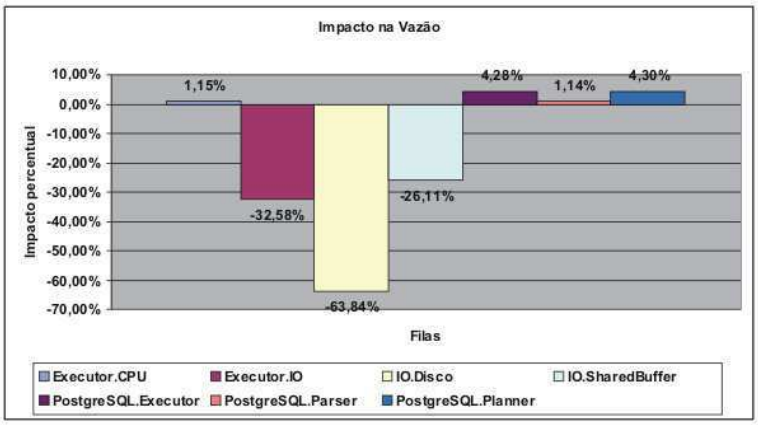

Figura 6.43: *Impacto do ajuste do framework na vazão das filas, na situação especial com ambiente acessando tabelas grandes*

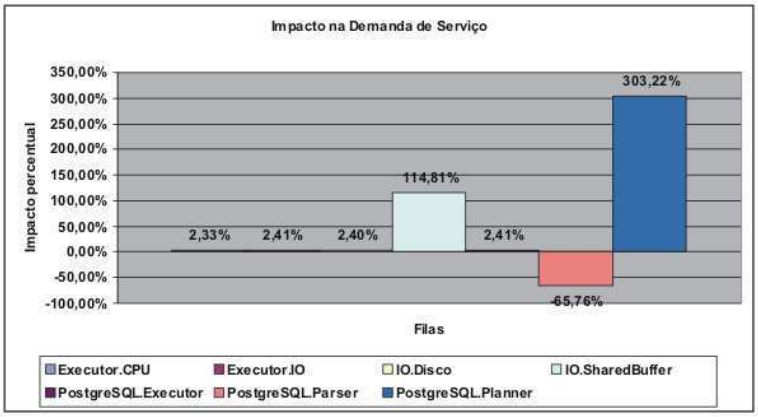

Figura 6.44: *Impacto do ajuste do framework na demanda de serviço das filas, na situação especial com ambiente acessando tabelas grandes*

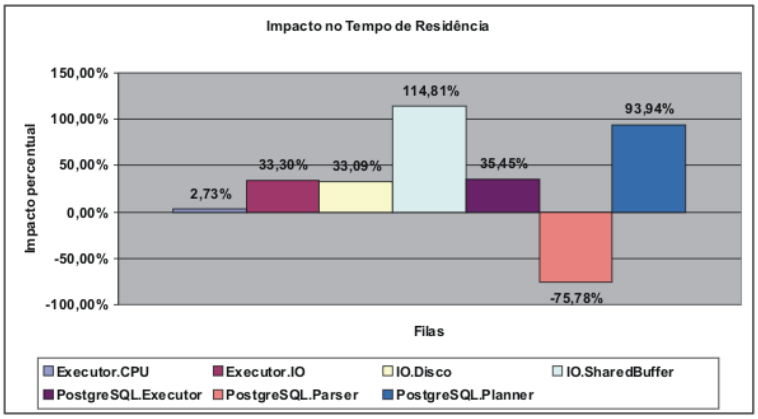

Figura 6.45: *Impacto do ajuste do framework no tempo de residência das filas, na situação especial com ambiente acessando tabelas grandes*

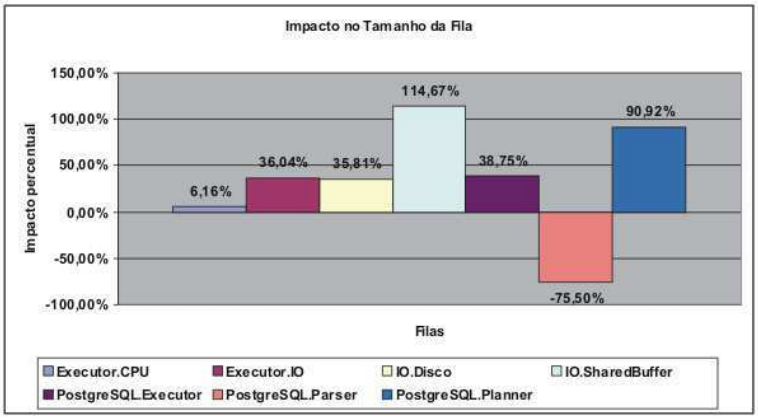

Figura 6.46: *Impacto do ajuste do framework no tamanho das filas, na situação especial com ambiente acessando tabelas grandes*

Verifica-se, a partir dos gráficos das figuras 6.42, 6.43, 6.44, 6.45 e 6.46, que a situação anterior se manteve. Ou seja, a utilização das filas mantiveram-se praticamente com o mesmo valor (com exceção das filas IO.SharedBuffer e PostgreSQL.Planner), porém, houve uma drástica diminuição na vazão das filas. Como conseqüência, o tempo de resposta médio dos comandos aumentou.

Destaca-se ainda o tremendo aumento ocorrido na utilização da fila PostgreSQL.Planner com sua vazão mantida constante. Esse fato pode ser interpretado como se o Planner estivesse gastando muito mais tempo (maior utilização) para fazer o mesmo trabalho (mesma vazão).

6.4 Discussão

Diante dos resultados obtidos nos casos de testes a que o *framework* foi submetido, é importante tecer alguns comentários sobre os diferentes aspectos do *framework*. De modo geral, levando em consideração os resultados obtidos nas seções 6.3.1, 6.3.2 e 6.3.3, pode-se afirmar que o *framework* e a extensão elaborada para o PostgreSQL apresentaram resultados satisfatórios, pois reduziu em pelo menos 16,17% o tempo de resposta do PostgreSQL em 3 dos 4 casos de teste a que o *framework* foi submetido e as metas de tempo de resposta foram restabelecidas.

Por outro lado, levando em conta o resultado obtido na seção 6.3.4, vê-se que a es-

tratégia de ajuste desenvolvida pode ser melhorada. Uma análise mais profunda do caso de teste da seção 6.3.4 revela um aspecto interessante da estratégia de dimensionamento do shared_buffer implementada no *framework*. A carga de comandos SQL do referido caso de teste acessa, predominantemente, tabelas muito grandes, tais como stock, order_line, customer, history e oorder; com tamanhos em torno de 3862Mb, 3287Mb, 1972Mb, 270Mb, 242Mb, respectivamente (vide apêndice A). Os índices dessas tabelas também ocupam espaços igualmente grandes.

Com as informações dos tamanhos das tabelas e levando em conta que a estratégia de dimensionamento do shared_buffer não aloca espaço de memória superior a 15% da RAM total do sistema (vide Capítulo 5), conclui-se que haverá situações onde o *framework* não será capaz de sugerir nenhum ajuste para o shared_buffer. Essa situação ocorre devido ao fato de nenhuma tabela envolvida na carga de comando SQL submetida caber, totalmente, em 15% da RAM total do sistema. Este cenário ocorreu na situação do caso de teste da seção 6.3.4, logo, o *framework* não sugeriu mudanças no parâmetro shared_buffer.

Se por um lado pode-se argumentar que esse comportamento do *framework* é falho, devido ao fato de ele não reagir na situação descrita, por outro, pode-se argumentar que ele não desperdiça memória. Esse argumento pode ser defendido da seguinte forma: apesar de o *framework* não sugerir aumento do shared_buffer, não se pode afirmar que haverá pouca memória para armazenar as tabelas acessadas, pois o *framework* tem ciência da presença do *cache* do Unix, que irá efetuar o *cache* dos dados das tabelas (vide Capítulo 2, seção 2.6.3). Além disso, diante de um conjunto de tabelas que não caibam na memória, cenário do caso de teste da seção 6.3.4, possivelmente, mesmo uma grande alocação de memória para o shared_buffer não irá afetar de forma relevante o desempenho do SGBD. De fato, essa situação foi comprovada com o resultado obtido na seção 6.3.4, quando uma configuração especial alocou 697Mb ao shared_buffer e o tempo de resposta aumentou em 32,33% (vide figura 6.41), impacto pior do que o obtido quando o *framework* não alterou o shard_buffer (vide figura 6.34).

Dessa forma, ao não alterar o valor do shared_buffer, o *framework* sugere que o problema de desempenho não poderá ser amenizado, de forma relevante, com uma intervenção na memória. Esse fato reforça a idéia de que, nem sempre, um problema de desempenho pode ser resolvido apenas ajustando-se os buffers de memória de um SGBD. Surge-se então a

necessidade de se avaliar outros aspectos do SGBD na hora de implementar algum ajuste – tais como a possibilidade de criação de um índice – e não só a memória RAM utilizada.

Com relação ao diagnóstico de problemas de desempenho, pode-se dizer que o *framework* se comportou de forma satisfatória, uma vez que o seu diagnóstico indicou corretamente as filas que estavam consumindo o maior tempo de processamento do SGBD. Além disso, o diagnóstico levou em conta, não só o *hit rate* do *cache* de dados ou tempo médio de acesso a dados – como considerado na maioria dos trabalhos relacionados – mas todas as principais etapas do processo de execução de um comando SQL no PostgreSQL (*Parser*, *Planner* e *Executor*) bem como os recursos de hardware utilizados na etapa Executor (CPU e I/O).

Com essa visão global do SGBD, o diagnóstico de um problema de desempenho torna-se mais preciso, uma vez que os componentes cruciais do SGBD são analisados antes de ser indicado qual deles está causando o maior impacto negativo no desempenho do sistema (o gargalo), o que dificilmente seria possível indicar analisando-se apenas o *hit rate* do *cache* de dados ou o tempo médio de acesso a dados.

Capítulo 7

Conclusão

Neste capítulo são apresentadas as principais contribuições do *framework* implementado nesta dissertação e algumas opções de trabalhos futuros para melhorar o *framework*. Na seção 7.1 são apresentadas as principais contribuições do *framework*, enquanto na seção 7.2 algumas opções de trabalhos futuros são sugeridas.

7.1 Principais Contribuições

A principal contribuição desta dissertação é desenvolvimento de um *framework* de gerência de desempenho de SGBDs baseada na análise operacional. O *framework* implementa o ciclo básico de gerência da computação autônoma e é capaz de coletar métricas dos SGBDs gerenciados; calcular as variáveis operacionais das filas de uma rede de filas, a partir dos valores das métricas coletadas; efetuar uma análise de desempenho do SGBD com base na rede de filas definida para ele; diagnosticar qual fila está consumindo o maior tempo de processamento do SGBD (o gargalo do sistema) e, se necessário, desencadear um processo de ajuste para que o desempenho satisfatório do SGBD seja restabelecido.

O *framework* é capaz de detectar, com uma análise holística, o componente responsável pelo maior impacto na degradação de desempenho do sistema e invocar um ajustador para este componente, contribuindo para o bom desempenho do SGBD.

Outra contribuição foi a instanciação do *framework* para o SGBD PostgreSQL, onde foi definida uma rede de filas para servir de base para o diagnóstico holístico do *framework*, bem como a implementação de um ajustador para a fila que representa a atividade de disco

do SGBD. O ajustador visou a reduzir a demanda de serviço do disco magnético ajustando as estruturas de memória do PostgreSQL (shared buffer e work memory). Outras contribuições desta dissertação são apresentadas na lista a seguir:

Extensibilidade do *framework* Permitir que o *framework* desenvolvido possa ser estendido para gerenciar o desempenho de diferentes SGBDs. Como discutido na seção 4.3, para estender o *framework*, é preciso que o desenvolvedor: defina um modelo de rede de filas para o SGBD a ser gerenciado; implemente, ou reuse, componentes de acesso para coletar métricas desse SGBD; defina o mapeamento entre as métricas coletadas e as variáveis operacionais das filas do modelo de rede de filas definido e implemente suas estratégias de ajuste automático para cada desse modelo.

Implementação em um SGBD livre Este trabalho contribuiu para o aumento do acervo de ferramentas de *self-tuning databases* voltadas para SGBDs livres, tais como o PostgreSQL. Viu-se, no Capítulo 3, que poucos trabalhos de gerência automática de memória são voltados para SGBDs livres, destacando assim a carência de ferramentas de *self-tuning databases* para esses SGBDs.

Diagnóstico holístico No Capítulo 3, também foi destacado um aspecto das ferramentas de gerência automática de memória relacionado ao processo de detecção de problemas de desempenho em SGBDs. A maioria dos trabalhos analisa apenas o DAT médio ou o *hit rate* do *cache* de dados para decidir se existe ou não um problema de desempenho. Este trabalho foi muito além e considerou os tempos gastos em cada uma das etapas de execução dos comandos submetidos a um SGBD, bem como o tempo gasto com CPU e atividades de disco para decidir se o SGBD está com um problema de desempenho.

No Capítulo 3 foi feito um panorama da situação dos trabalhos relacionados à gerência automática de memória em SGBDs, destacando-se alguns requisitos desejados a uma boa ferramenta de gerência automática de memória e quais dos trabalhos relacionados atendiam a esses requisitos. O quadro apresentado naquele Capítulo é repetido aqui, acrescentando-se as características do *framework* desenvolvido nesta dissertação.

Requisitos

A. **Considerar não só o tempo médio de I/O ou o *hit rate* de *buffers* como fatores determinantes do tempo de resposta das transações em SGBDs**. Existem outras etapas do processamento de transações que também contribuem para o tempo de resposta das transações, tais como compilação, criação do plano de consulta, execução de algoritmos de ordenação, de junção e de funções diversas, tais como funções matemáticas.

B. **Avaliar outros aspectos do SGBD, e não apenas sua memória, para detectar um problema de desempenho**. Durante a fase de avaliação de desempenho do SGBD, é preciso avaliar a situação de outros componentes, tais como o *parser*, o *planner*, a CPU. Afirmar que o SGBD está com um baixo desempenho analisando apenas o *hit rate* de seus *buffers* de memória ou o DAT médio pode levar a uma avaliação imprecisa do desempenho do SGBD.

C. **Extensibilidade**. O gerente automático de memória deve ser extensível para utilização com outros SGBDs. Essa característica é de fundamental importância para que uma boa solução possa ser reusada ou estendida para diferentes SGBDs.

D. **Implementação em SGBDs livres**. A grande maioria dos componentes de gerência de memória são implementados em SGBDs comerciais.

E. **Ajustes em várias estruturas de memória**. O gerente de memória deve ajustar várias estruturas de memória do SGBD, tais como *cache* de dados, área de ordenação e hashing, dentre outras; e não só o *cache* de dados, como faz a maioria dos trabalhos.

Comparação entre os Trabalhos

Trabalho	**Requisitos Desejados**				
	A	B	C	D	E
[MPXT06]					
[BCL93]					X
[SGAL+06]	X				X
[WPT05]				X	
[BCL96]					
[MPLR02]					
DBMS-Analyzer	X	X	X	X	X

Tabela 7.1: Comparação entre os trabalhos relacionados e o *framework* desenvolvido

Na tabela 7.1, as células marcadas com um 'X' indicam que o trabalho da linha correspondente se preocupa com o aspecto da coluna correspondente. Nota-se que a única ferramenta que atende aos requisitos B e C é a desenvolvida nesta dissertação.

7.2 Trabalhos Futuros

Durante o desenvolvimento do *framework* implementado nesta dissertação, surgiram várias questões e idéias interessantes a serem incorporadas. Tais questões e idéias vão desde a melhoria e correções no *framework* a novas funcionalidades. A seguir, é apresentado um conjunto de trabalhos futuros que não foram implementados no *framework* por falta de tempo ou por não fazerem parte do escopo deste trabalho:

- É interessante implementar novos componentes de acesso para o *framework* para facilitar sua extensibilidade a novos SGBDs. Componentes de acesso capazes de coletar métricas utilizando o protocolo SNMP ou JDBC seria de grande valia para o *framework* e facilitaria sua extensão para novos SGBDs, pois pouparia o trabalho dos desenvolvedores que poderiam reusar tais componentes de acesso.

- Poderia ser implantada uma melhoria no componente que efetua o mapeamento das métricas coletadas em variáveis operacionais. O desenvolvedor poderia definir o cálculo das variáveis operacionais em uma linguagem de especificação de função matemática, por exemplo, MathML [W3C08] ou uma linguagem proprietária com uma sintaxe mais simples. O *framework* então se encarregaria de ler a definição da função e efetuar o cálculo da variável operacional. Isso pouparia o trabalho do desenvolvedor de criar uma classe Java para cada variável operacional de cada fila do modelo de rede de filas definido. Em vez de informar o nome da classe Java, o desenvolvedor informaria a função para calcular a variável operacional.

- Com relação ao componente responsável por efetuar os ajustes nos SGBDs, seria interessante que as sugestões do *framework* fossem salvas em um banco de dados para a formação de um histórico. Esse histórico poderia servir de base de informação para algum mecanismo de mineração de dados que influenciasse as decisões futuras do *framework*.

- Também é interessante implementar componentes de execução dos comandos sugeridos pelo *framework* que utilizem protocolos tais como SNMP ou JDBC. Esses componentes, uma vez implementados, poderiam ser reusados pelos desenvolvedores para auxiliar na extensão do *framework* para novos SGBDs.

- Extensões do *framework* para gerenciar outros SGBDs, tais como MySQL, que também é pouco contemplado com ferramentas de *self-tuning databases*, também seriam de grande valia para os usuários desses SGBDs. Para isso, seriam necessários a definição de modelos de rede de filas que representassem os recursos utilizados por esses SGBDs, a definição de um conjunto de métricas que precisariam ser coletadas, o modo como, a partir dessas métricas, as variáveis operacionais das filas seriam calculadas e as estratégias de ajustes de cada fila do modelo.

- Para melhorar a extensão do *framework* para o PostgreSQL, desenvolvida nesta dissertação, seria interessante acrescentar ao processo de ajuste desenvolvido uma etapa que incorporasse aspectos da gerência automática de índices. Assim, além de tomar decisões acerca de intervenções na memória do PostgreSQL, a aplicação também seria capaz de sugerir, automaticamente, a criação ou remoção de índices.

- Além da gerência automática de índices e memória, melhorias no processo de ajuste desenvolvido nesta dissertação poderiam considerar também outros ajustes do SGBD, tais como a gerência de bloqueios, partições de tabelas, dentre outros. Com isso, cada vez mais o processo de ajuste efetuaria uma gerência holística de desempenho do PostgreSQL.

- Outra idéia interessante, que contribuiria bastante para gerência automática de desempenho de SGBDs, principalmente daqueles que carecem de ferramentas de *self-tuning databases* (a exemplo do PostgreSQL e MySQL), seria o compartilhamento dos modelos de performance definidos para esses SGBDs. Os modelos de performance poderiam ser coletivamente definidos e evoluídos por grupos de usuários na Internet. Por exemplo, os usuários do MySQL espalhados em todo o planeta definiriam um modelo de performance para gerenciar automaticamente o desempenho desse SGBD. Com as sugestões e pontos de vistas de milhares de usuários, o modelo de performance

definido tornaria-se bastante eficaz e padronizado entre seus usuários. Os usuários poderiam atribuir nomes e versões para cada modelo de performance definido.

- A mesma idéia do item anterior poderia ser implementada para os processos de ajustes implementados para os modelos de performance. Usuários poderiam implementar e evoluir processos de ajustes para os modelos de performance compartilhados no item anterior. Assim, os usuários poderiam testar e avaliar diferentes processos de ajustes para seus SGBDs e aperfeiçoá-los, o que contribuiria para a formação de um grande ferramental de gerência automática de SGBDs.

Referências Bibliográficas

[ACK+04] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. Database tuning advisor for microsoft sql server 2005. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *Proceedings of the VLDB 2004 Conference*, pages 1110–1121. Morgan Kaufmann, 2004.

[BCL93] Kurt P. Brown, Michael J. Carey, and Miron Livny. Managing memory to meet multiclass workload response time goals. In *VLDB '93: Proceedings of the 19th International Conference on Very Large Data Bases*, pages 328–341, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[BCL96] Kurt P. Brown, Michael J. Carey, and Miron Livny. Goal-oriented buffer management revisited. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 353–364, New York, NY, USA, 1996. ACM.

[Bel66] Laszlo A. Belady. A study of replacement algorithms for virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[Ben05] Darcy G. Benoit. Automatic diagnosis of performance problems in database management systems. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, pages 326–327, Washington, DC, USA, 2005. IEEE Computer Society.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford

Stein. *Introduction to Algorithms, Second Edition*. McGraw-Hill Science/Engineering/Math, July 2001.

[dCCLdNS05] Rogério Luís de Carvalho Costa, Sérgio Lifschitz, Maira Ferreira de Noronha, and Marcos Antonio Vaz Salles. Implementation of an agent architecture for automated index tuning. In *ICDE Workshops*, page 1215, 2005.

[DD06] Benoit Dageville and Karl Dias. Oracle's self-tuning architecture and solutions. *IEEE Computer Society: Bulletin of the Technical Committee on Data Engineering*, 29(3):24–31, September 2006.

[DDD+04] Benoit Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zait, and Mohamed Ziauddin. Automatic sql tuning in oracle 10g. In *vldb'2004: Proceedings of the Thirtieth international conference on Very large data bases*, pages 1098–1109. VLDB Endowment, 2004.

[DRS+05] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood. Automatic performance diagnosis and tuning in oracle. In *CIDR*, pages 84–94, 2005.

[Gro06] The PostgreSQL Global Development Group. Postgresql 8.2.5 documentation. http://www.postgresql.org/docs/8.2/interactive/index.html, 2006.

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[LSZK03] Sam Lightstone, Berni Schiefer, Danny Zilio, and Jim Kleewein. Autonomic computing for relational databases: the ten-year vision. In *AUCOPA '03: Proceedings of the IEEE Workshop Autonomic Computing Principles and Architectures*, pages 419–424, 2003.

[Lus06] Denis Lussier. Benchmarksql. http://sourceforge.net/projects/benchmarksql, 2006.

[MDA04] Daniel A. Menasce, Lawrence W. Dowdy, and Virgilio A. F. Almeida. *Performance by Design: Computer Capacity Planning By Example*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[Mom01] Bruce Momjian. Postgresql performance tuning. *Linux J.*, 2001(88):3, 2001.

[MPLR02] Patrick Martin, Wendy Powley, Hoi-Ying Li, and Keri Romanufa. Managing database server performance to meet qos requirements in electronic commerce systems. *Int. J. on Digital Libraries*, 3(4):316–324, 2002.

[MPXT06] Patrick Martin, Wendy Powley, Xiaoyi Xu, and Wenhu Tian. Automated configuration of multiple buffer pools. *Comput. J.*, 49(4):487–499, 2006.

[Mul02] Craig S. Mullins. *Database Administration: The Complete Guide to Practices and Procedures*. Addison-Wesley Professional, 2002.

[Sal04] Marcos Antonio Vaz Salles. Autonomic index creation in databases (in portuguese). Master's thesis, Computer Science Department, PUC-Rio, 2004.

[SAMP06] Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. Colt: continuous on-line tuning. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 793–795, New York, NY, USA, 2006. ACM.

[SGAL+06] Adam J. Storm, Christian Garcia-Arellano, Sam S. Lightstone, Yixin Diao, and M. Surendra. Adaptive self-tuning memory in db2. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 1081–1092. VLDB Endowment, 2006.

[SGS03] Kai-Uwe Sattler, Ingolf Geist, and Eike Schallehn. Quiet: continuous query-driven index tuning. In *vldb'2003: Proceedings of the 29th international conference on Very large data bases*, pages 1129–1132. VLDB Endowment, 2003.

[SPTU05] Roy Sterritt, Manish Parashar, Huaglory Tianfield, and Rainer Unland. A concise introduction to autonomic computing. *Advanced Engineering Informatics*, 19(3):181–187, 2005.

[SSG05] Kai-Uwe Sattler, Eike Schallehn, and Ingolf Geist. Towards indexing schemes for self-tuning dbms. In *ICDEW '05: Proceedings of the 21st*

International Conference on Data Engineering Workshops, page 1216, Washington, DC, USA, 2005. IEEE Computer Society.

[TAN08] Andrew S. TANENBAUM. *Modern Operating Systems*. Prentice Hall, 2008.

[(TP07] Transaction Processing Performance Council (TPC). Tpc benchmark c. www.tpc.org, 2007.

[Vir08] Openlink Virtuoso. Tpc c benchmark kit. http://docs.openlinksw.com/virtuoso/tpcc.html, 2008.

[W3C08] W3C. W3c math home. http://www.w3.org/Math/, 2008.

[WMHZ02] Gerhard Weikum, Axel Moenkeberg, Christof Hasse, and Peter Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 20–31. VLDB Endowment, 2002.

[WPT05] Nailah Ogeer Wendy Powley, Pat Martin and Wenhu Tian. Autonomic buffer pool configuration in postgresql. In *2005 IEEE International Conference on Systems, Man and Cybernetics*, pages 53–58, 2005.

[XMP01] Yongli Xi, Patrick Martin, and Wendy Powley. An analytical model for buffer hit rate prediction. In *CASCON '01: Proceedings of the 2001 conference of the Centre for Advanced Studies on Collaborative research*, page 18. IBM Press, 2001.

Apêndice A

Ambiente de Teste do Framework

Neste apêndice você encontrará algumas informações sobre o banco de dados utilizado nos testes do framework, tais como: esquema de tabelas do TPC-C; tamanho das tabelas e dos índices de tabelas; transações utilizadas.

O esquema de tabelas do TPC-C pode ser visto na figura A.1. A tabela *warehouse* determina o número de registros que serão inseridos em cada tabela do banco de dados e, conseqüentemente, o espaço em disco ocupado pelo banco de dados. Para os testes do framework foram utilizados 100 warehouses, criando um banco de dados com aproximadamente 10Gb de espaço ocupado em disco.

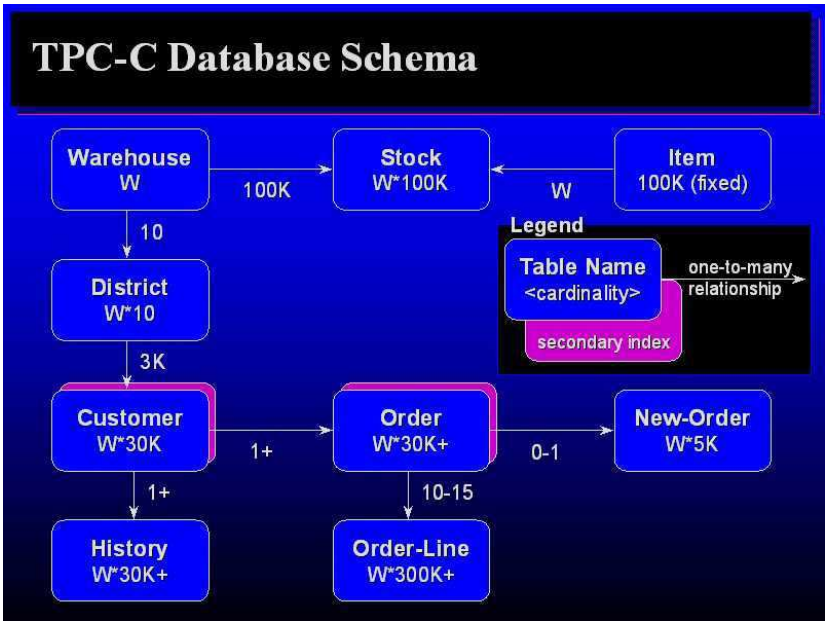

Figura A.1: *Esquema de tabelas do TPC-C Benchmark (fonte: http://www.tpc.org/)*

O tamanho das tabelas e de seus índices são apresentados a seguir:

Tamanho das Tabelas

- customer: 1972 Mb
- district: 0.64 Mb
- history: 270 Mb
- item: 11 Mb
- new_order: 38 Mb
- oorder: 242 Mb
- order_line: 3287 Mb
- stock: 3862 Mb
- warehouse: 0.242 Mb

Tamanho dos Índices

- pk_warehouse: 0.015 Mb
- pk_district: 0.054 Mb
- ndx_customer_name: 175 Mb
- pk_oorder: 77 Mb
- ndx_oorder_carrier: 90 Mb
- pk_new_order: 23 Mb
- pk_order_line: 907 Mb
- pk_stock: 214 Mb
- pk_item: 1.72 Mb

O TPC-C define cinco transações que podem ser combinadas para formar a carga a ser submetida a um SGBD. Os comandos SQL de algumas dessas transações foram extraídos do log do SGBD PostgreSQL e transcritos a seguir:

Transação Payment

```
UPDATE warehouse SET w_ytd = w_ytd + $1  WHERE w_id = $2

SELECT w_street_1, w_street_2, w_city, w_state, w_zip, w_name
FROM warehouse
WHERE w_id = $1

UPDATE district
SET d_ytd = d_ytd + $1
WHERE d_w_id = $2 AND d_id = $3

SELECT d_street_1, d_street_2, d_city, d_state, d_zip, d_name
FROM district
WHERE d_w_id = $1 AND d_id = $2

SELECT c_first, c_middle, c_last, c_street_1, c_street_2,
c_city, c_state, c_zip, c_phone, c_credit, c_credit_lim,
c_discount, c_balance, c_since
FROM customer
WHERE c_w_id = $1 AND c_d_id = $2 AND c_id = $3

UPDATE customer
SET c_balance = $1
WHERE c_w_id = $2 AND c_d_id = $3 AND c_id = $4

INSERT INTO history (h_c_d_id, h_c_w_id, h_c_id, h_d_id,
h_w_id, h_date, h_amount, h_data)
VALUES ($1,$2,$3,$4,$5,$6,$7,$8)
```

Transação Order-Status

```
SELECT count(*) AS namecnt
FROM customer
WHERE c_last = $1 AND c_d_id = $2 AND c_w_id = $3

SELECT c_balance, c_first, c_middle, c_id
FROM customer
WHERE c_last = $1 AND c_d_id = $2 AND c_w_id = $3
ORDER BY c_w_id, c_d_id, c_last, c_first

SELECT MAX(o_id) AS maxorderid
FROM oorder
WHERE o_w_id = $1 AND o_d_id = $2 AND o_c_id = $3

SELECT o_carrier_id, o_entry_d
FROM oorder
WHERE o_w_id = $1 AND o_d_id = $2
AND o_c_id = $3 AND o_id = $4

SELECT ol_i_id, ol_supply_w_id, ol_quantity, ol_amount,
ol_delivery_d
FROM order_line
WHERE ol_o_id = $1 AND ol_d_id =$2 AND ol_w_id = $3
```

Transação Delivery

```
SELECT no_o_id
FROM new_order
WHERE no_d_id = $1 AND no_w_id = $2
ORDER BY no_o_id ASC

DELETE FROM new_order
```

```
WHERE no_d_id = $1 AND no_w_id = $2 AND no_o_id = $3

SELECT o_c_id
FROM oorder
WHERE o_id = $1 AND o_d_id = $2 AND o_w_id = $3

UPDATE oorder
SET o_carrier_id = $1
WHERE o_id = $2 AND o_d_id = $3 AND o_w_id = $4

UPDATE order_line
SET ol_delivery_d = $1
WHERE ol_o_id = $2 AND ol_d_id = $3 AND ol_w_id = $4

SELECT SUM(ol_amount) AS ol_total
FROM order_line
WHERE ol_o_id = $1 AND ol_d_id = $2 AND ol_w_id = $3

UPDATE customer
SET c_balance = c_balance + $1,
c_delivery_cnt = c_delivery_cnt + 1
WHERE c_id = $2 AND c_d_id = $3 AND c_w_id = $4
```

Transação Stock-Level

```
SELECT d_next_o_id
FROM district
WHERE d_w_id = $1 AND d_id = $2

SELECT COUNT(DISTINCT (s_i_id)) AS stock_count
FROM order_line, stock
WHERE ol_w_id = $1 AND ol_d_id = $2 AND ol_o_id < $3
AND ol_o_id >= $4 - 20  AND s_w_id = $5
```

```
AND s_i_id = ol_i_id AND s_quantity < $6
```

Uma implementação detalhada das cinco transações do TPC-C em *stored procedures* pode ser vista em [Vir08].