View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositorio da Producao Cientifica e Intelectual da Unicamp

Avaliação de um Sistema de Gerência de Banco de Dados em Memória Principal para uso em aplicações WEB

Anderson Supriano

Trabalho Final de Mestrado Profissional em Computação

Instituto de Computação
Universidade Estadual de Campinas

Avaliação de um Sistema de Gerência de Banco de Dados em Memória Principal para uso em aplicações WEB

Anderson Supriano

31 de Julho de 2006

Banca Examinadora:

- Prof. Dr. Luiz Eduardo Buzato
  IC – UNICAMP (Orientador)

- Prof. Dr. Alcides Calsarara
  PUC - PR

- Prof. Dr. Célio Cardoso Guimarães
  IC - UNICAMP

- Prof. Dra. Maria Beatriz F. de Toledo
  IC – UNICAMP (Suplente)

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Miriam Cristina Alves – CRB8a / 859

	Supriano, Anderson.
Su766a	Avaliação de um sistema de gerência de banco de dados em memória principal para uso em aplicações WEB / Anderson Supriano -- Campinas, [S.P.: s.n.], 2006.
	Orientador: Luiz Eduardo Buzato.
	Trabalho final (mestrado profissional) - Universidade Estadual de Campinas, Instituto de Computação.
	1. Banco de dados – Avaliação. 2. Desempenho - Avaliação. 3. Serviços Web. 4. Processamento distribuído. I. Buzato, Luiz Eduardo. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Evaluation of a main-memory database for use on web applications.

Palavras-chave em inglês (Keywords): 1. Databases evaluation. 2. Performance evaluation. 3. Web services. 4. Distributed processing.

Área de concentração: Engenharia da Computação

Titulação: Mestre em Ciência da Computação

Banca examinadora: Prof. Dr. Luiz Eduardo Buzato (IC-UNICAMP)
Prof. Dr. Alcides Calsarara (PUC-PR)
Prof. Dr. Célio Cardoso Guimarães (IC-UNICAMP)
Profa. Dra. Maria Beatriz F. de Toledo (IC-UNICAMP)

Data da defesa: 31/07/2006

TERMO DE APROVAÇÃO

Trabalho Final Escrito defendido e aprovado em 31 de julho de 2006, pela Banca Examinadora composta pelos Professores Doutores:

Prof. Dr. Alcides Calsavara
PUC/ PR

Prof^a. Dr^a. Maria Beatriz Felgar de Toledo
IC - UNICAMP

Prof. Dr. Luiz Eduardo Buzato
IC - UNICAMP

Avaliação de um Sistema de Gerência de Banco de Dados em Memória Principal para uso em aplicações WEB

Este exemplar corresponde à redação final do Trabalho Final devidamente corrigido e defendido por Anderson Supriano e aprovado pela Banca Examinadora.

Campinas, 31 de Julho de 2006.

Prof. Dr. Luiz Eduardo Buzato
IC – UNICAMP (Orientador)

Trabalho Final apresentado ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Computação na área de Engenharia de Computação.

© Anderson Supriano, 2006

Todos os direitos reservados

Agradecimentos

Gostaria de agradecer à Paula, por estar em todos os momentos ao meu lado e sempre me dar o apoio e a motivação necessários para eu ter este trabalho concluído.

Agradeço também à minha família por sempre me apoiar e pela paciência nos momentos em que não pude estar próximo.

Agradeço ao Buzato, meu orientador, por me guiar nas pesquisas e me mostrar o caminho para a elaboração de um trabalho científico.

E agradeço também ao Gustavo, pelo suporte dado e por disponibilizar o ambiente computacional necessário à execução dos experimentos.

E por fim, agradeço aos meus superiores na Motorola que me incentivaram e deram condições para este trabalho pudesse ser realizado.

Resumo

Aplicações *web* são cada vez mais comuns em nosso cotidiano e com isto torna-se necessária a busca por soluções para a melhora do desempenho no acesso a essas aplicações. Várias técnicas existem para esta melhora de desempenho, entre elas a replicação de aplicações e bancos de dados e o uso de bancos de dados em memória principal.

Em busca da melhora de desempenho pensa-se em juntar um banco de dados de memória principal com as técnicas de replicação. Para isto, é necessário escolher um banco de dados de memória principal que seja estável e já tenha bom desempenho, para que a camada de replicação possa ser implementada utilizando-o como base.

Este trabalho tem o objetivo de analisar o desempenho de um banco de dados de memória principal e compará-lo com o desempenho de dois bancos de dados tradicionais. Os bancos de dados escolhidos foram: Monet, de memória principal, e MySQL e PostgreSQL, tradicionais.

Para que uma medida de desempenho seja feita de modo que seja válida para o uso em aplicações *web*, o *benchmark* escolhido foi o TPC-W, que especifica a implementação de uma loja de livros e *browsers* emulados para acessar essa loja, de modo que é possível fazer uma análise de desempenho.

Este trabalho irá mostrar um estudo sobre as teorias envolvidas e os resultados dos testes aplicados, em que o Monet não mostra ter um desempenho superior em todos os casos e nem está maduro o suficiente para ser usado na prática em replicação de aplicações *web*.

Portanto, outras soluções baseadas em sistemas de gerência de persistência alternativos devem ser consideradas.

Abstract

Web applications are very common applications nowadays and it is necessary to find solutions for performance improvements for these applications. There are several ways to implement these performance improvements, including applications and databases replication and usage of main-memory databases.

Looking for performance improvements we can think about using main-memory databases together with replication algorithms. In order to implement this, it is necessary to choose a main-memory database that are stable and with good performance to be used to implement a replication layer on it.

The objective of this work is analyzing a main-memory database performance and compares it with the performance of two traditional databases. The database systems chosen were: Monet, as a main-memory database, and MySQL and PostgreSQL, as traditional databases.

In order to have a benchmark that is valid for web applications usage we chose the TPC-W benchmark, which specifies a book store implementation and emulated browsers to access this shop, which allows an analysis on database performance.

This work will show a study about theories involved and the results of executed tests, where Monet's performance does not seem to be better performance in most cases and Monet seems not be stable enough to be used on a real system for replication of web applications.

Therefore, other solutions based on alternative persistence management systems should be considered.

Conteúdo

Capítulo 1 Introdução 1
1.1 Fatores de desempenho 2
1.2 Avaliação dos SGBDs 3
Capítulo 2 Sistemas de Gerência de Bancos de Dados 5
2.1 SGBDs relacionais: um breve histórico 5
2.2 Arquitetura de *software* de SGBDs 6
2.3 Paralelização 8
2.4 Sistemas de bancos de dados de memória principal 10
2.5 SGBDs alternativos 10
2.5.1 Orientação a objetos e SGBDs 11
2.5.2 Suporte de decisão e *data mining* 12
2.6 Sumário 13
Capítulo 3 Replicação de bancos de dados 15
3.1 Replicação 15
3.2 Um modelo para replicação 16
3.3 Primitivas de grupo 19
3.4 Replicação em sistemas distribuídos 20
3.4.1 Replicação ativa 20
3.4.2 Replicação passiva 20
3.4.3 Replicação semi-ativa 21
3.4.4 Replicação semi-passiva 21
3.5 Replicação em bancos de dados 22
3.5.1 Modelo de replicação em bancos de dados 22
3.5.2 Estratégias de replicação 22
3.6 Sumário 27

Capítulo 4 Monet 29
4.1 Requisitos 29
4.2 Arquitetura de *software* do Monet 30
4.2.1 Back-end e front-end separados 30
4.2.2 Armazenamento de dados em tabelas binárias 32
4.2.3 Não reinventar o sistema operacional 32
4.2.4 Otimização para execução de consultas em memória principal 33
4.3 Sumário 35
Capítulo 5 TPC-W 37
5.1 Especificação TPC-W 37
5.1.1 Métricas geradas 38
5.1.2 Solução típica de *hardware* e *software* para TPC-W 39
5.1.3 Projeto lógico dos dados 40
5.1.4 Preenchimento do banco de dados 41
5.1.5 Regras de interação e navegação 42
5.1.6 *Browsers* emulados e métricas 43
5.2 Implementação TPC-W 46
5.3 Restrições 47
5.4 Sumário 48
Capítulo 6 Experimento 49
6.1 Descrição do experimento 49
6.1.1 Testes realizados 51
6.2 Resultados e Análise 52
6.2.1 30 *browsers* e banco com dados para 30 *browsers* 52
6.2.2 150 *browsers* e banco com dados para 30 *browsers* 52
6.2.3 300 *browsers* e banco com dados para 30 browsers 55
6.2.4 600 *browsers* e banco com dados para 30 *browsers* 57
6.2.5 150 *browsers* e banco com dados para 150 *browsers* 59
6.2.6 300 *browsers* e banco com dados para 300 *browsers* 61
6.2.7 600 *browsers* e banco com dados para 300 *browsers* 63
6.3 Sumário 65

Capítulo 7 Conclusão 67
Referências Bibliográficas 69

Índice de figuras

Figura 1 – Arquitetura de um SGBD relacional 8
Figura 2 - Modelo de replicação em cinco fases. 17
Figura 3 – Arquitetura do Monet 31
Figura 4 – Arquitetura típica de *hardware* e *software* para TPC-W. 40
Figura 5 – Diagrama Entidade-Relacionamento do banco de dados de teste 41
Figura 6 – Fluxo de navegação entre as páginas do TPC-W 44
Figura 7 – Probabilidade de acesso a cada uma das páginas do TPC-W 45
Figura 8 – Tabela com pesos de navegação para o perfil *shopping*. 46
Figura 9 – Arquitetura de *hardware* e *software* utilizada no experimento.................................. 50
Figura 10 – Desempenho dos bancos de dados para 30 *browsers* simultâneos e banco populado para 30 *browsers*. 53
Figura 11 - Desempenho dos bancos de dados para 150 *browsers* simultâneos e banco populado para 30 *browsers*. 54
Figura 12 - Desempenho dos bancos de dados para 300 *browsers* simultâneos e banco populado para 30 *browsers*. 56
Figura 13 - Desempenho dos bancos de dados para 600 *browsers* simultâneos e banco populado para 30 *browsers*. 58
Figura 14 - Desempenho dos bancos de dados para 150 *browsers* simultâneos e banco populado para 150 *browsers*. 60
Figura 15 - Desempenho dos bancos de dados para 300 *browsers* simultâneos e banco populado para 300 *browsers*. 62
Figura 16 - Desempenho dos bancos de dados para 600 *browsers* simultâneos e banco populado para 300 *browsers*. 64

Capítulo 1

Introdução

Aplicações *web* são cada vez mais comuns em nosso cotidiano. Com o uso intenso dessas aplicações torna-se necessária a busca por soluções que possam melhorar o seu desempenho, a sua escalabilidade e principalmente a sua disponibilidade[1].

A arquitetura de software para aplicações *web* mais comum é composta de três camadas. A primeira camada é responsável por tratar as requisições dos clientes e o processamento relativo à interface com o usuário. Ela se comunica com um servidor de aplicações para que a operação desejada seja executada. Este, por sua vez, precisa de acesso aos dados para executar as operações e se comunica com a terceira camada, que é, em geral, implementada por um servidor de banco de dados.

Para melhorar o desempenho e ter escalabilidade faz-se necessário o uso de replicação nessas camadas de forma a atender uma demanda de clientes cada vez mais crescente. A replicação da primeira e segunda camadas é viável, apesar da complexidade envolvida nas soluções, porque essas camadas não retém estado. Em contraste, não existe uma forma simples de fazer a replicação do banco de dados. Assim, a replicação da primeira e segunda camadas pode tornar o banco de dados o gargalo da aplicação *web*. Existem soluções comerciais de replicação para bancos de dados, mas que têm o foco em tolerância à falhas e não a melhora de desempenho da aplicação. Nesse cenário, a solução comumente encontrada para a melhora de desempenho do banco de dados é a compra de um servidor mais poderoso.

---

[1] Do Inglês: *availability*

A próxima seção traz uma avaliação dos fatores que influenciam o desempenho de cada uma das camadas de aplicações *web* e aponta solução alternativa para a compra de um servidor mais poderoso para a terceira camada.

1.1 Fatores de desempenho

O poder de processamento dos computadores aumentou consideravelmente nas últimas décadas. Também tivemos aumento de tamanho e velocidade de memória principal e de disco. Infelizmente a taxa de diminuição da latência de acesso ao disco e à memória principal não se realizou em taxa comparável à do aumento de velocidade de acesso e tamanho desses componentes. Conseqüentemente, temos um aumento relativo no custo da latência de acesso ao disco e à memória em comparação às outras grandezas que têm impacto no desempenho geral do sistema.

Em relação à memória principal, se levarmos em conta, por exemplo, o aumento da velocidade de CPU e a diminuição da latência ocorrida nos últimos anos, percebe-se que a CPU gasta hoje muito mais ciclos aguardando a resposta da memória principal do que gastava anos atrás. A maneira utilizada pelos fabricantes de *hardware* para contornar este problema é o uso da memória cache, porém se uma aplicação tiver baixa taxa de acerto o cache continuará dependendo da latência de memória.

Nos discos este comportamento não é muito diferente. Embora haja uma diminuição maior da latência de disco em relação à da memória principal, ela continua sendo importante em consultas OLTP (*Online Transaction Processing*), que possuem um padrão de acesso de um bloco por vez. Para melhorar o desempenho poderíamos utilizar RAID[2] nos discos, porém teremos um aumento em velocidade e uma piora na latência.

Estudos mostram que em vários SGBDs a CPU fica parada na maioria do tempo. Grande parte desse tempo parado é causado pela latência de memória. Outro fator é que as instruções de máquina usadas nos algoritmos típicos de SGBD são altamente interdependentes, causando paradas por dependência. Como as CPUs modernas são mais rápidas não só devido ao aumento do *clock*, mas também devido à capacidade de executar mais instruções em paralelo, essas paradas por desempenho se tornam cada vez mais evidentes.

---

[2] RAID é a abreviação de *Redundant Array of Independent Disks*.

Portanto, torna-se necessário encontrar uma solução para aumentar a escalabilidade e o desempenho dos bancos de dados considerando a realidade atual do *hardware*.

Com o grande aumento da capacidade de memória principal e sua redução de custo o uso de bancos de dados em memória principal passa a ser possível. Desta forma, todo o acesso a dados é feito em memória principal e o disco é usado para garantir a persistência e tolerância à falhas, além de conter os *logs* de acesso. Esse tipo de banco de dados pode ser implementado utilizando estruturas de dados otimizadas para o uso em memória e tentar tirar proveito da memória cache do processador. Como os dados ficam em memória, os acessos a disco para leitura desses dados passa a ser seqüencial e todo acesso aleatório é feito em memória principal.

Em tese, o uso de SGBDs de memória principal pode melhorar o desempenho geral de uma aplicação *web*, porém ainda temos o problema da disponibilidade. Este problema pode ser resolvido com o uso de replicação. Existem muitas pesquisas voltadas ao uso de replicação em bancos de dados e temos duas formas de replicação que podem ser utilizadas. A primeira é síncrona, que se baseia em protocolos bloqueantes e originalmente é muito custosa. A segunda é assíncrona, que resolve o problema no custo de replicação, mas pode gerar inconsistência entre os dados replicados. Os sistemas comerciais existentes hoje implementam replicação síncrona e buscam principalmente melhorar a sua disponibilidade.

A solução para a escalabilidade e a disponibilidade pode estar no uso de replicação em bancos de dados de memória principal. Para isto, é necessário escolher um SGBD de memória principal que tenha desempenho melhor que o desempenho médio dos bancos de dados tradicionais e permita o seu emprego em uma configuração replicada.

1.2 Avaliação dos SGBDs

Dentro do contexto de banco de dados e aplicações web, o foco deste trabalho é a análise do desempenho de um banco de dados de memória principal para aplicações *web* e sua comparação com outros dois bancos de dados tradicionais. Para realizar esse trabalho, três SGBDs foram escolhidos:

Monet: banco de dados em memória principal que busca tirar proveito da arquitetura dos *hardwares* modernos.

MySQL: consagrado banco de dados de código aberto usado tanto para aplicações comerciais quanto no mundo acadêmico.

PostgreSQL: banco de dados de código aberto conhecido por seu excelente desempenho.

Para fazer a comparação do desempenho entre os SGBDs é necessário encontrar uma forma de medição do desempenho dos mesmos para uso em aplicações *web*. Existem vários *benchmarks* disponíveis, mas eles não levam em conta a carga de uso de uma aplicação *web*. Por exemplo, o TPC-C [10] é um *benchmark* para uma arquitetura cliente-servidor em que há comunicação direta entre o terminal cliente com o SGBD; e o TPC-H [13] é um *bechmark* de sistemas de suporte de decisão, onde os dados são lidos de um SGBD OLTP e armazenados no sistema de suporte de decisão. O primeiro *benchmark* específico para a avaliação de desempenho de aplicações *web* é o TPC-W [14].

TPC-W é uma especificação que foi desenvolvida para medir o desempenho de aplicações *web*, simulando a atividade de uma loja de livros, em que o usuário pode fazer buscas, escolher produtos e fechar suas compras. Para gerar a carga de uso existem *browsers* emulados que geram as requisições ao sistema, simulando um usuário real do sistema. Com isto foi possível usar uma implementação baseada na especificação TPC-W para medir o desempenho de cada um dos SGBDs em diversas condições de uso.

Outro teste que poderia ser executado é o teste de junção de tabelas relacionais, *winestore*, baseado no livro de Hugg Williams[17], que foi apresentado no livro de Fundamentos de Bancos de Dados, de Célio Guimarães[8], onde este teste avalia o desempenho de SGBDs para cargas com perfis de OLAP e *data mining*. Porém, não foi possível sua execução para todos os SGBDs, pois o Monet não conseguia executar as consultas, sendo seu processo interrompido cada vez que o teste era executado.

Este trabalho está organizado da seguinte forma. No Capítulo 2 resume-se os passos que levam ao desenvolvimento das arquiteturas atuais de SGBDs. O Capítulo 3 aborda a replicação de dados. O Capítulo 4 descreve o banco de dados de memória principal Monet. O Capítulo 5 apresenta o *benchmark* utilizado neste trabalho. No Capítulo 6, temos a descrição e discussão dos resultados obtidos neste experimento para avaliar Monet, MySQL e PostgreSQL. O Capítulo 7 é o capítulo de conclusão do trabalho.

Capítulo 2

Sistemas de Gerência de Bancos de Dados

Sistemas de Gerência de Bancos de Dados (SGBD) servem para garantir a persistência e acesso estruturado aos dados utilizados por diversas aplicações. Existem várias possibilidades para a implementação de um SGBD: arquivos, bancos hierárquicos, bancos relacionais e orientados a objetos; a organização mais freqüente é a relacional.

Este capítulo traz um histórico do desenvolvimento do banco de dados relacional, uma descrição da arquitetura empregada na sua implementação, e aponta possibilidades de revisão da arquitetura para a melhoria do seu desempenho. Outros SGBDs são mencionados com a finalidade de alertar o leitor para a possibilidade de avaliação desses sistemas no contexto de replicação.

2.1 SGBDs relacionais: um breve histórico

Em 1970 foi publicado o artigo que estabeleceu o modelo relacional [5]. As primeiras implementações conhecidas do modelo relacional são INGRES (UC Berkeley) [12] e System R (IBM) [1].

INGRES usava UNIX como plataforma de implementação e usava a linguagem QUEL, que é similar ao SQL atual. Algumas de suas idéias podem ainda ser encontradas nas plataformas atuais como implementação de *views* e restrições de integridade através do reescrevedor de consultas, além do uso das tabelas relacionais para guardar meta informação. Ele chegou a ser

comercializado como um produto e evoluiu para o Sybase DBMS, que posteriormente se transformou no Microsoft SQLServer.

O System R introduziu a linguagem SQL e representa um trabalho exemplar na área de *logging* e recuperação de estado, além de ser o pioneiro a implementar otimização de consultas. Seu algoritmo de consulta é ainda usado em vários dos produtos atuais.

O desempenho das transações foi uma das principais preocupações dos primeiros sistemas relacionais. Uma técnica importante usada pelo System R foi interpretação das consultas e esforço de otimização em transações padrão para várias execuções, compilando consultas repetidas em planos de busca *hard-coded*.

Alguns resultados importantes das pesquisas no System R levaram ao desenvolvimento de bancos de dados distribuídos e gerenciamento otimizado de transações. Sub-sistemas do System R foram usados em vários produtos da IBM e o Oracle foi desenvolvido com estrutura de software muito influenciada pela estrutura do System R.

Em busca de alto desempenho, no início do desenvolvimento dos bancos relacionais, pensou-se no uso de equipamentos especializados para rodar os SGBDs. Esses sistemas, que consistiam do *software* e do *hardware* específicos, eram chamados de *máquinas de banco de dados*.

Com o advento de processadores de uso geral e discos mais rápidos e de custo menor, *máquinas de banco de dados* foram gradativamente abandonadas porque representam, em relação à solução geral, maior custo de manutenção e menor aplicabilidade.

Atualmente dois desenvolvimentos lembram algumas técnicas usadas na era das máquinas especializadas: instruções específicas para multimídia nos processadores e IRAM, uma memória que contém uma CPU internamente para executar operações dentro da própria memória, liberando o gargalo do *bus* do sistema. Mesmo assim, os SGBDs não utilizam *hardware* especificamente desenvolvido para eles e são instalados em máquinas de propósito geral que foram apenas configuradas para abrigarem os bancos de dados.

Neste capítulo serão mostradas outras técnicas que podem ser usadas em busca de melhor desempenho e que não dependem de equipamentos especializados.

2.2 Arquitetura de *software* de SGBDs

Uma arquitetura de *software* típica de SGBDs atuais é ilustrada na Figura 1 e contém pelo menos os seguintes sub-sistemas:

- **processador da linguagem de consulta[3]:** lê as entradas de uma linguagem, como SQL, por exemplo, verificando inconsistências e, se nenhuma é encontrada, traduz para uma representação interna (normalmente uma árvore de processamento[4]).
- **reescrevedor de consulta:** lê a consulta já processada e a reescreve para uma forma padrão, verificando regras de autorização e semântica, expandindo as *table-views* para sua definição completa.
- **otimizador de consulta:** traduz a descrição lógica para um plano de execução da consulta. Entre as várias traduções possíveis tenta escolher a melhor em relação a tempo de resposta e uso de recursos.
- **executor de consulta:** executa o plano de execução e produz o resultado final.
- **métodos de acesso:** são serviços que provêem acesso aos dados usados nas tabelas do banco. Usa estrutura de índices como árvores balanceadas e *hashes* para acelerar o acesso.
- **gerenciador de *buffer*:** cuida do cache dos dados, em memória principal, das tabelas que estão no disco, para minimizar o número de I/Os.
- **gerenciador de transação:** provê serviço de *lock* para as transações.
- **gerenciador de recuperação:** garante a persistência dos dados confirmados e o apagamento dos dados das transações abortadas.

Dada uma arquitetura de *software* tão complexa quanto a exibida por SGBDs relacionais modernos parece ser razoável que se procure uma forma de replicação de dados em memória principal que permita tratar o SGBD como uma caixa preta. Essa é a abordagem escolhida para esse trabalho comparativo. A estratégia de caixa branca oferece a possibilidade de otimização da arquitetura de *software* como um todo e provavelmente poderá produzir um MMDB (*Main-Memory Database*) replicado de melhor desempenho. Entretanto, ela impedirá a substituição de SGBDs, compromentendo a construção de sistemas replicados onde SGBDs diferentes convivem.

---

[3] Do Inglês: *query language parser*

[4] Do Inglês: *parse tree*

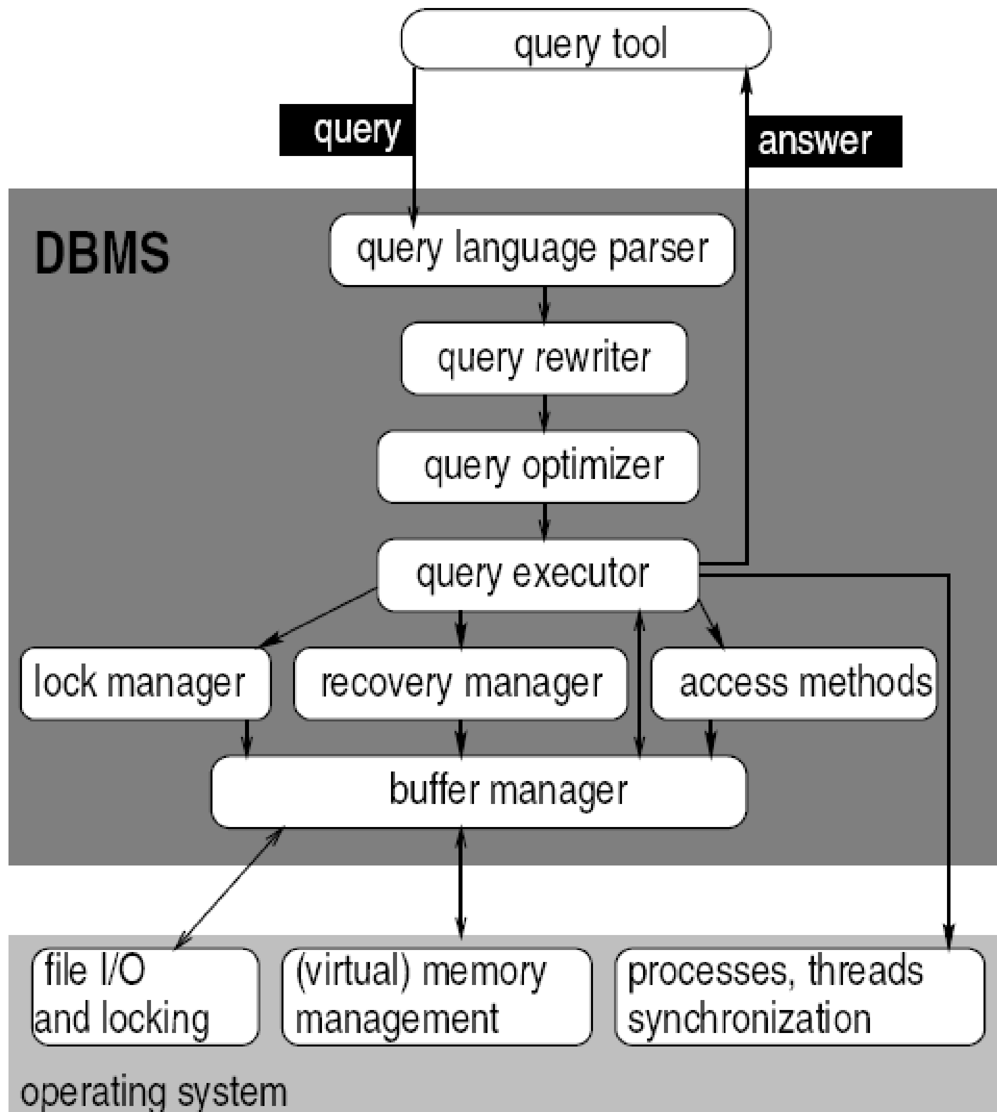

Figura 1 – Arquitetura de um SGBD relacional

Fonte: Tese de Doutorado de Peter Boncz [2]

2.3 Paralelização

Em busca de maior desempenho dos SGBDs, novas técnicas foram criadas, entre elas o uso de paralelismo das execuções. Existem dois alvos para o uso de paralelismo em bancos de dados: aumento de velocidade, em que tarefas de igual tamanho são executadas mais rapidamente no sistema paralelo ou aumento de escalabilidade, em que uma tarefa maior pode ser executada no mesmo tempo.

Os tipos de paralelismo existentes são:

- entre consultas[5], em que várias sentenças são executadas paralelamente em diferentes máquinas e é normalmente usada para aumentar a escalabilidade.
- dentro das consultas[6], em que uma sentença é quebrada em várias tarefas a serem executadas por diferentes máquinas.

A arquitetura de *hardware* dos SGDBs paralelos pode ser classificada como:

- tudo compartilhado: Todas as CPUs têm acesso à mesma memória e discos;
- memória compartilhada: acesso à mesma memória, mas discos diferentes;
- disco compartilhado: memória individual, mesmos discos;
- nada compartilhado: não compartilha nada e se comunica via rede.

As três primeiras opções caem no caso de *hardware* especializado, que não apresentam vantagem devido aos problemas já levantados. Sistemas com memória compartilhada começam a se tornar populares atualmente. Também houve grande avanço nos sistemas de rede. Com isso, uma arquitetura híbrida em que vários equipamentos, cada um com duas ou quatro CPUs se comunicam pela rede se tornam viáveis e razoável em termos de custo/benefício.

Existem duas técnicas para paralelizar a execução das sentenças que formam uma transação relacional:

- paralelismo vertical: os operadores da consulta são distribuídos entre as várias CPUs. Este sistema tem um balanço de carga ruim, pois os operadores não possuem custos iguais.
- paralelismo horizontal: os mesmos operadores são executados em parelelo em CPUs diferentes e cada processador processa um conjunto diferente de ênuplas[7].

O algoritmo mais popular para *join* distribuído é o *hash-join*. Usando esse algoritmo, as duas relações a serem unidas são fragmentadas por *hash* e distribuídas através da *hash-key* nas CPUs. Cada par de fragmentos em que batem as *hash-keys* são então unidos.

É possível também ter sistemas em que ambos os tipos de paralelismo são usados simultaneamente.

---

[5] Do Inglês: *inter-query*

[6] Do Inglês: *intra-query*

[7] Do Inglês: *tuples*

Quase todos os bancos relacionais atuais utilizam paralelismo entre consultas e alguns produtos oferecem opções para se usar paralelismo dentro das consultas em *clusters* nada compartilhado. Essas implementações têm como alvo consultas simples.

2.4 Sistemas de bancos de dados de memória principal

Outra técnica existente para melhora de desempenho em SGBDs é o uso de bancos de dados em memória principal. Esta técnica se tornou factível com a queda nos preços das memórias e então os bancos de dados poderiam ser armazenados totalmente em RAM, eliminando o custo de I/O.

Um problema levantado é como fazer transações e recuperação de dados numa maneira eficiente. Alguns algoritmos supõem que parte da memória nunca irá se perder e são usadas para o *log* de recuperação. Outros ainda usam disco para armazenar somente as informações de *log*, que, embora não eliminem I/O, somente escritas no *log* são feitas. Estruturas de dados foram estudadas para usar-se com esses sistemas, e chegou-se à conclusão que estruturas de dados simples podem ser usadas, como árvores balanceadas e *hashes*.

Outro problema é a otimização de consultas, já que o maior problema em sistemas com acesso a disco é justamente o I/O, portanto os pontos de possíveis otimizações acabam sendo pontos obscuros como o custo de execução de uma rotina na CPU.

2.5 SGBDs alternativos

Os primeiros bancos de dados foram criados para aplicações cujos dados se resumiam a números e textos simples. Porém, existem outros domínios de aplicações que também precisam processar grande quantidade de dados, por exemplo, aplicações geográficas.

Para isto, foram feitas pesquisas para tornar os SGDBs extensíveis para esses domínios. O Postgres (sucessor do INGRES) colocou no núcleo do sistema algumas interfaces de dados abstratos, permitindo que módulos fossem colocados para tratar novos tipos de dados atômicos, novas funções nesses tipos e novas estruturas de índices.

Quando novos tipos de dados são adicionados, o otimizador de consulta precisa saber o que fazer quando encontrar esse tipo de dado. Para isto, precisa ser estendido também. Essa extensão

pode ser feita pelo programador do núcleo, por um desenvolvedor de módulo, pelo desenvolvedor da aplicação ou pelo usuário final.

A realização de extensões diretamente no núcleo pode introduzir instabilidade no funcionamento do banco, já que defeitos no projeto e/ou implementação da extensão podem comprometer o funcionamento do SGBD como um todo.

Extensões realizadas como módulos externos ao núcleo impedem que o SGBD como um todo seja comprometido se tiverem problemas. Porém, se rodar num espaço de endereçamento independente do núcleo, um sistema de comunicação entre processos deve ser usado, o que pode gerar um grande *overhead*. Os sistemas comerciais usam diferentes implementações, alguns permitindo execução diretamente no núcleo, outros permitindo somente execução em processos separados.

2.5.1 Orientação a objetos e SGBDs

Sistemas de bancos de dados orientados a objeto permitem que o modelo de dados seja estendido mais facilmente porque se apóiam no conceito de classes para representar os dados. Um banco de dados orientado a objetos é a união de coleções, que contém todos os objetos de uma classe. Esses sistemas têm algumas vantagens sobre os convencionais, porque o comportamento das classes pode ser especializado usando herança e polimorfismo. Há várias alternativas para a implementação de SGBDs orientados a objeto:

***Tookits*:** com a utilização de *tookits*, o acesso aos dados é feito utilizando-se as ferramentas da própria linguagem. Por exemplo, um método de uma classe pode ser considerado persistente e uma extensão na linguagem de programação se encarrega de criar esta persistência. Neste caso, não há necessidade de se utilizar uma linguagem intermediária, como o SQL, para realizar o acesso aos dados no SGBD. Esta alternativa não é muito usada em aplicações comerciais.

Orientado a objetos: sistemas que são puramente bancos de dados voltados a objetos, onde as linguagens orientadas a objeto podem ser usadas para acessar diretamente os dados armazenados nos bancos. O banco de dados é somente um objeto persistente e não há necessidade de se usar comandos SQL e processar resultados de retorno. Um padrão foi estabelecido pela ODMG [3][4], definindo o modelo de dados ODL, porém, os sistemas disponíveis não implementam ainda o padrão totalmente. Essas qualidades também são o ponto fraco desse tipo de sistema. O *layout* do banco fica totalmente a cargo da forma com que foi

empregado na linguagem, eliminando a liberdade de se usar estruturas físicas que otimizem alguns padrões de acesso. Também, existe dificuldade para otimização, paralelização, mudança física da localização dos dados, uso de índices e mudança de esquemas relacionais.

Objeto relacional: Os fornecedores de bancos de dados estendem gradualmente seus bancos já existentes para fornecer algumas funcionalidades dos bancos orientados a objetos. Isto não garante uma arquitetura elegante, porém preserva as qualidades do modelo relacional.

2.5.2 Suporte de decisão e *data mining*

Data Warehousing é uma área de negócios interessante, onde uma organização junta os dados de vários sistemas de informação em um banco de dados enorme. Normalmente é usado em aplicações de consultas massivas[9], OLAP (*On Line Analytical Processing*) e *data mining*, onde os dados são preenchidos a partir de sistemas comuns OLTP, fazendo um pré-processamento nos dados[10], criação de índices, etc, em que um grande tempo computacional é usado.

OLAP e *data mining* são usados para fazer análises complexas. Fazer isso num banco de dados relacional comum tem sérios problemas de desempenho. Uma solução para isso, desenvolvida pelos criadores de SGDB comerciais, foi um sistema de suporte de decisão especializado que fica entre a aplicação OLAP e o sistema relacional, ou um servidor independente que manipula estruturas de dados multi-dimensionais ao invés de tabelas relacionais. Existem duas soluções: MOLAP, usando vetores multidimensionais e ROLAP, em que um *middleware* que fica entre o sistema OLAP e o gerenciador de banco, fazendo cache de informações, otimização de carga e uso de alguns índices. Também é possível encontrar um misto dessas duas tecnologias.

O problema dessas soluções é o uso de otimização através do reuso de resultados pré-computados e a área de uso, suporte de decisão, é um processo interativo, que não é previsível, fazendo com que as operações pré-processadas possam não ser úteis e pode-se obter um desempenho muito ruim do sistema. Isso faz com que esse tipo de solução seja impraticável para sistemas de *data mining*.

---

[9] Do Inglês: *query-intensive*

[10] Do Inglês: *data cleaning*

2.6 Sumário

Este capítulo sumarizou a evolução pela qual têm passado os sistemas de gerência de bancos de dados, com ênfase nos SGBDs relacionais. SGBDs orientados a objeto, híbridos relacional-objeto e sistemas de consulta baseados em linguagens de programação também foram abordados brevemente.

Observa-se que aplicações modernas que utilizam SGBDs, tanto para OLAP quanto para OLTP, podem ter seu desempenho, disponibilidade e escalabilidade melhorados se incorporarem avanços realizados em MMDBs e replicação. Esta dissertação é parte da investigação necessária para validar o desempenho no emprego de MMDBs em aplicações *web*.

O próximo Capítulo aborda replicação com o objetivo de mostrar as dificuldades existentes na sua aplicação para SGBDs.

Capítulo 3

Replicação de bancos de dados

Replicação de dados é um assunto de interesse tanto da área de sistemas distribuídos quanto da área de banco de dados. Em geral, em sistemas distribuídos a replicação é usada para garantir tolerância à falhas e em bancos de dados para melhorar desempenho.

Este Capítulo resume trabalho recente realizado nessas duas áreas que permite ter uma visão homogênea das técnicas de replicação utilizadas em sistemas distribuídos e SGBDs. Esta visão unificada permite a verificação de que é possível a proposição de um mecanismo de replicação de bancos de dados em memória principal que utiliza protocolos originalmente desenvolvidos para replicação em sistemas distribuídos.

3.1 Replicação

Para o estudo sobre replicação, supõe-se que o sistema replicado é composto por um conjunto de cópias dos dados sobre os quais as operações são executadas. Estas operações são iniciadas pelos clientes e a comunicação entre os diferentes componentes do sistema é realizada exclusivamente via troca de mensagens.

Neste contexto, os sistemas distribuídos se distinguem em síncronos e assíncronos. No modelo síncrono há um limite superior para a velocidade de execução do processo e para o atraso de comunicação, e no assíncrono não há. A principal diferença é que um sistema síncrono permite uma detecção precisa de falhas[11] e o assíncrono não[7]. Se a detecção precisa não for possível, a implementação de replicação é dificultada. Uma forma de reduzir-se essa complexidade é através do uso de algoritmos de comunicação em grupo.

---

[11] Do Inglês: *correct crash detection.*

Sistemas distribuídos usam protocolos não bloqueantes. Em contraste, SGBDs são construídos, na grande maioria das vezes, sobre protocolos bloqueantes e síncronos. Assim, no caso de ocorrência de falhas, os protocolos de replicação de bancos de dados aceitam, em alguns casos, a intervenção do operador para resolver casos anormais, o que não é possível aceitar para a maioria dos protocolos empregados em sistemas distribuídos.

Finalmente, sistemas distribuídos consideram a possibilidade de comportamento determinista ou probabilístico de uma réplica. No comportamento determinista, se as réplicas recebem as operações[12] exatamente na mesma ordem, garante-se que fornecerão sempre os mesmos resultados. Isso pode não ser verdade em bancos de dados porque internamente os bancos alteram a ordem de leituras e escritas. Portanto, pode ser necessária a comunicação entre as réplicas para obter o consenso dos resultados.

Dessa introdução ao problema de replicação em SGBDs e em sistemas distribuídos verifica-se que a área de bancos de dados trata replicação de uma forma diferente que a área de sistemas distribuídos. Para tentar uniformizar essas visões, Wiesmann elaborou uma taxonomia dos protocolos de replicação de dados existentes para bancos de dados e sistemas distribuídos [16], unificando a sua descrição. As seções seguintes se apóiam no trabalho de Wiesmann para argumentar que é possível produzir um SGBD replicado que aproveita avanços em replicação realizados em sistemas distribuídos.

3.2 Um modelo para replicação

Wiesmann [16] propõe um modelo funcional de um protocolo de replicação para o estudo dos vários modelos de replicação tanto em sistemas distribuídos quanto em bancos de dados. Neste modelo, o protocolo de replicação pode ser descrito em cinco fases. Os diferentes protocolos podem ser comparados pela maneira que implementam cada uma das fases e como combinam fases diferentes.

- **Requisição (*Request*):** o cliente submete uma operação a uma ou mais réplicas.

[12] Operação pode ser considerada como uma simples leitura ou escrita de dados, um processamento mais complexo com vários parâmetros ou a chamada de um método ou função.

- **Coordenação dos servidores (*Server coordination*):** os servidores de réplica se coordenam uns com os outros para sincronizar a execução das operações (ordem de operações concorrentes).
- **Execução (*Execution*):** a operação é executada nas réplicas.
- **Coordenação de decisão (*Agreement coordination*):** as réplicas entram em acordo com os resultados, para garantir atomicidade.
- **Resposta (*Response*):** o resultado da operação é enviado ao cliente.

A figura a seguir ilustra as cinco fases mostradas acima.

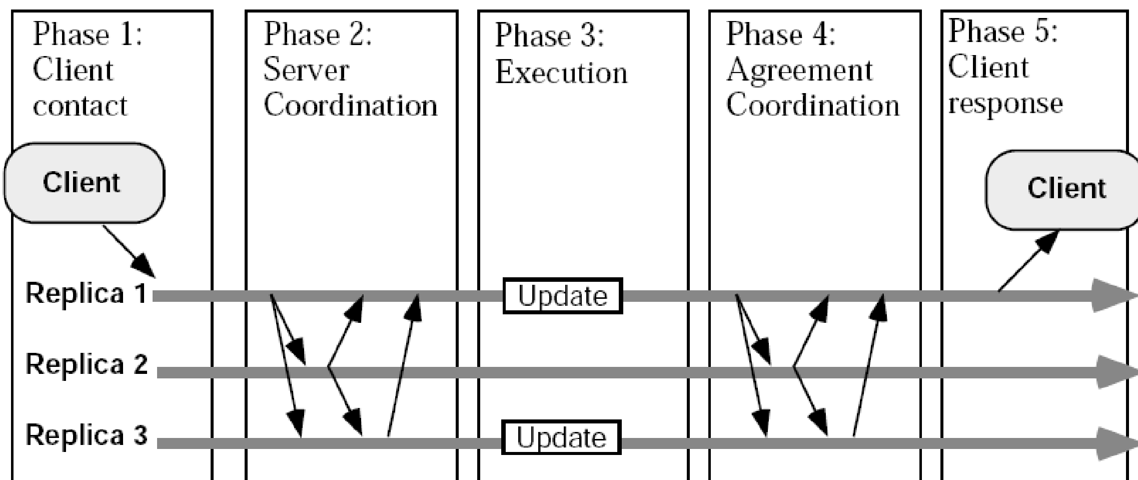

Figura 2 - Modelo de replicação em cinco fases.
Fonte: Artigo publicado por Wiesmann [16]

Para a análise do funcionamento de cada fase, considera-se que as transações são compostas por uma única operação. Em sistemas distribuídos isso é o que geralmente acontece, mas uma transação de banco de dados pode ser composta por várias operações. Durante a análise das diferentes técnicas de replicação serão mostradas as diferenças para várias operações.

Fase de requisição (*Request*): Durante esta fase, o cliente envia a operação para o sistema. Isto pode ser feito enviando somente a uma réplica ou a todas.

Em bancos de dados, os clientes normalmente nunca entram em contato com todas as réplicas. O cliente submete a operação a um nó que repassa para os outros. Com isto, o cliente não tem que saber da existência de todas as réplicas, sendo uma operação transparente. Em

sistemas distribuídos isso é diferente, podendo ter replicação ativa ou passiva, conforme mostrado na seção 3.4 .

Fase de coordenação dos servidores (*Server coordination*): Nesta fase as réplicas tentam encontrar uma ordem à qual as operações serão executadas. É neste ponto em que os diferentes protocolos têm comportamentos diferentes.

Em relação às estratégias de ordenação, os bancos de dados ordenam suas operações de acordo com suas dependências, e todas as réplicas devem ter os mesmos dados dependentes. Neste caso a semântica é muito importante: uma operação só de leitura é diferente de uma que faz leitura e escrita. Se elas não têm relação uma com a outra, não é necessário ordená-las. Em sistemas distribuídos, por outro lado, a ordenação total é feita independente da operação, não sendo importante a semântica da operação.

Em relação à correção, protocolos de bancos de dados usam seriação[13] adaptada aos cenários replicados, onde a base da exatidão é a dependência dos dados. Sistemas distribuídos usam linearizabilidade e consistência seqüencial. Linearizabilidade é baseada nas dependências em tempo real enquanto na consistência seqüencial é considerada a ordem das operações em cada processo individual, permitindo em algumas situações a leitura de “dados velhos”. Todas as técnicas de replicação mencionadas neste Capítulo garantem linearizabilidade.

Fase de execução (*Execution*): Representa a execução corrente da operação. Não existem muitas diferenças entre os protocolos. Representa somente a execução e a atualização dos dados é feita na fase de coordenação de decisão.

Fase de coordenação de decisão (*Agreement Coordination*): É nesta fase que as réplicas garantem que todas fizeram a mesma coisa. Em banco de dados, esta fase corresponde ao *two-phase commit*, que é necessário porque na fase de coordenação de servidores somente a ordenação foi garantida e, em bancos de dados, garantir a ordenação das operações não significa que a mesma será executada com sucesso em todas as réplicas. Em sistemas distribuídos isso é diferente porque, uma vez ordenada, a operação será executada, isto é, entregue e não há necessidade de nenhuma verificação adicional.

Fase de resposta ao cliente (*Response*): Este é o momento em que o cliente recebe a resposta do sistema. Existem duas situações: (i) a resposta é enviada depois que todas as réplicas

---

[13] Do Inglês: *serializability*.

executaram a operação, ou (ii) a resposta é enviada assim que uma termina e a propagação de mudanças é feita a posteriori. Em bancos de dados, essa distinção leva a protocolos síncronos (*eager*) ou assíncronos (*lazy*). Em sistemas distribuídos a resposta é enviada após o protocolo ter sido executado sem criar inconsistências.

3.3 Primitivas de grupo

Um sistema distribuído pode ser modelado como um conjunto de serviços implementando pelos processos servidores que podem ser requisitados pelos processos clientes. Cada processo servidor tem um estado local que é modificado através das requisições, que modificam o estado do servidor numa maneira atômica, ou seja, os resultados não são aplicados parcialmente. O isolamento das requisições é feito pelo servidor e normalmente usa algum mecanismo de sincronização local. Para a obtenção de tolerância à falhas, os serviços são implementados usando réplicas, ou seja, vários processos servidores.

Para lidar com a complexidade envolvida em replicação, foi criada a noção de grupo ou primitivas de comunicação de grupo. A noção de grupo permite um mecanismo de endereçamento lógico, permitindo ao cliente ignorar o grau de replicação e a identidade de cada processo individual. Primitivas de comunicação de grupo provêem comunicação de um para muitos com várias semânticas poderosas, que escondem muitas das complexidades de manter a consistência dos servidores replicados. As duas principais primitivas de comunicação em grupo são *Atomic Broadcast (ABCAST)* ou *View Synchronous Broadcast (VCAST)* [16]

***Atomic Broadcast (ABCAST)*:** Provê atomicidade e ordenação total. Atomicidade garante que se um membro enviou uma determinada mensagem, então todos os membros do grupo também enviem a mesma mensagem. A propriedade de ordem garante que se dois membros enviam duas mensagens, elas serão na mesma ordem.

***View Synchronous Broadcast (VSCAST)*:** É definida no contexto de um grupo e é baseada no conceito de seqüência de visões do grupo, onde cada visão define a composição do grupo num certo momento. Uma vez que um processo em alguma visão seja suspeito que estar quebrado, ou novo processo quer entrar no grupo, uma nova visão é instalada para refletir a mudança. Garante a seguinte propriedade: se um processo envia uma mensagem antes de instalar uma nova visão, então nenhum processo instalará uma nova visão antes de ter enviado a mensagem.

3.4 Replicação em sistemas distribuídos

3.4.1 Replicação ativa

Replicação ativa é uma técnica de replicação onde todas as réplicas recebem e processam a mesma seqüência de operações. A consistência é garantida executando as mesmas entradas na mesma ordem que resultam nas mesmas saídas, o que implica que os servidores processam as requisições de maneira determinista.

Os clientes não entram em contato com um servidor em específico, pois os servidores são vistos como um grupo. Para isso as requisições dos clientes podem ser propagadas usando *Atomic Broadcast*.

As principais vantagens desta replicação são sua simplicidade e tolerância à falhas. Falhas são totalmente escondidas dos clientes. O principal problema é garantir a ordem, que faz com que os dados sejam processados em todas as réplicas. A garantia de ordem é fonte de custos nem sempre pequenos.

Os seguintes passos são executados:

- O cliente envia a requisição com *Atomic Broadcast.*
- A coordenação dos servidores garante a ordem de execução.
- Todas as réplicas executam a requisição.
- Não é necessário coordenação de decisão, pois foi tudo executado na mesma ordem.
- Todas as réplicas enviam o resultado para o cliente, que geralmente processa só a primeira.

3.4.2 Replicação passiva

A principal característica desta replicação é que o cliente envia a requisição ao servidor primário, que executa a ação e envia o resultado às réplicas para ser aplicado. Com isto, não temos nenhum problema de ordenação e/ou determinismo.

A comunicação entre o primário e os *backups* deve garantir que as alterações sejam feitas na mesma ordem. Comunicação FIFO poderia resolver o problema, mas não é suficiente em caso de falha do servidor primário. O mecanismo VSCAST pode garantir que isto não é um problema e normalmente é utilizado para implementar a técnica de replicação entre primário e *backup*.

Replicação passiva pode tolerar servidores não deterministas e usar menos poder de processamento, mas tem um custo alto de recuperação no caso de falha do primário.

Os seguintes passos são executados:

- O cliente envia a requisição ao primário.
- Não há coordenação inicial entre os servidores.
- O primário executa a requisição.
- O primário coordena com as réplicas enviando as atualizações.
- O primário envia a resposta ao cliente.

3.4.3 Replicação semi-ativa

Replicação semi-ativa é uma solução intermediária entre a ativa e a passiva. Ela não requer que as réplicas processem as invocações de serviço de uma maneira determinista.

A principal diferença entre a semi-ativa e a ativa é que cada vez que as réplicas tomarem uma decisão não determinista, um processo, chamado líder, faz a escolha da decisão e envia aos processos seguidores[14]. Neste caso, as fases de execução e coordenação de decisão são repetidas para cada escolha não determinista.

Os seguintes passos são executados:

- O cliente envia a requisição usando *atomic broadcast.*
- Os servidores se coordenam usando a ordem recebida no *atomic broadcast*.
- Todas as réplicas executam a requisição na ordem entregue.
- No caso de uma escolha não determinista, o líder faz a escolha usando VSCAST.
- Os servidores enviam a resposta ao cliente.

3.4.4 Replicação semi-passiva

É uma variação da replicação passiva que pode ser implementada no modelo assíncrono sem a noção de visões. Como esta técnica não tem equivalente em banco de dados ela não é mostrada em detalhes.

---

[14] Do inglês: *followers*.

Em resumo, a replicação em sistemas distribuídos é influenciada por dois parâmetros: (i) transparência de falhas para o cliente e (ii) determinismo.

3.5 Replicação em bancos de dados

A replicação em bancos de dados é feita geralmente por questões de desempenho, com o objetivo de acessar dados localmente ao invés de haver comunicação remota. Isso normalmente é possível quando as operações são de leitura. Em operações de escrita, porém, coordenação entre as réplicas é necessária e é impossível evitar comunicação entre servidores remotos.

3.5.1 Modelo de replicação em bancos de dados

Um banco de dados é uma coleção de itens de dados controlados por um SGBD. Um banco de dados replicado é uma coleção de bancos de dados que armazenam cópias dos mesmos dados. A unidade básica de replicação é o item de dados.

Os clientes acessam os dados submetendo transações. Uma operação de uma transação pode ser de leitura ou escrita. A transação é executada atomicamente, ou seja, ela é confirmada ou abortada em todas as réplicas. Além disso, as transações são executadas em paralelo e é necessário que sejam isoladas umas das outras para evitar conflitos que gerem inconsistências; isso pode ser feito através de *locks*.

Um cliente submete transações para somente um banco de dados e, em geral, está conectado somente a um banco de dados. Se ele falhar, a transação é abortada e uma falha é enviada ao cliente, que pode se conectar em outro servidor e re-executar a transação.

3.5.2 Estratégias de replicação

Replicação de bancos de dados pode ser categorizada usando dois parâmetros. Um é quando ocorre a propagação e o outro é quem faz as atualizações. Nos esquemas de replicação síncrona (*eager*), o usuário só irá receber a notificação de confirmação quando todas as cópias do sistema tiverem sido atualizadas. Nos esquemas assíncronos (*lazy*), primeiro é feita a atualização da cópia local, confirmado ao cliente e somente depois é feita a propagação. O primeiro provê consistência numa maneira natural, mas é caro em relação a custo de troca de mensagens e tempo de respostas. Replicação assíncrona permite várias otimizações, mas como as cópias podem ter dados divergentes, gerando uma série de inconsistências.

Em relação a quem pode fazer atualizações, o método de cópia primária (*primary copy*) requer que todas as atualizações sejam feitas primeiro em uma cópia e depois nas outras cópias, simplificando o controle de réplicas e colocando um ponto único de falha. O método de atualização em todo lugar[15] permite que cada cópia seja atualizada, aumentando a velocidade de acesso com o preço de uma coordenação mais complexa.

Replicação síncrona de cópia primária

Neste método, uma operação de atualização é primeiramente executada na cópia primária e depois propagada para as outras cópias. Quando o primário recebe a confirmação que as cópias secundárias foram atualizadas, é feita a confirmação (*commit*) e a então a notificação é retornada ao cliente. A ordenação das operações é feita pelo primário e deve ser obedecida pelos secundários. As operações de leitura podem ser feitas em qualquer lugar e sempre terão a última versão de cada objeto.

Quanto ao modelo funcional, a fase de coordenação dos servidores desaparece porque a operação somente é executada no primário. A fase de execução envolve executar a transação para gerar os registros de *log* que são enviados aos secundários e aplicados. Então o *2-phase commit* (2PC) é executado durante a fase de coordenação de decisão. Quando terminado, a resposta é enviada para o cliente.

Percebe-se que a replicação síncrona de cópia primária é equivalente à replicação passiva com VSCAST. As únicas diferenças são no coordenador de decisão, pois bancos de dados utilizam transações. No caso do VSCAST, há garantia de que as operações são ordenadas corretamente mesmo que uma falha ocorra. No 2PC garante-se que se o primário falhar, todas as transações serão abortadas.

Para o caso de transações com múltiplas operações, é feito um *loop* para cada operação entre a fase de execução e coordenação de decisão, onde cada operação é executada na cópia primária e as mudanças enviadas às réplicas. No final, uma nova fase de coordenação de decisão é executada para que seja feito o 2PC e a transação é confirmada ou abortada.

[15] Do Inglês: *update everywhere*

Replicação síncrona com atualização em todo lugar

Do ponto de vista funcional há dois tipos de protocolos a considerar, dependendo se eles usam *lock* distribuído ou *atomic broadcast* para ordenar operações conflitantes.

Lock* Distribuído*: Um item somente pode ser acessado depois de obter o *lock* em todas as réplicas. Para transações com uma operação, o cliente envia a requisição para a réplica local e esta réplica envia requisição de *lock* para todas as outras, que o concedem ou não. Quando todos os *locks* forem concedidos a operação é executada em todas elas. Na fase de coordenação de decisão, o protocolo 2PC é usado para garantir que todas as réplicas confirmaram a transação. Então, o cliente recebe a resposta.

Pode-se perceber que replicação síncrona com atualização em todo lugar com *lock* distribuído é similar à replicação semi-ativa. As diferenças são os mecanismos nas fases de coordenação de servidores e coordenação de decisão. Em bancos de dados, a coordenação de servidores usa *2-phase locking* e sistemas distribuídos ABCAST. O coordenador de decisão usa *2-phase commit* para bancos de dados e VSCAST para sistemas distribuídos.

Para o caso de múltiplas operações, o *lock* deve ser obtido para cada operação na transação, que requer repetição das fases de coordenação de servidores e execução de cada operação. No final, utiliza-se 2PC na fase de coordenação de decisão.

Atomic Broadcast: A idéia é usar a ordem total garantida pelas primitivas de comunicação em grupo para prover uma dica ao gerenciador de transações sobre como ordenar operações conflitantes, então a requisição é enviada a todos os servidores. Ao invés de usar *2-phase locking*, a coordenação dos servidores é feita usando ABCAST e algumas técnicas para se obter os *locks* de maneiras consistentes em todos os lugares. As cinco fases são:

- O cliente envia a requisição ao servidor local.
- O servidor encaminha a requisição a todos os servidores com *Atomic Broadcast.*
- Os servidores executam a transação.
- Não há coordenação de decisão.
- O servidor local envia a resposta ao cliente.

Para o caso de múltiplas operações, não faz sentido usar o ABCAST para enviar cada operação da transação separadamente. Para isso usam-se cópias *shadow* em uma réplica para executar as operações e, então, quando a transação é completada, enviam-se todas as mudanças

numa única mensagem. Com isso, a fase de coordenação de decisão fica mais complicada, pois envolve tomada de decisão para que as operações sejam executadas corretamente.

Os protocolos de replicação síncronos são usados para manter a consistência dos dados entre as várias réplicas, porém, à medida que o número de nós aumenta, também há aumento no tempo de resposta para as transações com o aumento da probabilidade de conflito das operações iniciadas em cada uma das réplicas e o aumento da taxa de *deadlock*. Mesmo com o uso de primitivas de comunicação em grupo, segundo Kemme e Alonso [9], os protocolos podem ser muito simples a ponto de ter altas taxas de *abort* ou os protocolos serem muito complexos para serem implementados num banco de dados real.

Replicação assíncrona (*lazy*)

Replicação assíncrona evita o *overhead* de sincronização da síncrona enviando a resposta ao cliente antes que haja qualquer coordenação entre os servidores. Também pode ser de cópia primária ou atualização em todo lugar. No caso de cópia primária, todos os clientes devem entrar em contato com o mesmo servidor para executar atualizações e na atualização em todo lugar qualquer servidor pode ser acessado. Após a execução o servidor local envia a resposta ao cliente e somente após o *commit* as atualizações são enviadas para as réplicas. No caso de cópia primária a fase de coordenação de decisão é relativamente simples, pois toda a ordenação é feita no servidor primário. No caso de atualização em todo lugar a coordenação é mais complicada, pois todos os servidores podem estar executando transações ao mesmo tempo e as cópias podem estar inconsistentes. É necessário reconciliação para decidir quais atualizações são as vencedoras e quais transações serão desfeitas.

Como as mudanças são propagadas somente após o *commit*, não faz diferença para os protocolos se a transação possui uma ou mais operações, pois as mudanças são enviadas todas de uma vez.

Existem vários estudos que exploram estratégias assíncronas que ainda conseguem prover consistência, o que é fundamental em bancos de dados. O problema é que essas alternativas são de cópia primária e somente uma réplica pode ser usada para fazer atualizações.

Uso de replicação síncrona em sistemas reais

Como a replicação síncrona padrão sofre pelo desempenho e a assíncrona sofre pela consistência, Bettina Kemme e Gustavo Alonso [9] propuseram um protocolo de replicação

síncrono que melhora o desempenho e pode ser usado na prática. Este protocolo foi implementado em cima do banco de dados PostgreSQL e chamado de Postgres-r. A seguir serão mostradas as formas utilizadas para fazer esta otimização.

Redução do *overhead* de mensagens e sincronização: Protocolos tradicionais de replicação síncrona executam uma operação por vez. Essa alternativa não pode ser escalável. Uma maneira de evitar esse problema é agrupar todas as escritas numa única mensagem com um conjunto de escrita[16]. Em replicação assíncrona isso é fácil porque as atualizações são propagadas após o *commit*. Para replicação síncrona, usa-se *shadow copies* para fazer as atualizações, onde são feitas as escritas, verificação de consistência, captura de dependências entre leitura e escrita e execução de *triggers*. Essas mudanças são então propagadas para as outras réplicas no *commit*, reduzindo o *overhead* de mensagens e o perfil de conflito das transações.

Operações de leitura locais: As leituras são executadas somente na réplica local e nenhuma informação sobre ela é propagada para as outras réplicas. Portanto, não tem custo de mensagens e não tem *overhead* nos locais remotos, o que é muito desejável. Porém, introduz alguma complexidade relacionada com conflitos de leitura e escrita, uma vez que as leituras são vistas somente localmente. A solução para esses conflitos é mostrada no protocolo proposto, descrito adiante.

Transações pré-ordenadas: Primitivas de comunicação de grupo são usadas para prover semântica de ordem total para fazer o *multicast* do conjunto de escritas e para determinar a ordem de seriação das transações. A ordem total garante que todas as réplicas recebam o conjunto de escritas exatamente na mesma ordem. Cada gerenciador de transação usa esta ordem para adquirir os *locks* e em seguida faz a execução da transação. Garantindo os *locks* na ordem que as transações chegam garante que todas as réplicas executem atualizações conflitantes na mesma ordem. Além disso, as transações nunca ficam em *deadlock*. Note que isso não significa execução serial, pois transações não conflitantes são executadas em paralelo. A máquina local pode fazer *commit* de uma transação uma vez que a ordem de seriação seja determinada, confiando no fato de que os outros lugares irão serializar a transação da mesma maneira.

Protocolo síncrono proposto: O protocolo proposto executa a transação em quatro fases diferentes:

---

[16] Do inglês: *write set*.

- *Fase de leitura local:* Faz as leituras localmente, executa as escritas em *shadow copies* e adquire os *locks* antes de executar a operação.
- *Fase de envio* (send)*:* Se a transação for só de leitura, faz *commit*. Senão coloca todas as escritas num conjunto de escritas e faz o *multicast* para todas as réplicas, incluindo a si própria.
- *Fase de* lock*:* Na entrega da mensagem, requisita todos os *locks* numa única vez:

  Para cada operação:
    - Faz um teste de conflito: se uma transação local tem o *lock* e ainda está na fase de leitura ou envio, a aborta. Se estiver na fase de envio, faz o *multicast* da mensagem de decisão de abortar.
    - Se não tem nenhum lock, adquire o lock. Senão, enfilera a requisição de *lock* diretamente depois todos os *locks* das transações que estão além de suas fases de *lock*.
    - Se é uma transação local, faz o *multicast* da mensagem de decisão de *commit*.
- *Fase de escrita:* Uma vez que o *lock* de escrita seja adquirido, aplica a atualização correspondente. Uma transação local pode fazer *commit* e liberar os *locks*. Uma transação remota deve esperar pela chegada da mensagem de decisão e terminar de acordo.

Neste protocolo, ordem total é usada para seriar os conflitos escrita/escrita em todos os lugares. Conflitos leitura/escrita são também detectados na fase de *lock*. Uma vez que as operações de leitura somente são vistas localmente, usa-se a solução de abortar as leituras locais que conflitam com a escrita que chegou. Isso evita *deadlock* e execuções inconsistentes. Quando uma transação é abortada durante a fase de leitura, ela é completamente local e nenhuma mensagem precisa ser trocada. Quando é abortada na fase de envio, o processo local deve informar aos outros através uma mensagem de *abort*. Portanto, este protocolo requer que o processo local envie duas mensagens por transação, uma com o conjunto de escrita e outra para confirmar que a transação será confirmada ou abortada.

3.6 Sumário

Este capítulo mostrou um modelo para replicação e um estudo de protocolos de replicação para sistemas distribuídos e para bancos de dados, utilizando este modelo. Mostrou que a

replicação em bancos de dados pode ser síncrona, que garante consistência, mas tem baixo desempenho, e assíncrona, que tem melhor desempenho, mas não garante consistência. Além disso, discute também um protocolo para permitir bom desempenho mesmo usando replicação síncrona.

O bom desempenho obtido pelo algorítmo proposto por Kemme e Alonso[9] se deve a duas características importantes: (i) uso reduzido de escrita em disco porque somente o *log* das operações executadas é escrito em disco e (ii) redução do número de mensagens de coordenação (*commit*). Vemos então que protocolos como esse ou talvez novas adaptações das diferentes técnicas propostas para bancos de dados tradicionais podem ser utilizadas para a replicação de MMDBs. É por isso que este Capítulo foi incluído neste trabalho, para (i) mostrar que é possível realizar replicação de bancos de dados utilizando protocolos derivados dos protocolos gerados para sistemas distribuídos e (ii) que esses protocolos podem ser adaptados para uso em MMDBs.

No próximo capítulo estuda-se o banco de dados Monet, que foi implementando usando-se técnicas de bancos de dados em memória principal para a melhoria de desempenho.

Capítulo 4

Monet

Uma das razões para a escolha do sistema Monet para a realização do estudo comparativo apresentado nessa dissertação é a similaridade entre as características do Monet e as características desejáveis em um banco de dados para a sua replicação. Segundo o seu arquiteto principal, Dr. Peter Boncz[2], o Monet foi um SGBD de memória principal construído para explorar ao máximo os ganhos em velocidade, latência e capacidade oferecidos pelos computadores atuais para a construção de aplicações como OLAP e *data mining*.

Este trabalho visa a verificação experimental das características do Monet. Através do estudo comparativo com MySQL e PostgreSQL será possível verificar se ele tem na prática características de desempenho que o tornam adequado à replicação.

A seguir serão mostrados os requisitos e a forma com que o Monet foi construído para tentar atingir essas metas.

4.1 Requisitos

Obter o melhor desempenho das modernas CPUs e memórias nas aplicações de SGBD com consultas massivas: Pesquisas passadas eram focadas em minimizar a quantidade de I/O aleatórios e otimizações de CPU e memória eram colocados em segundo lugar. O projeto do Monet quis ter uma arquitetura que permitisse a utilização de todo o poder das novas CPUs e a otimização do tráfego na memória principal. As aplicações alvo são aplicações de consultas massivas. A primeira meta é alto desempenho, sendo projetado para explorar execução paralela.

Suporte a modelos de dados múltiplos: O modelo relacional e a linguagem SQL são atualmente o mais popular nos SGDBs, e outros modelos, como orientados a objeto e objeto-relacional, estão se tornando popular. O sistema deve ser usado por ambos os modelos ou até por

linguagens que não sejam o SQL. Então, o projeto do Monet foi feito de um modo a prover todas as necessidades do SGBD, mas de uma maneira neutra ao que o usuário vê como modelo de dados e linguagem de consulta.

Adaptabilidade: embora as aplicações de consultas massivas sejam OLAP e *data mining*, elas não estão limitadas somente ao domínio dos negócios, então o Monet leva em conta outros domínios em seu projeto e está sendo usado por pesquisadores em vários novos domínios.

4.2 Arquitetura de *software* do Monet

4.2.1 Back-end e front-end separados

A arquitetura de *software* do Monet agrupa os seus sub-sistemas em dois sub-sistemas principais: um *front-end* e um *back-end*. A arquitetura de *software* utilizada no Monet é mostrada na Figura 3. Em comparação à arquitetura de um SGBD convencional, mostrada na Figura 1, temos que no Monet os serviços do processador da linguagem de consulta, reescrevedor de consulta, e otimizador de consulta são responsabilidades do *front-end*. No *back-end* ficam os serviços de execução de consultas, métodos de acesso e gerenciamento de recuperação. O gerenciador de transação não é implementado da mesma maneira no Monet, pois o *front-end* fica responsável por solicitar os *locks* necessários ao *back-end*. E o gerenciador de *buffer* não é implementado, pois torna-se desnecessário num SGBD de memória principal.

O Monet é basicamente a implementação do *back-end* do SGBD. Com isto, ele é independente para que diferentes *front-ends* possam ser usados com o mesmo *back-end*. Por exemplo, podemos ter um *front-end* relacional recebendo requisições em SQL e ao mesmo tempo um *front-end* orientado a objetos recebendo consultas OQL e também *front-ends* para OLAP e *data-mining*.

Para que o *front-end* pudesse se comunicar com o *back-end*, uma linguagem intermediária entre o *front-end* e o *back-end* foi criada. Esta linguagem provê um conjunto mínimo de primitivas e é conhecida como *Monet Interpreter Language* (MIL).

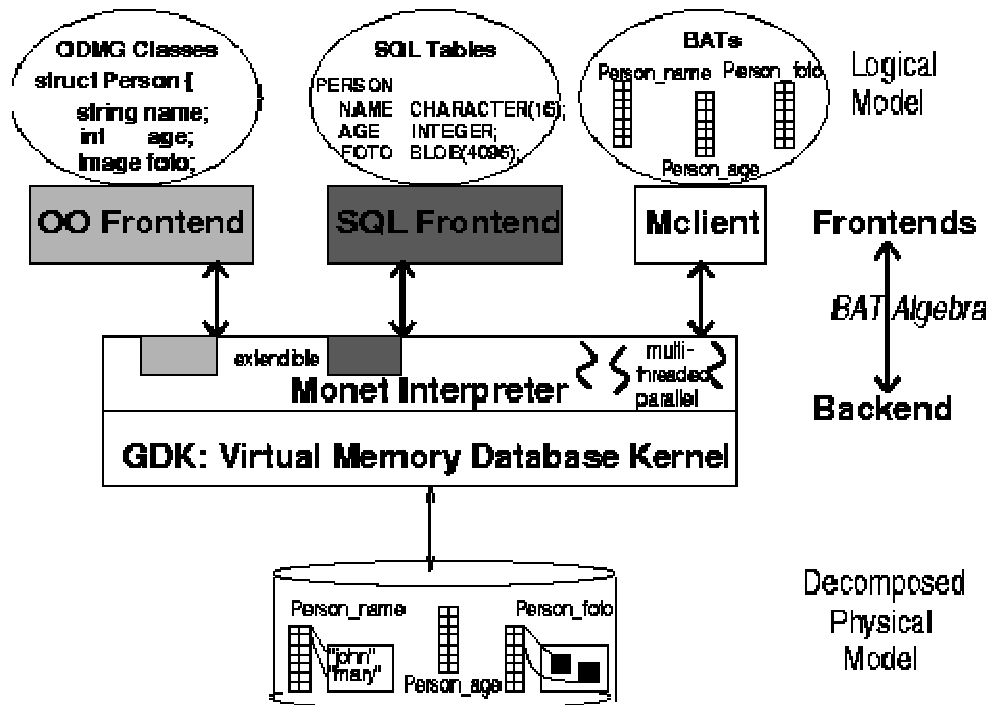

Figura 3 – Arquitetura do Monet

Fonte: Tese de Doutorado de Peter Boncz [6]

Na linguagem MIL, as facilidades de gerenciamento de transações são explicitamente separadas das facilidades de processamento de consulta. A linguagem contém comandos específicos para transação que provê blocos para construir transações ACID, ao invés de implementar um protocolo de gerenciamento de transação *hard-coded*. Fica a cargo do *front-end* usar estas primitivas de transação para implementar seu próprio protocolo de transação. As primitivas de processamento de consulta supõem que todos os *locks* para as respectivas consultas já foram pegos e a consulta é executada sem nenhum *overhead* de transação. Isto facilita implementações que não precisam de um controle rigoroso, como OLAP e *data mining*, que podem usar um protocolo simples de *locking*, com potencial ganho de desempenho.

Como o alvo é o uso em aplicações de consultas massivas, a ênfase nas funcionalidades do Monet é em execução de consultas. A execução de consultas no Monet inclui algumas tarefas que normalmente são feitas pelo otimizador de consultas; a otimização da consulta é dividida em fase estratégica e tática. A estratégica é a feita pelo *front-end* e a tática é feita pelo Monet em tempo de execução para encontrar o melhor algoritmo e configuração do gerenciamento de *buffer*. Este projeto leva em conta as características das tabelas somente em tempo de execução, o que melhora a qualidade das decisões de otimização tomadas.

4.2.2 Armazenamento de dados em tabelas binárias

Como o Monet utiliza memória principal para executar suas consultas, uma forma de otimizar o acesso aos dados é utilizar fragmentação vertical total dos dados, fazendo o uso de tabelas binárias, que consistem em tabelas com exatamente duas colunas, que são chamadas *Binary Association Tables* (BATs). Cada *front-end* usa regras de mapeamento para mapear o modelo lógico visto pelo usuário nas tabelas binárias do Monet, sendo possível armazenar uma diversidade de modelos lógicos, tendo um papel importante em atingir as metas de suporte a vários modelos e extensibilidade para novos domínios.

No modelo relacional, as tabelas são otimizadas de forma com que cada coluna seja armazenada num BAT diferente. A coluna da direita desses BATs tem o valor da coluna relacional e o da esquerda tem a chave da linha ou objeto, conhecido como OID. No caso de OIDs crescentes com valores muito altos o Monet possui suporte para OIDs virtuais (VOID), em que somente o OID base é armazenado e desta forma as tabelas binárias do Monet são quase sempre tabelas unárias, implementadas como vetores, em que a busca por um OID é tão simples e barata quanto indexar um vetor.

Aplicações de consultas massivas possuem um padrão de acesso voltado a ler poucas colunas. Tem a vantagem que consultas que usem muitas linhas, mas poucas colunas, somente precisam varrer os BATs das colunas que necessitem. Com isto, se reduz o I/O e banda de memória gerada por consultas OLAP e *data mining*, reduzindo o custo de acesso. A fragmentação vertical é explícita no modelo lógico de dados e, com isto, os algoritmos puderam ser otimizados para trabalhar bem com dados fragmentados e ter certeza que não só os dados principais estão fragmentados como também os resultados intermediários.

4.2.3 Não reinventar o sistema operacional

O Monet utiliza as facilidades providas pelos sistemas operacionais modernos, sem duplicação de esforços entre o SGBD e o SO, o que pode muitas vezes ser feito em alguns bancos de dados relacionais.

Algumas facilidades do SO utilizadas pelo Monet são:

- ***Multi-threading*** **e** ***scheduling*****:** em cargas OLTP, é importante que os *locks* sejam feitos pelo menor tempo possível, para favorecer a execução simultânea de várias consultas. Uma causa específica para segurar o *lock* por um grande tempo é o processo ser colocado

em espera logo após conseguir o *lock*. Para prevenir isto, o processo de *scheduling* do SO precisa saber das *threads* de controle no processo, suas primitivas de sincronização e permitir que o SGBD influencie nas prioridades das *threads*. Esse problema foi resolvido no padrão POSIX *pthreads*. O Windows provê facilidades similares.

- **I/O de arquivo:** Os sistemas de arquivo não provêem facilidades para suportar o conceito de transações atômicas. Além disso, a granularidade mínima de I/O costuma ser de tamanho fixo, não se adaptando às necessidades do SGBD. Essas deficiências ainda existem nos sistemas modernos e acesso direto ao disco pode prover melhor consistência e desempenho. Entretanto, esses problemas envolvem OLTP e seus acessos aleatórios. Para aplicações de consultas massivas o uso de grande volume de dados faz com que a grande demanda seja a banda (*trhouput)*, o que é implementado satisfatoriamente no SO.
- ***Buffering* de memória virtual para acesso aleatório:** o sistema UNIX *mmap* permite acessar dados no disco usando a interface de memória, funcionando da seguinte forma: o SO retorna uma faixa de memória que representa o arquivo armazenado no disco. Acessos a essa faixa de memória causam *page fault* e o SO lê os dados necessários do disco, de forma transparente.

4.2.4 Otimização para execução de consultas em memória principal

Como o tamanho das memórias disponíveis nos sistemas atuais tem aumentado muito, uma forma de aumentar o desempenho de execução de consultas em bancos de dados é executar essas consultas em memória principal.

Essa orientação em direção à execução em memória principal é percebida em várias áreas da arquitetura de *software* do Monet:

Estruturas de dados básicas

O Monet armazena os dados em BATs, que basicamente é um vetor de memória com registros de tamanhos fixos. Esta estrutura otimiza o uso do cache de memória em leitura seqüenciais e não há espaço perdido. Quanto às estruturas auxiliares, usam-se resultados passados em métodos de índice de bancos de dados de memória principal, que identificou árvores balanceadas e *hashes* simples como tendo o melhor desempenho.

Gerenciamento de *buffer*

No projeto do Monet foi tomado cuidado para que facilidades que só são necessárias para consultas OLTP não prejudicassem o desempenho de aplicações de consultas massivas. Isto se aplica ao gerenciamento de transação e de *buffer*. Produtos SGBDs relacionais costumam ser centrados no conceito de uso de blocos ou páginas de disco e o trabalho principal é feito pelo componente de gerenciamento de *buffer*, usando algoritmos que tentam usar o mínimo possível de I/O. Aplicações de consultas massivas são diferentes, pois usam muita banda de leitura seqüencial e os desafios de otimização estão no acesso à memória e utilização de CPU. Deste ponto de vista, muito do gerenciamento de *buffer* relacional só serve para OLTP e pode atrapalhar OLAP. O gerenciamento de *buffer* no Monet é feito ao nível de um BAT (ele é todo lido ou não é lido), para eliminar o gerenciamento de *buffer* como fonte de *overhead*. No caso de conjuntos grandes de dados não é possível trazer o BAT inteiro na memória, onde o gerenciamento de *buffer* se resume a fragmentação horizontal, trazendo fragmentos do BAT por vez.

Algoritmos de processamento de consultas

Algoritmos de processamento de consultas tradicionais supunham que acesso à memória principal é barato e o custo de acesso à memória é uniforme. Isto não é mais verdade. Nas CPUs mais modernas a latência de acesso é de vários ciclos de CPU. Portanto os algoritmos de processamento devem levar em conta o gargalo de acesso à memória. O efeito desse gargalo é mostrado a seguir para operadores comuns em aplicações de consultas massivas:

- *Seleções:* em OLAP e *data mining*, a seletividade é normalmente baixa, o que significa que o dado deve ser visitado e o melhor a fazer é uma varredura. Acesso seqüencial otimiza o uso do sistema de memória e evita problemas.
- *Agrupamento e agregação:* dois algoritmos são usados aqui: *sort/merge* e *hash-join*. No *sort/merge* primeiro temos a tabela ordenada pelo atributo GROUP-BY seguido pela seleção. *Hash-groupings* varre primeiro, mantém uma tabela temporária onde os valores de GROUP-BY são as chaves numa *hash-table* temporária. Esse número de grupos é normalmente limitado, fazendo que o *hash-grouping* seja superior ao *sort/merge* em relação ao acesso à memória; o *sort* usa acesso aleatório e é feito na seleção inteira a ser agrupada, e provavelmente não cabe em nenhum cache.

- *Equi-joins: hash-join* é o preferido para sistemas em memória principal, uma vez que ele constrói uma *hash-table* na relação menor e a outra relação é varrida. Se a relação menor não couber na memória cache um problema de desempenho pode ocorrer, devido ao padrão de acesso aleatório. *Merge-join* não é uma alternativa viável porque requer ordenação nas duas relações primeiro, gerando acessos aleatórios em grandes áreas de memória.

Conseqüentemente, identifica-se *equi-join* como o operador mais problemático em relação a acesso a memória e um novo algoritmo foi definido para otimizar o padrão de acesso à memória.

No Monet, que concentra a execução em memória principal, o *tunning* deve ser feito no acesso à memória. O I/O é feito com a ajuda do SO para mapear grandes BATs em disco.

Levando-se em conta que os dados a serem processados estão em memória principal, o desenvolvimento do Monet foi focado em utilizar técnicas que fazem o compilador crie um código que é eficiente em relação ao acesso à memória e desempenho de CPU. Por exemplo, utiliza-se uma técnica que evita chamadas de função em *loops* internos de rotinas críticas para desempenho, diminuindo os ciclos de CPU e de acesso à memória.

Com a arquitetura descrita o Monet pretende ser um sistema de alto desempenho em aplicações de consultas massivas otimizando o acesso aos recursos disponíveis.

4.3 Sumário

Este capítulo fez uma análise da estrutura e das principais características do SGBD Monet. Este SGBD foi escolhido porque tem uma estrutura que, em tese, permite sua adaptação para replicação, através de (i) uso de *back-end* e *front-end* separados, possibilitando alterações somente no *front-end* para a inclusão da camada de replicação; (ii) uso da linguagem MIL para processamento de consultas no *back-end*, de forma que um novo *front-end* poderia ser construído especificamente para o uso replicado e (iii) suporte a pouco I/O, pois as consultas são feitas em memória principal e podem, em tese, ser executas mais rapidamente.

Para verificar se realmente essa estrutura leva a uma melhora de desempenho em aplicações *web*, é necessário a execução de um *benchmark* e comparar o desempenho com outros bancos tradicionais. O capítulo a seguir faz a análise do *benchmark* escolhido para este experimento.

Capítulo 5

TPC-W

Para termos uma medida confiável do desempenho de SGBDs quando utilizados em aplicações *web* devemos procurar emular o comportamento dessas aplicações sob condições controladas de execução e monitoração. Alguns *benchmarks* são utilizados por vários fabricantes para medir o desempenho de seus servidores de comércio eletrônico, como SPECWeb e TPC-C. Porém, segundo Wayne Smith [11], esses *benchmarks* não representam corretamente o ambiente complexo de carga de uso em comércio eletrônico.

Uma solução para este problema resultou na especificação TPC-W [14], que determina uma carga de uso de um sistema de comércio eletrônico de modo a medir o desempenho deste sistema como um todo.

Este capítulo mostra a especificação TPC-W, a implementação usada para os testes e as adaptações que foram necessárias.

5.1 Especificação TPC-W

O TPC-W especifica uma carga que simula as atividades de uma loja de livros, que pode ser considerada uma aplicação típica de comércio eletrônico. Nesta loja, os clientes, que são representados por *browsers* emulados, podem visitar as páginas da loja, fazer buscas, procurar por produtos e realizar compras.

A especificação contém os requisitos para que esta loja seja implementada, definindo como são as páginas de acesso, como e que tamanho devem ter as imagens dessas páginas, a estrutura das tabelas do banco de dados e, para que a medida de desempenho possa ser feita, define como o acesso a essas aplicações deve ser feito, através de *browsers* emulados (*Remote Browser Emulator, RBE*), os quais navegam automaticamente através das páginas, usando um critério

estatístico e gera as métricas colhidas durante sua execução. A navegação entre as páginas é aleatória, porém a seleção das páginas está sujeita à distribuição encontrada normalmente para esse tipo de aplicação.

Para efeito de testes, é possível navegar nas páginas desta loja virtual com um *browser* normal. Teremos a página inicial, com imagens e sugestões de livros, possibilidade de fazer buscas, colocar itens no carrinho, fechar a compra, entrar com dados de cartão de crédito e muitas outras operações presentes numa loja virtual. Percebe-se que a implementação da especificação TPC-W é muito próxima da implementação que um *site* de comércio eletrônico utiliza. A especificação, inclusive, determina as páginas que devem ter conexão segura, através de SSL e emulação da validação do cartão de crédito com a operadora do cartão.

5.1.1 Métricas geradas

Várias métricas podem ser geradas durante a execução dos testes. A principal métrica é a quantidade de interações executadas por segundo pelo servidor da aplicação, conhecida na especificação como WIPS (*Web Interactions Per Second*). Essas interações levam em conta o acesso global às páginas *web*, portanto, é incluído como parte das interações o acesso às imagens, acesso às páginas e o acesso ao banco de dados.

Para que as métricas fiquem mais próximas da realidade, os *browsers* emulados devem aguardar um tempo de espera (*think time*) entre uma página e outra, para simular o tempo que o usuário leva para ler a página e tomar uma decisão. Este tempo tem uma média de 7 segundos e um máximo de 70 segundos.

É especificado que o WIPS resultante esteja entre (Número de *browsers* / 14) e (Número de *browsers* / 7). Como o tempo de espera tem uma média de 7 segundos, espera-se que o WIPS seja, numa configuração balanceada, (Número de *browsers / 7*). A restrição inferior existe para que o teste não reporte valores errados para bancos de dados muito grandes.

O WIPS leva em conta o perfil de acesso à loja virtual. Supõe-se que 80% dos acessos são feitos a páginas basicamente de navegação, como a página principal, novos produtos, mais vendidos e páginas de busca. Os outros 20% são páginas usadas para compra de produtos, como o carrinho de compras, pedido, fechamento da compra e páginas de administração.

Duas outras métricas auxiliares são especificadas: WIPSb e WIPSo.

WIPSb leva em conta que 95% das páginas acessadas são páginas de navegação, onde o usuário entra na loja apenas para navegar, fazer buscas, olhar os produtos e muito raramente um cadastro é feito ou um produto é comprado. A maior parte da carga fica no servidor *web*, servidor de imagens e caches. A maior parte dos acessos ao banco de dados é acesso de leitura. Este perfil de uso é chamado na especificação de *browsing*.

WIPSo leva em conta que 50% dos acessos são feitos a páginas de compra e administração, simulando o usuário que pouco navega no *site*, e vai direto às compras. A maior parte carga é colocada no banco de dados, pois temos muitos acessos de escrita. Este perfil de uso é chamado de *ordering*.

O perfil de uso padrão, onde o WIPS é calculado, é chamado na especificação de *shopping*.

5.1.2 Solução típica de *hardware* e *software* para TPC-W

A especificação TPC-W não diz que configuração de *hardware* ou topologia de rede a ser utilizada para a realização das medidas. Porém, um sistema típico utilizado em aplicações web é mostrado na Figura 4.

Cada RBE representa um *browser* emulado que se conecta ao sistema em teste para a requisição das páginas.

O sistema sob teste consiste em um servidor *web*, um servidor de cache, um servidor de imagens e um servidor de banco de dados. O servidor de cache pode guardar resultados de buscas por autor e título durante todo o tempo que o sistema estiver no ar, porém resultados de busca de mais vendidos, novos produtos e assunto somente podem ser guardados por 30 segundos.

Fora deste sistema está o servidor de pagamentos, que faz a emulação da aprovação da compra pela operadora do cartão de crédito.

As métricas consideram somente o desempenho do sistema sob teste, representando o desempenho do sistema como um todo, e não de cada servidor individualmente.

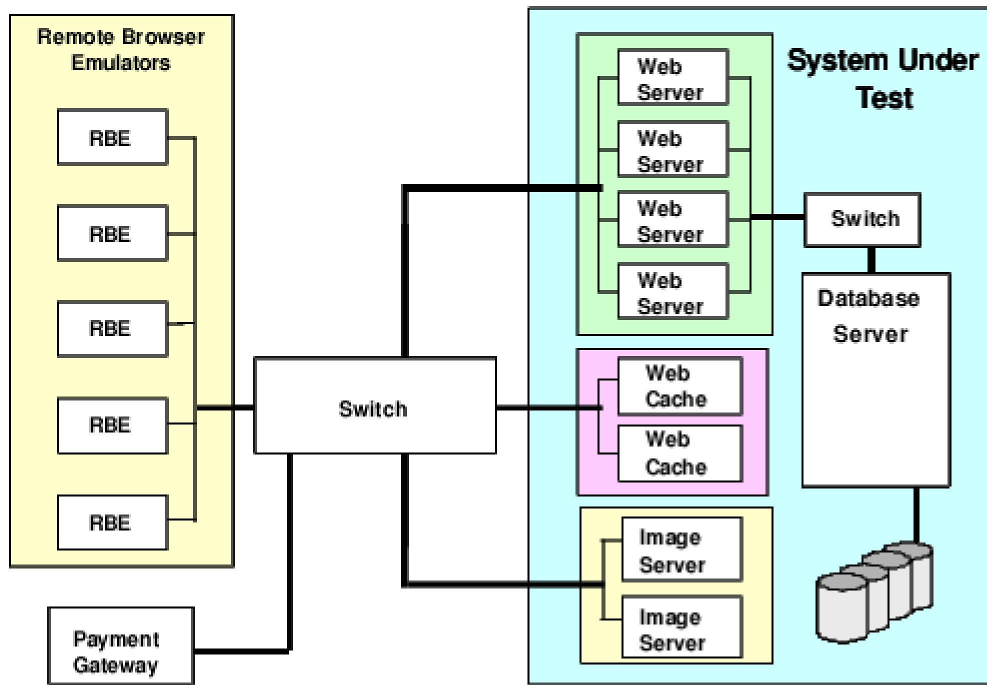

Figura 4 – Arquitetura típica de *hardware* e *software* para TPC-W.

Fonte: Especificação TPC-W [14]

5.1.3 Projeto lógico dos dados

A especificação fornece a forma com que os dados devem ser armazenados no banco de dados, através de quais tabelas devem ser geradas, qual o relacionamento entre elas e o tipo dos dados. A Figura 5 mostra o diagrama Entidade Relacionamento desse banco de dados.

Usando um banco de dados com especificação padrão consegue-se garantir que as medidas realizadas entre diferentes SGBDs sejam comparáveis.

É permitido fazer otimizações específicas para o SGBD em teste, porém elas são restritas e se referem, por exemplo, a particionamento horizontal e vertical de tabelas, uso de *clusters* e replicação, desde que explicitados no relatório final do desempenho. Mudanças que peçam a alteração da estrutura do modelo de dados não são permitidas.

Cada um dos campos, formatos de dados e tamanhos também são descritos na especificação.

As imagens utilizadas também possuem regras específicas de tamanho e o formato deve ser gif ou jpg.

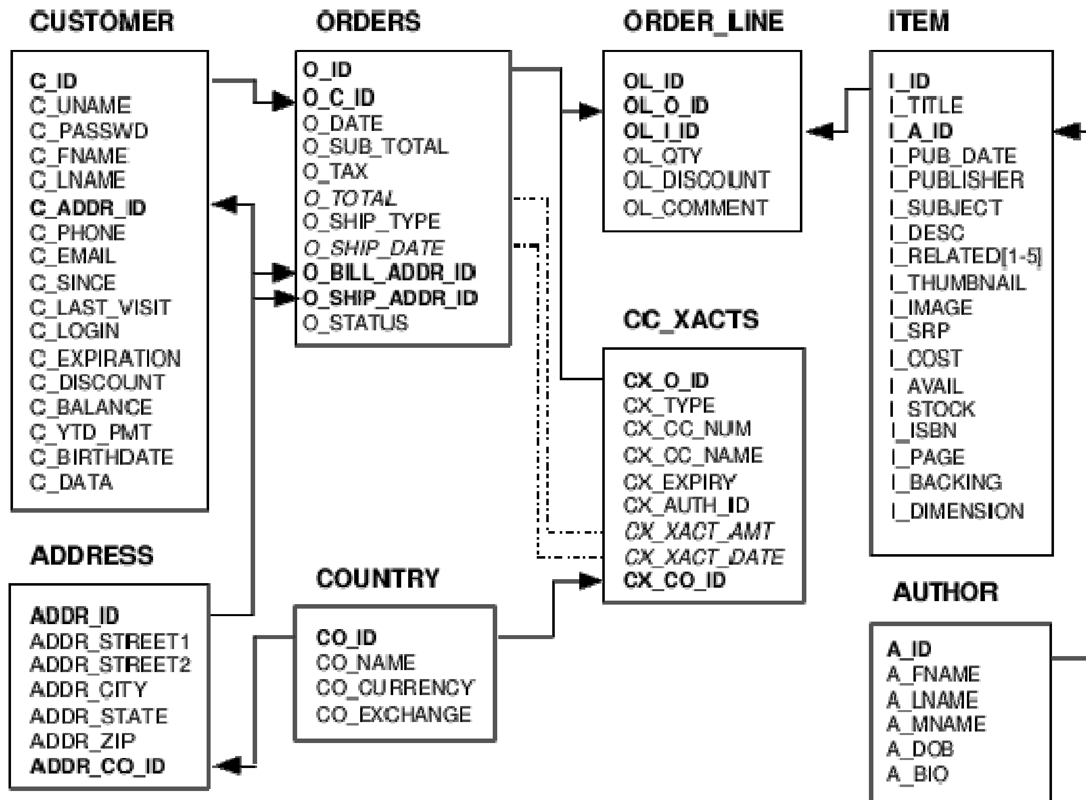

Figura 5 – Diagrama Entidade-Relacionamento do banco de dados de teste

Fonte: Especificação TPC-W [14]

5.1.4 Preenchimento do banco de dados

As regras para o preenchimento inicial do banco de dados também fazem parte da especificação. Os valores para a maioria dos campos são gerados aleatoriamente, porém algumas regras devem ser seguidas, como valores máximos e mínimos. Campos do tipo *string* são preenchidos gerando-se valores numéricos aleatórios e transformando-os em sílabas. As sílabas são concatenadas formando a *string* com o tamanho desejado. A tabela a seguir mostra a correspondência entre cada dígito do número aleatório (que deve ter 5 dígitos) e as sílabas:

Dígito	0	1	2	3	4	5	6	7	8	9
Sílaba	BA	OG	AL	RI	RE	SE	AT	UL	IN	NG

Os campos da tabela *COUNTRY,* ao contrário dos outros, possuem valores fixos com o nome dos países em inglês.

A quantidade de dados presente em cada tabela do banco de dados também é definida. Para que os dados sejam comparáveis entre diferentes execuções, o fator de escala deve ser o mesmo. Esse fator de escala é a quantidade de produtos presentes no banco de dados, que possuem seus possíveis valores definidos. Também temos vários dados que são dependentes do número de *browsers* emulados rodando simultaneamente e do fator de escala. A seguir são listadas as quantidades de dados em cada tabela do banco:

Nome da tabela	**Cardinalidade (em linhas)**
CUSTOMER	2880 * no. de *browsers*
COUNTRY	92
ADDRESS	2 * CUSTOMER
ORDERS	.9 * CUSTOMER
ORDER_LINE	3 * ORDERS
AUTHOR	.25 * ITEM
CC_XACTS	1 * ORDERS
ITEM	1k, 10k, 100k, 1M, 10M

5.1.5 Regras de interação e navegação

A especificação TPC-W contém as regras para a formação das páginas *web*, os algoritmos utilizados para mostrar cada um dos campos e regras e limitações a serem seguidas pela implementação.

São fornecidos requisitos de entrada, definições de processamento, definições da página de resposta e opções de navegação do *browser* emulado, além de regras para formação do *log* do servidor.

Os Requisitos de Entrada definem o conjunto mínimo de dados que deve ser usado como entrada da interação *web*. Pode-se aumentar o conjunto de dados, caso a implementação deseje.

As definições de Processamento definem as regras lógicas de interação para cada uma das páginas. É especificada a lógica usada para montar a página solicitada a partir dos parâmetros de entrada.

As definições de saída contêm uma amostra da página de saída com o respectivo código HTML, que é fornecido no Apêndice da especificação. Podem-se fazer extensões a essa amostra, mas o conteúdo original e os objetos envolvidos devem ser mantidos. Acrescentar novos objetos,

além de mudanças de cor, fonte e fundo são permitidos. É proibido referenciar, no código HTML, elementos ativos usados internamente na implementação.

As Opções de Navegação definem as possíveis opções que o *browser* emulado tem para navegar. O fluxo de navegação entre as páginas é mostrado na Figura 6.

5.1.6 *Browsers* emulados e métricas

Os *browsers* emulados são os responsáveis pela execução do teste, gerando a carga necessária como se fossem vários usuários utilizando o sistema.

Para cada página que é lida, o *browser* emulado toma a decisão de qual é a próxima operação e a executa em seqüência. Para tomar essa decisão, escolhe um número aleatório de 1 a 9999 e de acordo com o número escolhido e o perfil de teste executado verifica numa tabela qual a próxima página. Esta tabela contém pesos para cada uma das páginas para que de acordo com o número aleatório escolhido seja feita a decisão para se respeitar a probabilidade de escolha de cada uma das páginas pelo usuário.

Estas tabelas existem para cada um dos perfis de utilização. Como visto anteriormente, no perfil de *browsing*, 95% das interações nas páginas de consulta e 5% das interações nas páginas em que uma compra é realmente efetuada ou um cadastro de novo usuário. No perfil de *shopping*, temos 80% e 20% respectivamente. E no perfil de *browsing*, 50% e 50%.

A probabilidade de acesso para cada página, de acordo com a especificação, são mostradas na Figura 7.

Para que essa probabilidade seja garantida, existe a tabela de navegação, com os pesos para cada página. A Figura 8 é um exemplo de tabela de navegação para o perfil de *shopping*. As tabelas para os outros dois perfis são descritas na especificação, porém não foram listadas aqui.

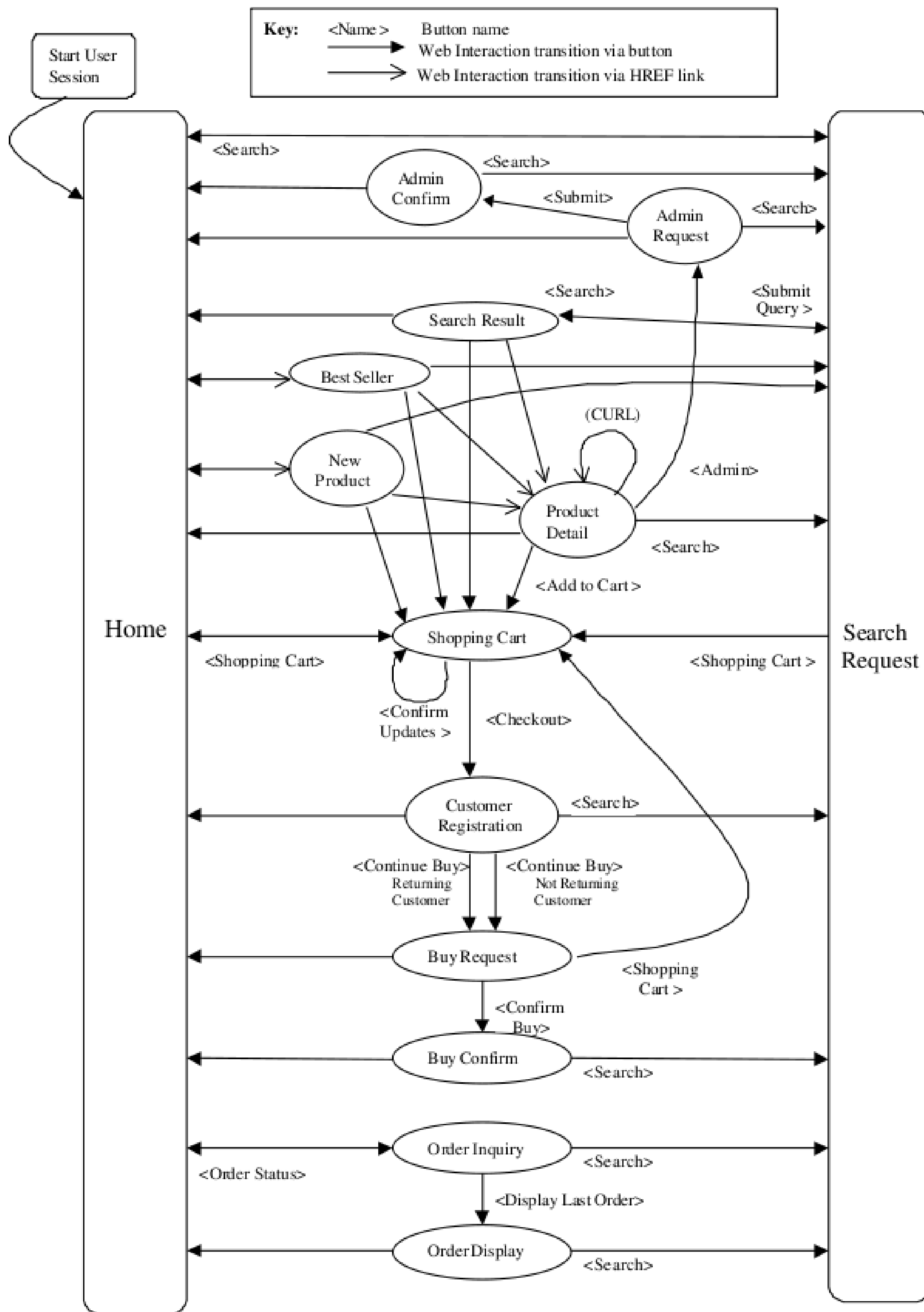

Figura 6 – Fluxo de navegação entre as páginas do TPC-W

Fonte: Especificação TPC-W [14]

Mix of Web Interactions

Web Interaction	Browsing Mix (WIPSb)	Shopping Mix (WIPS)	Ordering Mix (WIPSo)
Browse	95 %	80 %	50 %
Home	29.00 %	16.00 %	9.12 %
New Products	11.00 %	5.00 %	0.46 %
Best Sellers	11.00 %	5.00 %	0.46 %
Product Detail	21.00 %	17.00 %	12.35 %
Search Request	12.00 %	20.00 %	14.53 %
Search Results	11.00 %	17.00 %	13.08 %
Order	5 %	20 %	50 %
Shopping Cart	2.00 %	11.60 %	13.53 %
Customer Registration	0.82 %	3.00 %	12.86 %
Buy Request	0.75 %	2.60 %	12.73 %
Buy Confirm	0.69 %	1.20 %	10.18 %
Order Inquiry	0.30 %	0.75 %	0.25 %
Order Display	0.25 %	0.66 %	0.22 %
Admin Request	0.10 %	0.10 %	0.12 %
Admin Confirm	0.09 %	0.09 %	0.11 %

Figura 7 – Probabilidade de acesso a cada uma das páginas do TPC-W

Fonte: Especificação TPC-W [14]

A escolha é feita baseando-se na tabela e no número aleatório escolhido. Suponha que o *browser* emulado esteja na página de *Home*. Ele escolhe um número de 1 a 9999. Até 3124, a escolha será *Best Sellers*. De 3125 a 6249 a escolha será *New Products* e assim por diante. Com isto, garante-se a probabilidade mostrada na tabela anterior para cada uma das páginas.

Os *browsers* emulados são os responsáveis por gerar as métricas de execução dos testes, gerando o WIPS caso o perfil escolhido na execução seja *shopping*, o WIPSb, caso seja *browsing* e, para *ordering*, o WIPSo.

Thresholds for the Shopping Interval (WIPS)

To this Web Interaction ? From this Response Page ?	Admin Confirm	Admin Request	Best Sellers	Buy Confirm	Buy Request	Customer Regist.	Home	New Products	Order Display	Order Inquiry	Product Detail	Search Request	Search Results	Shopping Cart
Admin Confirm							9952					9999		
Admin Request	8999						9999							
Best Sellers							167				472	9927		9999
Buy Confirm							84					9999		
Buy Request				4614			6546							9999
Customer Regist.					8666		8760					9999		
Home			3124					6249		6718		7026		9999
New Products							156				9735	9784		9999
Order Display							69					9999		
Order Inquiry							72		8872			9999		
Product Detail		58					832				1288	8603		9999
Search Request							635						9135	9999
Search Results							2657				9294	9304		9999
Shopping Cart						2585	9552							9999

Figura 8 – Tabela com pesos de navegação para o perfil *shopping*.

Fonte: Especificação TPC-W [14]

5.2 Implementação TPC-W

Para a execução dos testes e cálculo das métricas, foi necessária a utilização de uma implementação da especificação TPC-W. Dada a sua complexidade não era factível criar uma implementação desde o início para o escopo deste trabalho. Portanto uma busca foi feita de forma a encontrar uma implementação que pudesse ser adaptada para o uso neste trabalho. A única implementação encontrada foi a implementação de TPC-W desenvolvida na Universidade de Winsconsin-Madison [15][17]. Esta implementação foi desenvolvida em Java e contêm as

[17] Além da implementação original, foram encontradas também algumas modificações. Para este trabalho foi optado por usar a implementação original com as adaptações necessárias.

implementações para popular os bancos de dados, geração das imagens, dos *servlets* que executam a interface *web* e dos *browsers* emulados.

Esta implementação foi originalmente escrita para rodar no banco de dados DB2, logo foram necessárias adaptações para garantir que a mesma fosse executada com sucesso nos bancos de dados escolhidos.

Foram feitos ajustes de *drivers* JDBC, modificações de formatos de datas e alguns tipos de dados e alterações em várias sentenças SQL de forma que as mesmas pudessem funcionar com os três SGBDs escolhidos. No final obteve-se uma implementação compatível com Monet, MySQL e PostgreSQL sem modificações. A implementação não foi desenvolvida para rodar no Tomcat, que foi o servidor de aplicação escolhido, portanto algumas alterações foram necessárias para que os *servlets* funcionassem neste ambiente.

A implementação revisada permite a execução de cada um dos três perfil do TPC-W, além de se poder especificar parâmetros como tempo de teste, tempo de *ramp-up*[18], tempo de *ramp-down*[19], endereço do servidor *web*, número de *browsers* simultâneos, entre outros parâmetros. Como saída esta implementação gera um arquivo no formato Matlab que contém as métricas pedidas pelo TPC-W. Como não tínhamos o software *Matlab* disponível para uso foi feita uma conversão de saída para que o gráfico de WIPS (WIPS, WIPSo ou WIPSb, dependendo do perfil escolhido no teste) fosse gerado pelo gnuplot.

5.3 Restrições

A implementação possui algumas restrições em relação à especificação.

Neste implementação não existe do emulador de pagamento, que simularia o acesso *online* às operadoras de cartão de crédito para aprovação da compra.

Também não existe cache das consultas, o que faz com que toda vez que uma consulta seja executada seja necessário consultar o banco de dados para obter a resposta.

O servidor de imagens é o mesmo servidor da aplicação, de forma que os recursos do servidor de aplicação são divididos entre a geração das páginas e o fornecimento de imagens.

---

[18] Tempo de espera antes que o *browser* emulado comece a executar os testes. Usado para esperar que o SGBD esteja num estado estável, sem requisições, antes de iniciar o teste.

[19] Tempo que deve ser mantida uma carga constante após o intervalo de medição.

Também não há suporte a SSL. Embora sistemas reais usem SSL quando uma compra é fechada, esta implementação simplifica sua implementação restringindo o uso de SSL.

Também foi preciso introduzir algumas outras restrições para a execução do experimento, além das restrições da aplicação apresentada.

Foi percebido durante o preparo do experimento, e confirmado pelo time de desenvolvimento do Monet, que ele não consegue lidar com várias transações simultâneas. Portanto, as transações foram serializadas, de forma que somente uma transação por vez pode ser executada. Mesmo nesta condição, também não foi possível o reuso de conexões JDBC com o banco de dados, o que implicou que para cada nova transação a conexão anterior fosse fechada e uma nova aberta. Para garantir a igualdade de condições entre todos os SGDBs, as mesmas limitações foram introduzidas na implementação utilizada para que tanto o MySQL quanto o PostgreSQL rodassem nas mesmas condições.

Também não foi seguido o requisito de geração de *log* pelo servidor *web*, de forma que o *log* gerado foi o *log* padrão do Tomcat.

Embora existam essas restrições, elas não influem na comparação de desempenho entre os diferentes SGBDs, pois todos eles serão executados utilizando as mesmas restrições, de forma a obter um ambiente comum que permita a comparação.

5.4 Sumário

Este capítulo mostrou que existe uma forma de medir o desempenho de aplicações *web* usando uma carga compatível com o perfil de uso desse tipo de aplicação através do *benchmark* TPC-W. Este *benchmark* é apresentado, mostrando as métricas a serem geradas, a forma de emular a carga gerada para as aplicações e as regras de navegação entre as páginas, a solução típica de *hardware* e *software* normalmente encontrada para aplicações *web*, o modelo de dados a ser utilizado no SGBD, a forma com que os dados são preenchidos.

A seguir, apresentou a implementação TPC-W utilizada neste experimento, incluindo as adaptações que foram necessárias para que fosse executada com os três SGBDs escolhidos e as restrições encontradas na aplicação e execução dos testes.

O próximo capítulo mostra a descrição do experimento utilizado na comparação dos SGBDs e os resultados obtidos na execução do teste e a análise desses resultados.

Capítulo 6

Experimento

Conforme descrito anteriormente, o *benchmark* TPC-W foi o escolhido para a execução dos testes e comparação dos resultados entre os três SGBDs escolhidos. Para que a comparação seja feita de forma de forma a fazer uma comparação entre o desempenho dos SGBDs no uso de aplicações *web*, o ambiente utilizado foi o mesmo em todos os experimentos e os testes rodados para cada um dos SGBDs. Esses testes foram feitos variando-se o tamanho do banco de dados e a carga de acesso de forma a obter o desempenho e compará-los em vários perfis de utilização.

Este capítulo detalha o experimento realizado e analisa os resultados obtidos.

6.1 Descrição do experimento

O experimento realizado consiste em rodar a implementação TPC-W descrita no capítulo anterior num ambiente que permita fazer comparações entre os SGBDs.

As versões de SGBDs utilizadas foram:

- Monet 4.10.2;
- MySQL 5.0.15-max;
- PostgreSQL 8.1.3;

Além disso, utilizou-se, como servidor *web* e de aplicação o Apache Tomcat 5.5.15.

O ambiente de *hardware* foi simplificado em relação ao ambiente típico para rodar TPC-W. Foram utilizados dois computadores para realizar o experimento: um para rodar os *browsers* emulados e o servidor *web* e de aplicação e outro para o servidor de banco de dados. Como o objetivo deste trabalho é realizar uma comparação do desempenho dos bancos de dados, o mesmo ficou isolado em um equipamento independente. Na tentativa de se utilizar várias máquinas de forma a deixar o servidor de aplicação, os *browsers* emulados e o SGBD tentamos utilizar o

laboratório guará, que contém um *cluster* de computadores. Porém, os equipamentos presentes neste laboratório tinham pouca memória e pouco espaço em disco, de forma que não foi possível utilizá-lo para o experimento. Portanto, utilizamos dois computadores do laboratório LSD para sua execução. A Figura 9 resume a arquitetura usada no experimento.

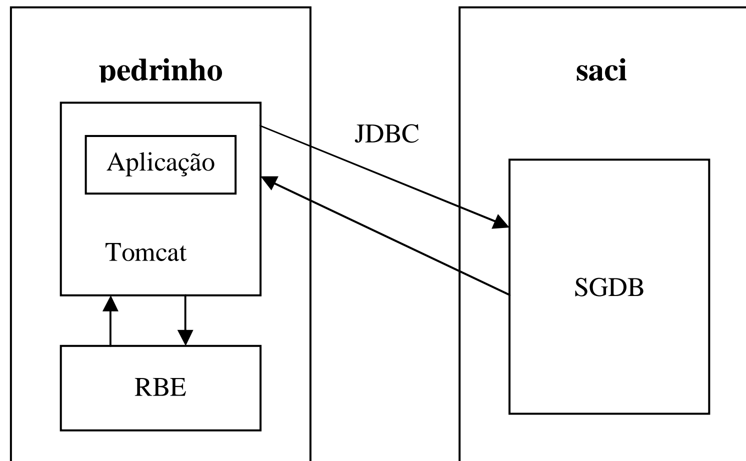

Figura 9 – Arquitetura de *hardware* e *software* utilizada no experimento.

A seguir tem-se a configuração de cada um dos computadores:

- **pedrinho:** Pentium 4 2.8 Ghz, 1Gb RAM, SO: Fedora Core 5;
- **saci:** 2 x Pentium 4 3 Ghz, 2Gb RAM, SO: Fedora Core 5.

A comunicação entre o Tomcat e os SGDBs foi feita utilizando-se JDBC e a comunicação entre os *browsers* e o servidor de aplicação foi feita utilizando-se HTTP sobre TPC/IP.

A configuração utilizada foi pensada de forma a comparar o desempenho dos três bancos de dados. Para que o desempenho dos SGDBs fossem comparados independente do ambiente, em todos os testes foram utilizados os mesmos equipamentos e os mesmos softwares, de forma que a única variável era o SGDB.

Para popular os bancos de dados, foram utilizadas as quantidades de dados descritos na especificação TPC-W, para as seguintes configurações:

- 30 *browsers*, 10000 items;
- 150 *browsers*, 10000 items;
- 300 *browsers*, 10000 items.

Para cada um dos SGDBs, foram executados diversos testes, variando a quantidade de *browsers* simultâneos e a quantidade de dados existentes no banco de dados.

Os bancos de dados eram populados com a quantidade de dados necessária e em seguida uma cópia de segurança era feita desses dados. Desta maneira, foi possível guardar cópias de segurança de todas os dados utilizados em cada um dos experimentos.

6.1.1 Testes realizados

Os testes foram executados sempre sob as mesmas condições iniciais. No início de cada teste, a cópia de segurança da quantidade de dados desejados para o respectivo SGDB era restaurada e o SGDB era inicializado em seguida. A seguir, o Tomcat era inicializado e em seguida os *browsers* emulados eram executados para começar as medidas.

No primeiro teste realizado variou-se a quantidade de browsers simultâneos num conjunto pequeno de dados (dados para 30 *browsers*), de forma que fosse medida a capacidade dos bancos de dados num conjunto que caberia na memória do servidor.

Em outro teste executado variou-se a quantidade de dados conforme variava-se a quantidade de browsers simultâneos, seguindo o que era requerido pela especificação do TPC-W. Somente para os testes de 600 *browsers* simultâneos foram utilizados dados para 300 *browsers*, pois foi encontrada uma dificuldade em popular o Monet com uma quantidade tão grande de dados.

Todos os testes foram executados nos três perfis de uso especificados no TPC-W, como *browsing*, *shopping* e *ordering*. Desta forma, foi possível medir o WIPSb, WIPS e WIPSo em todas as configurações.

A duração de cada um dos testes foi de 10 minutos, além de 1 minuto de *ramp-up time* e mais um minuto de *ramp-down time*.

Não foram levadas em conta as restrições de número de *browsers* em relação ao WIPS obtido, pois o intuito deste trabalho é verificar o comportamento dos diferentes SGBDs em várias condições diferentes.

Após a execução dos testes temos um gráfico com a média acumulada dos WIPS para cada uma das execuções. A análise desses dados será feita a seguir.

6.2 Resultados e Análise

A seguir serão analisados os dados obtidos para cada uma das configurações de teste executadas.

6.2.1 30 *browsers* e banco com dados para 30 *browsers*

Neste teste todos os bancos se comportaram de maneira muito parecida. Mesmo entre os três diferentes perfis do TPC-W os resultados foram muito próximos, entre 4 e 4.5 WIPS.

Isto se deve à pequena quantidade de acessos requerida dos SGDBs, de forma que nesta situação os bancos não estavam sendo o gargalo das transações.

A Figura 10 mostra o gráfico com o desempenho dos SGBDs para esta situação.

6.2.2 150 *browsers* e banco com dados para 30 *browsers*

Aqui se aumenta o acesso aos bancos, porém mantendo-se a mesma quantidade de dados que no teste anterior.

Nesta situação os desempenhos dos bancos de dados ainda estão próximos, porém já se consegue notar algumas diferenças.

O Monet, no perfil de *ordering*, ficou abaixo do desempenho dos outros bancos, com uma média pouco acima de 19 WIPS.

Todos os outros bancos em todos os perfis tiveram desempenho entre 20 e 21 WIPS, muito próximos uns dos outros.

O comportamento dos SGBDs para este perfil de testes está ilustrado na Figura 11.

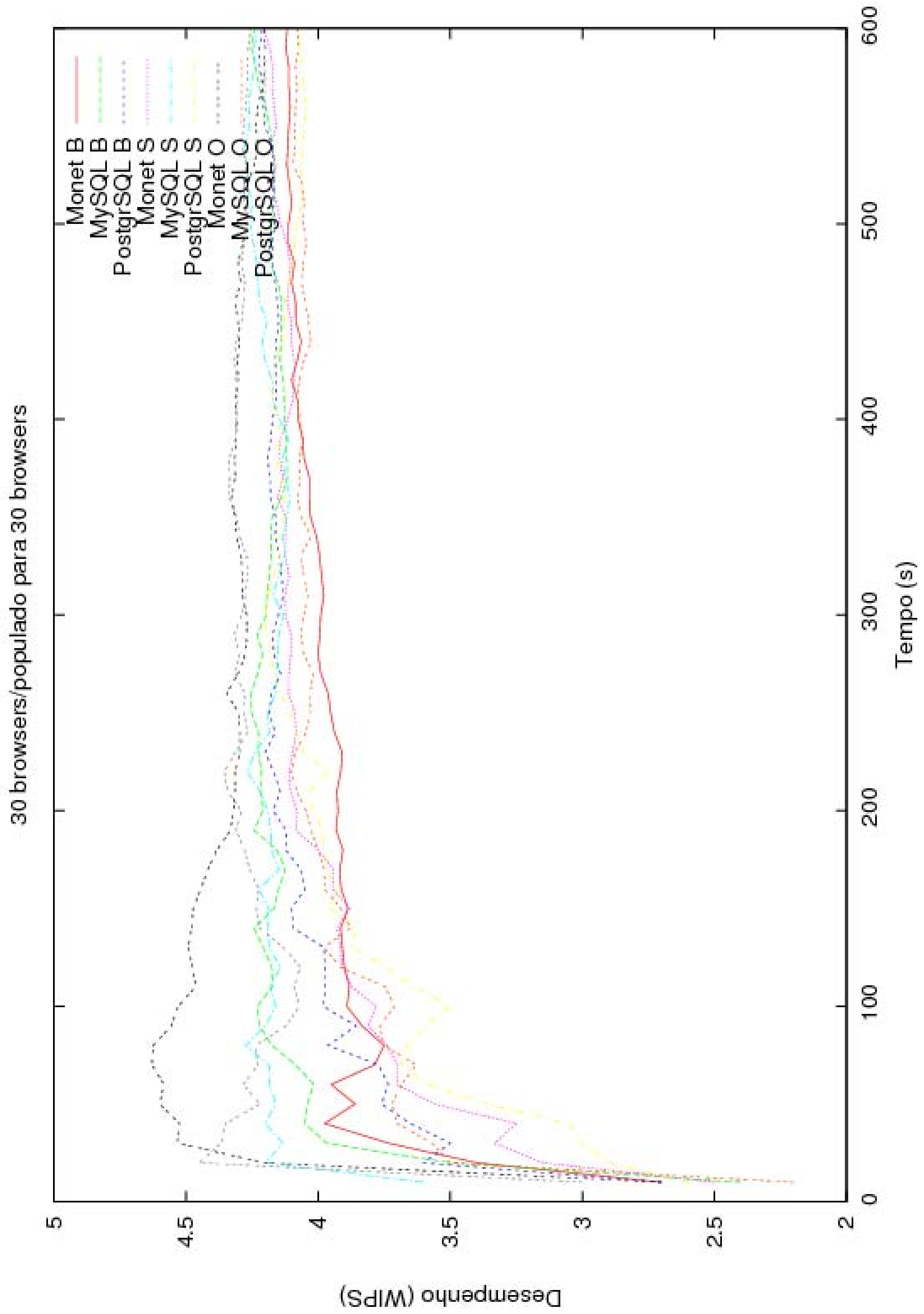

Figura 10 – Desempenho dos bancos de dados para 30 *browsers* simultâneos e banco populado para 30 *browsers*.

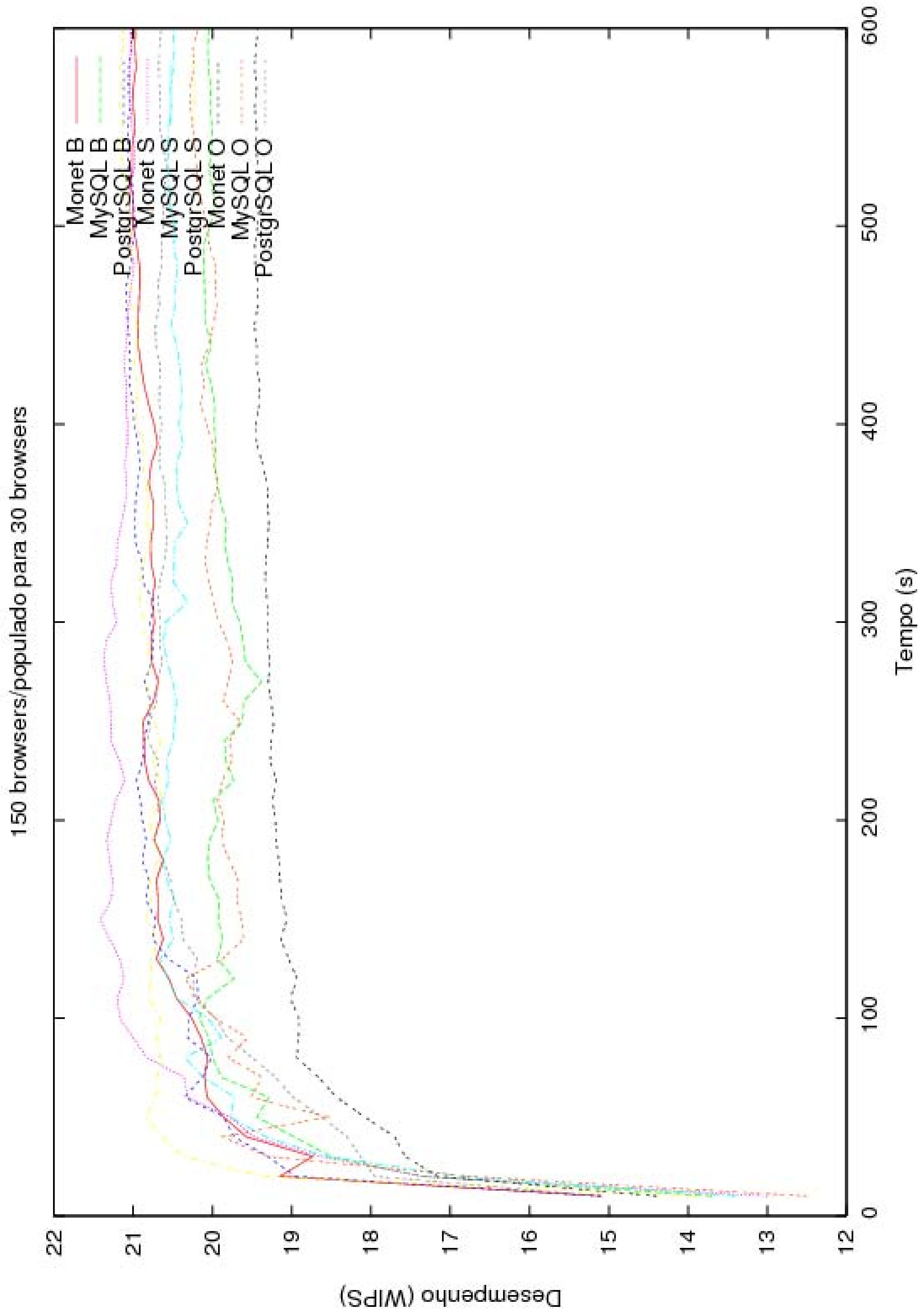

Figura 11 - Desempenho dos bancos de dados para 150 *browsers* simultâneos e banco populado para 30 *browsers*.

6.2.3 300 *browsers* e banco com dados para 30 browsers

Aqui já se consegue notar a diferença de desempenho entre os bancos, já que com esta quantidade de acessos os bancos de dados passam a ser gargalo para as transações.

É possível perceber claramente que o pior desempenho foi do Monet no perfil de *ordering*, com uma média próxima de 20 WIPS. Este é um comportamento que se repete ao longo dos testes, mostrando que o desempenho deste banco para escrita é muito inferior aos outros dois. Para os perfis de *browsing* e *shopping*, obteve 32 e 28 WIPS respectivamente.

É possível perceber que o PostgreSQL teve um desempenho muito superior aos outros tanto no perfil de *browsing*, com muitas leituras, quanto no perfil de *shopping*, com uma mistura no tipo de transações. Ficou com uma média um pouco acima de 40 WIPS para *shopping* e *browsing* e 34 WIPS para *ordering*.

Um fato interessante a ser notado é o MySQL ter desempenho muito inferior tanto em *browsing* quanto *shopping*, com performance de 23 WIPS e 26 WIPS respectivamente. O desempenho em *ordering* foi superior, próximo de 29 WIPS.

Outro fato interessante em relação ao MySQL é que sua média tem uma grande ondulação, não permanecendo estável ao longo da execução.

Estas distorções são explicadas devido aos testes terem sido executados com a limitação de uma transação por vez no banco de dados e em alguns momentos da leitura de dados os MySQL usava 100% da CPU e parava o tráfego até que a transação corrente fosse finalizada. No perfil de *ordering* era muito mais raro ter essa parada para executar uma transação demorada. Isto fez com que, além das médias oscilarem bastante, o perfil de escrita tivesse um desempenho superior ao perfil de leitura.

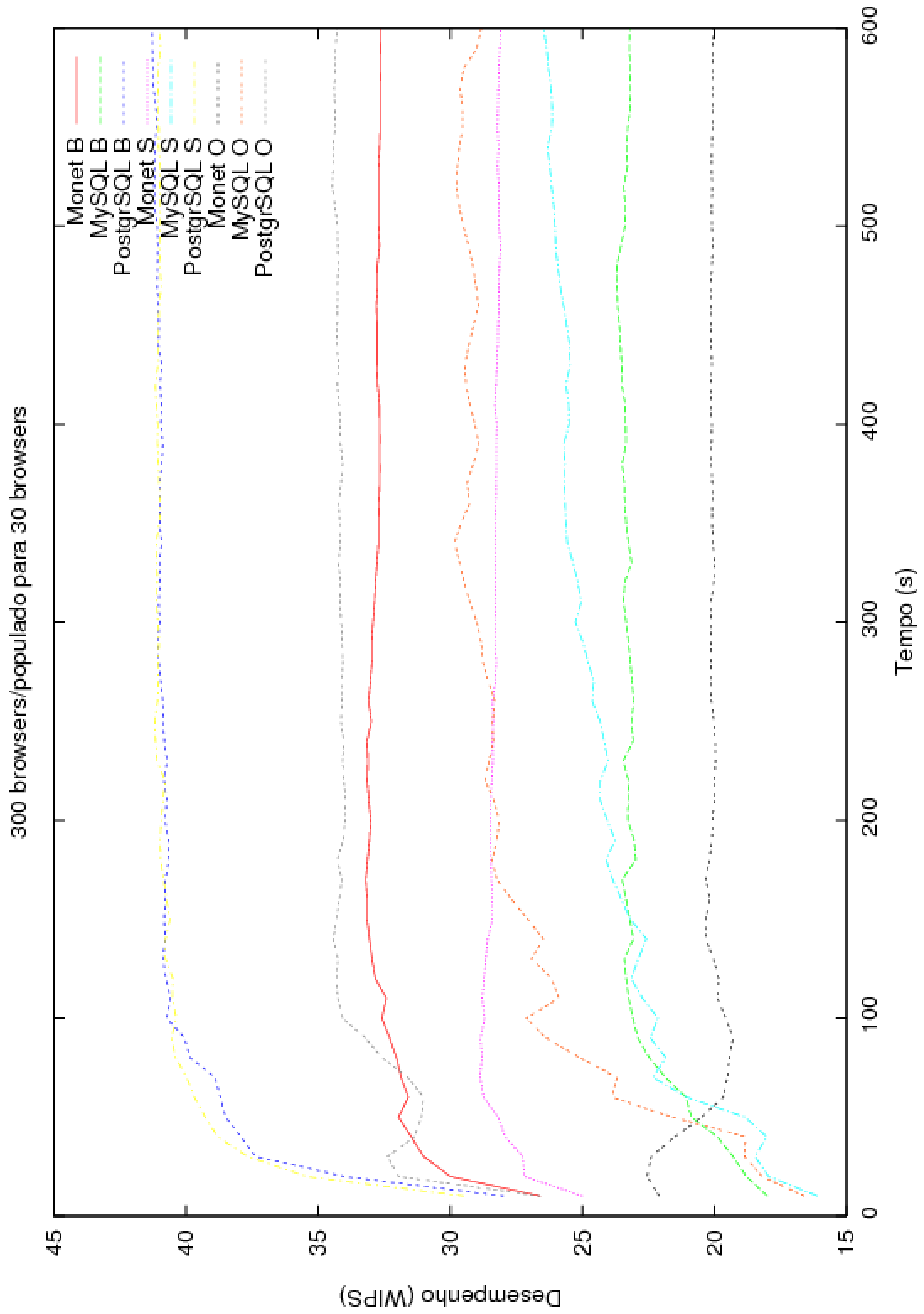

Figura 12 - Desempenho dos bancos de dados para 300 *browsers* simultâneos e banco populado para 30 *browsers*.

6.2.4 600 *browsers* e banco com dados para 30 *browsers*

Neste caso verifica-se que o número de transações por segundo foi pouca coisa maior que no caso com 300 *browsers*, pois, como os acessos são feitos de forma serializada, os bancos de dados já estão nos seus limites de atendimento das transações.

Nota-se claramente que o pior desempenho é do Monet no perfil de *ordering*, com média próxima de 19 WIPS.

O PostgreSQL lidera nos três perfis, com 44 WIPS para *browsing*, 43 WIPS para *shopping* e 33 WIPS para *ordering*.

O Monet consegue bons resultados para *browsing* e *shopping*, com 31 e 27 WIPS respectivamente.

O MySQL mantém seu comportamento de melhor desempenho para escrita que para leitura, pelos mesmos motivos citados anteriormente. Obteve 23 WIPS para *browsing*, 25 para *shopping* e 31 WIPS para *ordering*.

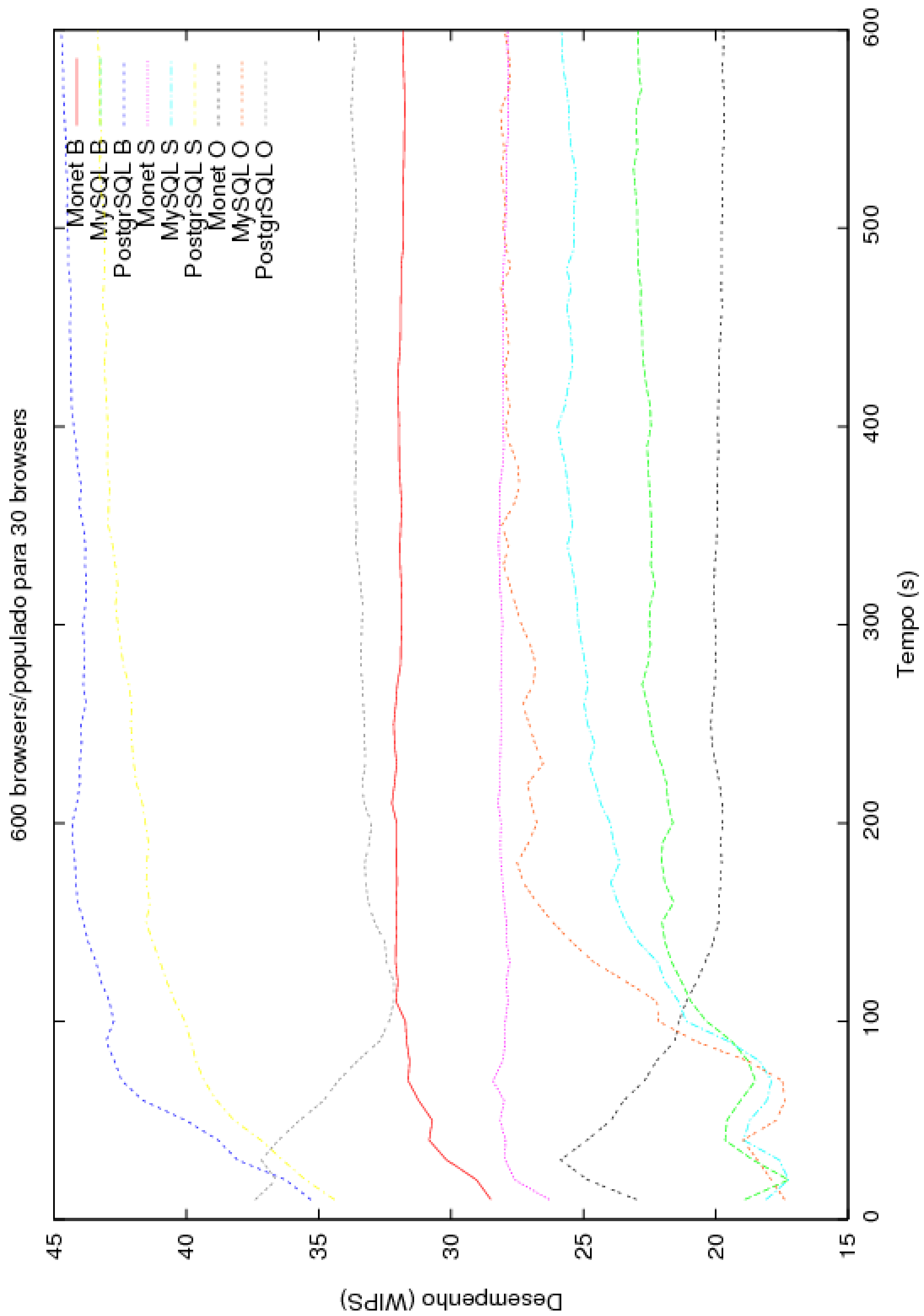

Figura 13 - Desempenho dos bancos de dados para 600 *browsers* simultâneos e banco populado para 30 *browsers*.

6.2.5 150 *browsers* e banco com dados para 150 *browsers*

Neste teste foi aumentada a quantidade de dados existente nos SGBDs. A quantidade de dados apresentada está de acordo com a especificação do TPC-W.

Com o incremento do tamanho do banco de dados, já se percebe as diferenças de desempenho entre os bancos de dados claramente.

O PostgreSQL ficou na frente de todos em todos os perfis, comportamento observado ao longo de todos os testes. Ficou em torno de 20 WIPS para todos os perfis. Pode-se perceber, comparando a performance deste teste com o teste de 150 *browsers* e banco com dados para 30, que seu desempenho não piorou, para a mesma quantidade de browsers, com o acréscimo de dados no banco.

O MySQL ficou com o pior desempenho para *browsing* e *shopping*, em torno de 19 WIPS, um pouco inferior ao teste com menos dados no banco. Observa-se que as irregularidades da média acentuaram-se com mais dados no SGDB, pois o MySQL gastou mais tempo nas transações que demoram para retornar um resultado.

O Monet teve seu desempenho de *ordering* bem baixo, como nos outros testes, pouco acima de 8 WIPS. Nos perfis de *browsing* e *shopping* obteve 20 e 18 WIPS respectivamente. Percebe-se que o desempenho cai consideravelmente para escrita à medida que se têm mais dados populados no banco. Para leitura o desempenho também caiu, porém muito menos do que para escrita.

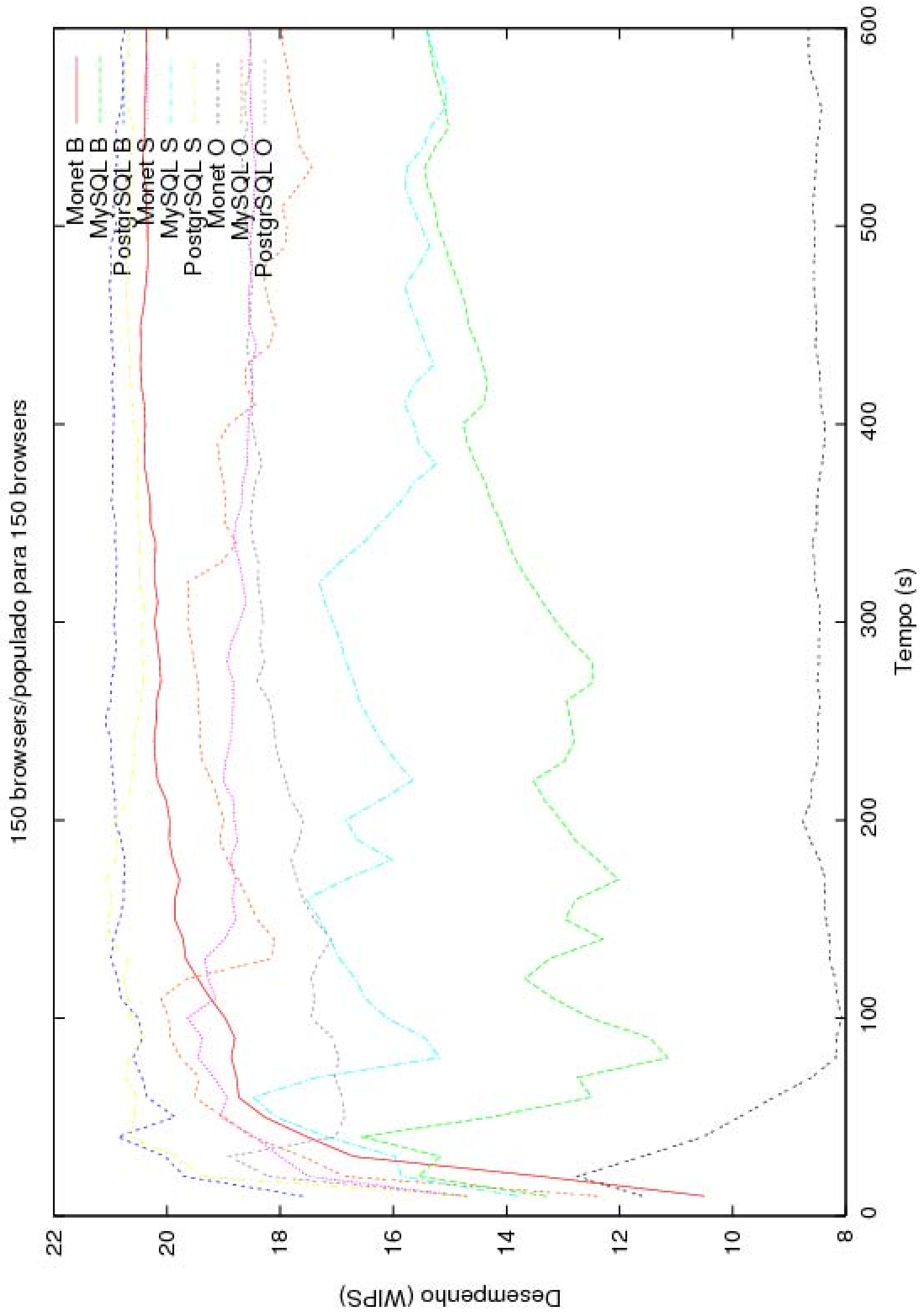

Figura 14 - Desempenho dos bancos de dados para 150 *browsers* simultâneos e banco populado para 150 *browsers*.

6.2.6 300 *browsers* e banco com dados para 300 *browsers*

Aqui aumenta-se a quantidade de *browsers* simultâneos e também a quantidade de dados nos SGBDs.

Como nos outros testes o PostgreSQL teve o melhor desempenho em todos os perfis, obtendo 27, 25 e 22 WIPS para *browsing*, *shopping* e *ordering* respectivamente. Como a quantidade de dados aumentou consideravelmente em relação ao outro teste com 300 *browsers*, pode-se perceber uma queda no desempenho, mas que foi inferior à queda dos outros bancos.

O MySQL tem seu comportamento variável mais acentuado ainda com a grande quantidade de dados. Obteve 12 WIPS para *browsing*, 14 para *shopping* e pouco mais de 20 para *ordering*. Houve uma queda de desempenho com o aumento da quantidade de dados e acentuou-se a diferença no desempenho entre o perfil de leitura e de escrita, com o perfil de escrita muito melhor que o de leitura.

O Monet continua com o segundo melhor desempenho para os perfis de *browsing* e *shopping*, com 22 e 15 WIPS respectivamente. Houve um piora no desempenho com o incremento do tamanho do banco, da mesma forma que aconteceu com os outros SGBDs. Para o perfil de *ordering*, obteve um desempenho muito inferior aos outros, pouco acima de 5 WIPS.

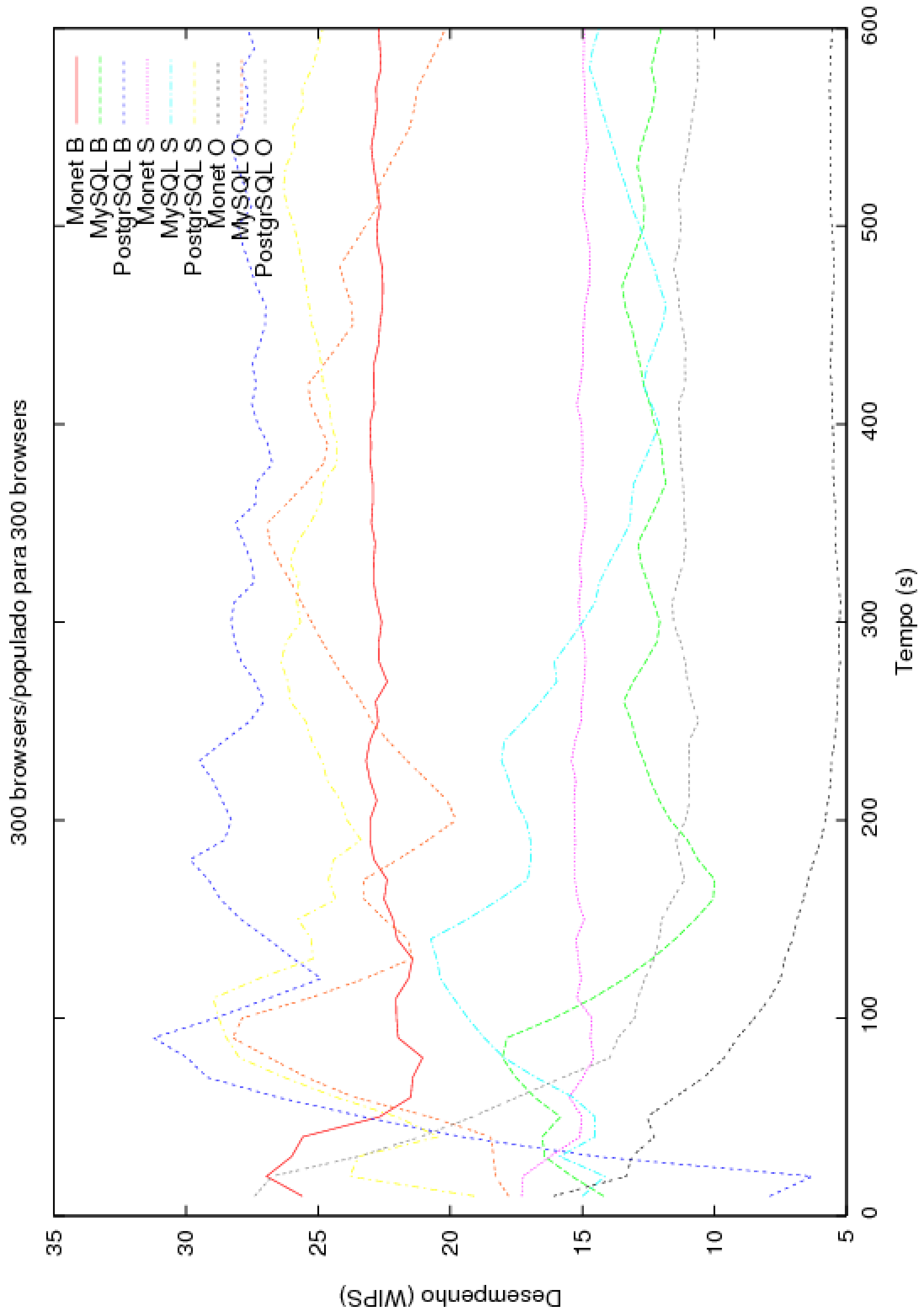

Figura 15 - Desempenho dos bancos de dados para 300 *browsers* simultâneos e banco populado para 300 *browsers*.

6.2.7 600 *browsers* e banco com dados para 300 *browsers*

Aqui se manteve a quantidade de dados do teste anterior, pois o Monet teve um desempenho muito ruim enquanto estava sendo populado. Observando seu desempenho no perfil de escrita explica-se o motivo pelo qual o tempo para conseguir inserir dados neste banco é muito grande.

O PostgreSQL continua na frente, com desempenho de 26 WIPS para *browsing,* 23 WIPS para *shopping* e 22 WIPS para *ordering*. Com o mesmo volume de dados e com aumento da carga em relação ao teste anterior houve um pequeno declínio no desempenho. Também nota-se que o desempenho caiu com o aumento do volume de dados, comparando-se com o teste feito para a mesma quantidade de *browsers* e volume menor de dados.

O MySQL obteve 15 WIPS para *browsing*, 13 para *shopping* e 15 para *ordering*. Com o aumento no volume de acessos houve uma queda no desempenho em geral, mas nota-se que a queda foi maior para *ordering*. A queda também foi grande em relação à mesma carga, mas menos dados.

O Monet apresentou uma queda de desempenho em relação ao teste anterior para o perfil de *browsing,* passando para 11 WIPS. Em *shopping*, obteve 15 WIPS e em *ordering* teve desempenho próximo ao teste anterior, com 5 WIPS. Também nota-se, como nos outros bancos, queda de desempenho devido ao aumento da carga e devido ao aumento do volume de dados.

Um comportamento interessante observado neste teste é que em todos os SGDBs e em todos os perfis houve uma queda da média ao longo do tempo de execução. Isso pode ser causado pela fragmentação dos dados ou pelos algoritmos empregados na reconstrução dos índices. Porém, não se pode afirmar com exatidão, pois uma análise detalhada deste comportamento não foi realizada, pois isso significaria entrar na implementação dos SGBDs.

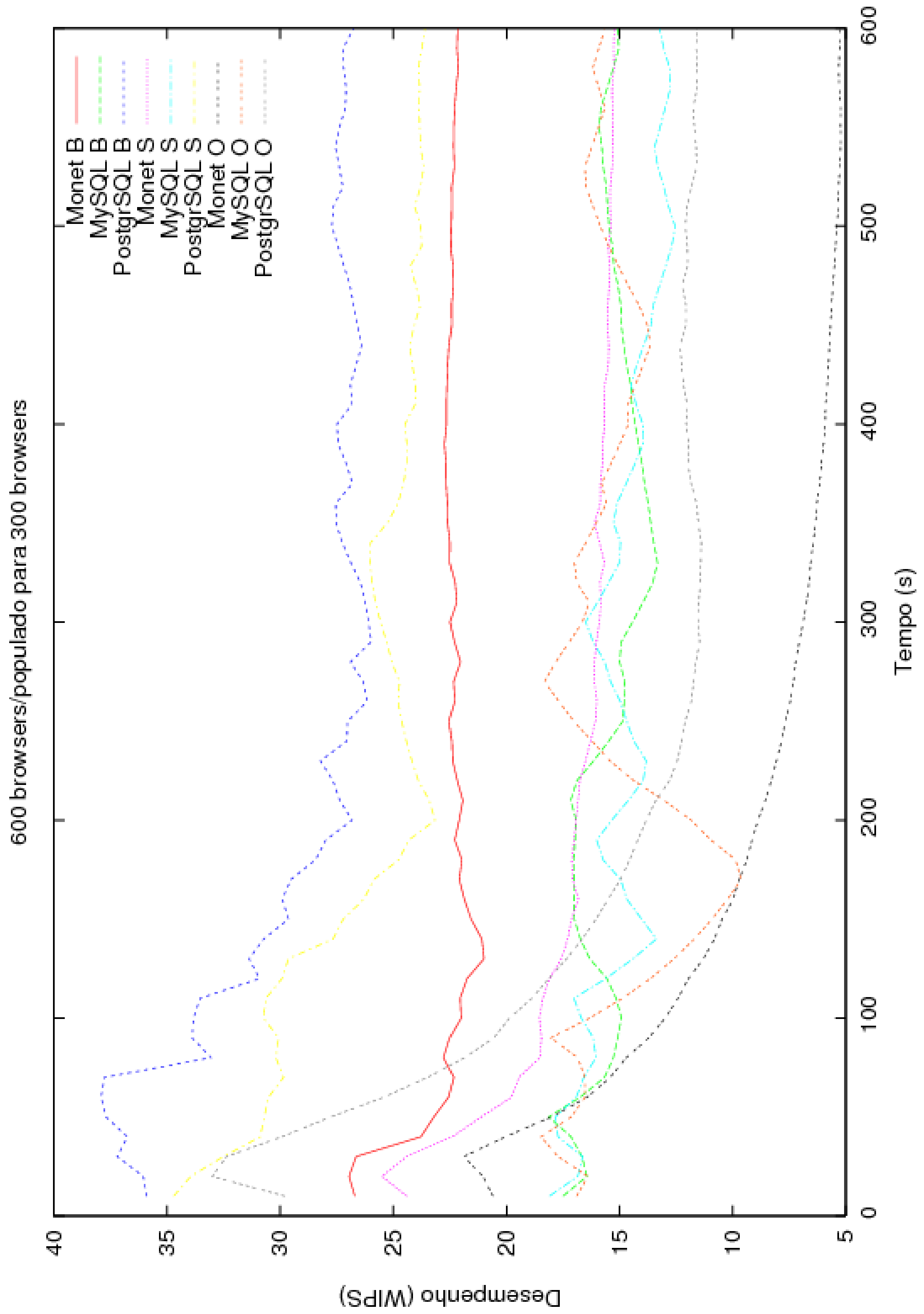

Figura 16 - Desempenho dos bancos de dados para 600 *browsers* simultâneos e banco populado para 300 *browsers*.

6.3 Sumário

Este capítulo mostrou a descrição do experimento realizado, a configuração de hardware e software e as variações de carga e tamanho de banco de dados empregadas em cada um dos experimentos.

Também mostrou a análise dos resultados obtidos, onde para pequena quantidade de dados e pouca carga os SGBDs se comportam de maneira parecida. Porém, conforme aumenta-se a carga e a quantidade de dados o comportamento varia entre eles. O PostgreSQL, em geral, teve o melhor desempenho em cada um dos três perfis do TPC-W, *shopping*, *browsing* e *ordering*. O Monet conseguiu bom desempenho, comparado com o MySQL, nos perfis de *shopping* e *browsing*, porém no perfil de *ordering* seu desempenho piorava à medida que a quantidade de dados e a carga aumentavam. O MySQL, devido à limitação de execução de uma operação por vez, não obteve bons resultados para *shopping* e *browsing*, pois algumas consultas eram demoradas. Em *ordering*, se comportou melhor que o Monet.

No próximo capítulo tem-se a conclusão final deste trabalho.

Capítulo 7

Conclusão

Este trabalho apresenta um estudo sobre o desempenho de um SGBD de memória principal (Monet), comparando com outros dois SGBDs tradicionais (MySQL e PostgreSQL), para avaliar seu uso num ambiente replicado em busca de alto desempenho. Embora existam várias pesquisas de desempenho em bancos de dados tradicionais, não foi encontrado nenhum outro trabalho que analise o desempenho de um banco de dados de memória principal. Como este trabalho é o primeiro na área, não temos como comparar o resultado obtido aqui com outros trabalhos.

Os resultados observados foram:

- Embora o Monet tenha sido construído pensando-se em tirar o melhor desempenho das máquinas modernas, utilizando técnicas de SGBDs em memória principal, para a análise utilizando o perfil de aplicações *web*, ele não se mostrou vantajoso, uma vez que o PostgreSQL ficou normalmente com um desempenho superior ao Monet. Além disso, o desempenho de escrita do Monet é muito ruim comparados aos bancos de dados tradicionais.
- O Monet possui ainda algumas deficiências para uso num sistema real, como a impossibilidade de uso de operações em paralelo. Para o teste, isto foi contornado utilizando-se uma operação por vez, mas, num ambiente real, o desempenho de se fazer uma operação por vez será muito aquém do desejado.

Como o Monet foi construído pensando-se em OLAP e *data mining*, tentamos executar o teste de junção de tabelas relacionais, *winestore*[8]. Porém, não foi possível executá-lo no Monet, pois o processo do Monet era interrompido cada vez que o teste era executado. Com isto não foi possível compará-lo com os bancos tradicionais no perfil de OLAP e *data mining*.

Devido a isto conclui-se que o Monet ainda não tem o desempenho e não está maduro o suficiente para ser utilizado num ambiente de replicação de alto desempenho. Para dar continuidade à análise de um possível banco de dados de memória principal para uso em replicação, outros sistemas poderiam ser avaliados e comparados com sistemas tradicionais ou até com alguns sistemas replicados existentes no mercado. Outra análise que mereceria atenção seria a comparação do Monet com bancos de dados tradicionais onde todos os SGBDs teriam fragmentação total em seu modelo de dados, o que o Monet faz internamente, de forma que não teríamos essa diferença de fragmentação na comparação de desempenho.

Referências Bibliográficas

[1] Astrahan, M., Blasgen, M., Chamberlin, D., Eswaran, K., Gray, J., Griffiths, P., King, W., Lorie, R., McJones, P., Mehl, J., Putzolu, G., Traiger, I., Wade, B., Watson, V. System R: Relational Approach to Database Management. *ACM Trans. On Database Systems*. 1976.

[2] Boncz, P. A. Monet: A Next-Generation DBMS Kernel for Query Intensive Applications. *PhD Thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands*, páginas 7-38, Mai. 2002.

[3] Cattel, R., Atwood, T., Barry, D., Dubl, J., Ferran, G., Loomis, M., Wade, D. *The Object Database Standard: ODMG.*, 1994.

[4] Cattel, R., Barry, D., Bartels, D.,Berler, M., Eastman, J., Gamerman, S., Jordan, D., Springer, A., Strickland, H., Wade D. *The Object Database Standard: ODMG 2.0.*, 1997.

[5] Codd, A. A Relational Model Of Data for Large Shared Data Banks. *Communications of the ACM*, Junho de 1970.

[6] CWI database research group. An architectural Overview of MonetDB. Disponível em http://monetdb.cwi.nl/TechDocs/Core/monet/index.html. Última data de consulta: 26 de Junho de 2006.

[7] Fisher, M. J., Lynch, N. A., Paterson, M. S. Impossibility of distributed consensus with one fault process. *Journal of the ACM*, Abril de 1985.

[8] Guimarães, C. C. Fundamentos de Bancos de Dados: Modelagem, Projeto e Linguagem SQL. Ed. Unicamp, 2003.

[9] Kemme, B., Alonso, G. Don't be lazy, be consistent: Postgres-R, A new way to implement Database Replication. In *Proceeding of the 26th VLDB Conference, Cairo, Egito*, 2000.

[10] Raab, F., Kohler, W., Shah, A. Overview of the TPC Benchmark C: The Order-Entry Benchmark. Disponível em: http://www.tpc.org/tpcc/detail.asp. Última data de consulta: 3 de Julho de 2006.

[11] Smith, W. D. TPC-W: Benchmarking An Ecommerce Solution. *Intel Corporation*, Revision 1.2.

[12] Stonebraker, M., Wong, E., Kreps, P., Held, G. The Design and Implementation of INGRES. *ACM Trans. On Database Systems*, 1976.

[13] Transaction Processing Performance Concil. TPC Benchmark H (Decision Support) Standard Specification, revision 2.3.0, Ago. 2005.

[14] Transaction Processing Performance Concil. TPC Benchmark W (Web Commerce) Specification, version 1.8, Fev. 2002.

[15] UW-Madison Computer Architecture Group. Java TPC-W Implementation Distribution. *University of Wisconsin-Madison*. Disponível em: http://www.ece.wisc.edu/~pharm/tpcw.shtml. Última data de consulta: 26 de Junho de 2006.

[16] Wiesmann, M., Pedone F., Schiper, A., Kemme B., Alonso G. Understanding Replication in Databases and Distributed Systems. In *Proceedings of 20th International Conference on Distributed Computing Systems*, 2000.

[17] Williams, H. Web Database Applications. Ed. O'Reilly, 2002.