JORGE AUGUSTO MEIRA

UMA METODOLOGIA INCREMENTAL DE TESTE DE ESTRESSE DE BANCO DE DADOS TRANSACIONAL DE GRANDE ESCALA

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Eduardo Cunha de Almeida

CURITIBA

2010

JORGE AUGUSTO MEIRA

UMA METODOLOGIA INCREMENTAL DE TESTE DE ESTRESSE DE BANCO DE DADOS TRANSACIONAL DE GRANDE ESCALA

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Eduardo Cunha de Almeida

CURITIBA

2010

.

Dedico esta dissertação a minha mãe Sueli, meu pai Orlando e meu irmão Joel que sempre me apoiaram e me incentivaram em todas as decisões e escolhas que tomei até chegar aqui.

Dedico também ao professor, amigo e orientador Eduardo Cunha de Almeida por toda ajuda, orientação e dedicação durante a realização desse trabalho.

AGRADECIMENTOS

.

Gostaria de agradecer ao amigo João Eugênio pela ajuda em vários momentos importantes e dúvidas que surgiram durante a realização desse trabalho.

Ao amigo Bruno Ribas pelo apoio na configuração do ambiente de testes.

Ao professor Marcos Sunye por co-orientar esse trabalho.

A Universidade Federal do Paraná e ao Departamento de Informática, professores e funcionários.

E a todos que de alguma forma estiverem presente e me apoiaram durante a realização desse trabalho.

SUMÁRIO

LISTA DE FIGURAS v

LISTA DE TABELAS vi

LISTA DE SIGLAS vii

RESUMO viii

ABSTRACT ix

1 INTRODUÇÃO 1

2 BANCO DE DADOS TRANSACIONAIS DE GRANDE ESCALA 3

2.1 Testes em banco de dados de grande escala 6

3 TESTE DE SOFTWARE 7

3.1 Teste de desempenho e Benchmark 9

3.1.1 DebitCredit 10

3.1.2 TPC 11

3.1.2.1 TPC-C 14

3.1.2.2 TPC-H 16

3.1.2.3 TPC-E 18

3.1.3 Set Query 20

3.1.4 AS^{3}AP 22

3.2 Teste de Estresse 22

3.2.1 TPC-B 23

3.3 Comparativo das Abordagens 25

4 METODOLOGIA INCREMENTAL DE TESTE DE ESTRESSE (MITE) 27

4.1 Especificação do Banco de Dados 27

4.2 Sequência de execução 29
4.3 MITE vs TPC-like 31
4.4 Implementação 31
4.5 Uma visão geral do PeerUnit 32
4.6 Implementação da MITE 35

5 EXPERIMENTOS **37**
5.1 Configuração do SGBD 37
5.2 Configuração de hardware 38
5.3 Comparação das Arquiteturas de Teste 38
5.4 Teste de Grande Escala 40

6 CONCLUSÃO **45**
6.1 Trabalho Futuro 46

ANEXOS **47**

REFERÊNCIAS **52**

LISTA DE FIGURAS

2.1 Passos de uma transação 4
2.2 Tipos de transação 5

3.1 Espiral (teste de software) [18] 8
3.2 Arquitetura DebitCredit [8] 10
3.3 Transação DebitCredit [8] 10
3.4 Etapas do TPC-DS [25] 14
3.5 Exemplo Set Query [16] 20
3.6 Exemplo Tabela BENCH [16] 21
3.7 Perfil de transação - TPC-B [23] 23
3.8 Distribuição de tempo de residência [23] 24

4.1 Esquema TPC-B [23] 28
4.2 Arquitetura do PeerUnit [2] 33
4.3 ClasseTestMITE 36

5.1 Teste comparativo de carga incremental 40
5.2 Teste 1000 transações 41
5.3 Teste 10000 transações 42
5.4 Teste 100000 transações 43
5.5 Degradação do tempo para concluir uma transação 44

LISTA DE TABELAS

3.1 Benchmarks TPC 12
3.2 Sequência de consultas por escala 17
3.3 Comparativo entre abordagens 25

4.1 Caso de teste da MITE 35

5.1 Máquinas-cliente 39
5.2 Teste de cargas incrementais 39
5.3 Taxa de erro 43

LISTA DE SIGLAS

.

ACID - Atomicidade, Consistência, Isolamento e Durabilidade

BD - Banco de Dados

DW - Data Warehouse

FS - Fator de Escala

OLAP - Online Analytical Processing

OLTP - Online Transaction Processing

P2P - Peer-to-peer

SGBD - Sistema Gerenciador de Banco de Dados

SQL - Structured Query Languade

SUT - System Under Test

RT - Tempo de Residência

TPC - Transaction Processing Performance Council

RESUMO

O uso de sistemas de grande escala é cada vez mais comum nos mais diferentes tipos de aplicação e requerem Sistemas Gerenciadores de Banco de Dados (SGBD) robustos e de alta escalabilidade. Neste contexto, é importante avaliar o desempenho do SGBD para saber qual é o mais adequado para cada tipo de aplicação. Este trabalho apresenta uma nova metodologia de teste de estresse. A Metodologia Incremental de Teste de Estresse (MITE) para banco de dados de grande escala prevê testes sequenciais com incremento de carga de trabalho até o limite do sistema, momento no qual inicia a degradação de desempenho e aparecem erros relacionados (i.e., bugs). O objetivo é avaliar um sistema segundo seus limites e a degradação de seu desempenho em ambientes de grande escala. Experimentos demonstraram que a MITE foi efetiva em levar o sistema sob teste até seu limite. Como consequência o sistema entra claramente em estado de degradação além de apresentar erros relacionados à carga de trabalho.

ABSTRACT

Large-scale database systems are widely used in many types of applications, such as: e-commerce, data warehousing or stock exchange. Transactional database systems are especially important for dealing safely with a huge number of critical operations, e.g. registering sales and business operations for e-commerce applications. Such large-scale systems must be robust to limit scalability conditions. This work presents a testing methodology for large-scale transactional database systems. The methodology is focused on stress testing in order to search for bugs linked to the scale of the system. We want to ensure the availability of the database under limit conditions, and measure its ability to avoid a denial of service. We validate our methodology with empirical studies on a popular open source database system and detail the real bugs which have been found.

CAPÍTULO 1

INTRODUÇÃO

Sistemas em grande escala são amplamente utilizados em todos os tipos de aplicações (e.g. bancos, vendas, VoIP). Um sistema de grande escala é composto por um grande número de componentes que trabalham juntos. Por exemplo, um cluster de computadores reúne um conjunto de máquinas (i.e. nodos) para formar um único computador de alto desempenho. Desta forma, o sistema é facilmente escalável apenas adicionando um novo nodo. Desempenho e custo são portanto a motivação para o uso desses sistemas.

Outra vantagem é a alta disponibilidade, uma vez que os componentes com falha podem ser facilmente substituídos. Por exemplo, nos resultados dos benchmarks do *Transaction Processing Performance Council* (TPC) encontramos um sistema de cluster com quase a mesma capacidade da CPU de um sistema centralizado, mas nove vezes mais barato, mostrando o custo como ponto determinante para justificar o uso de um sistema de processamento distribuído.

Aplicações de banco de dados também avançam para sistemas de grande escala. Novos sistemas de gerenciamento de banco de dados (SGBD) são executados dentro de clusters, beneficiando-se do desempenho e da escalabilidade.

O desempenho dos sistemas é geralmente avaliado através de um *benchmark*. Existem vários benchmarks disponíveis para todo tipo de sistema e equipamento. No contexto de bancos de dados, os benchmarks tiveram um papel importante no desenvolvimento dos principais SGBDs do mercado. Entre eles estão: Debit/Credit [8], AS3AP [29], TPCs [21], *SetQuery* [16], etc. Estes benchmarks, que são chamados de TPC-like, são focados em mostrar índices de desempenho sob diferentes cargas de trabalho. Então, alguém interessado em adquirir um dado SGBD para ser instalado em um dado sistema operacional, pode se basear em algum destes *benchmarks* para saber qual o seu índice de desempenho e então planejar sua aquisição. De fato, benchmarks norteiam o mercado de

SGBD e são ferramentas poderosas na divulgação de novas implementações das principais empresas do ramo [8].

Neste trabalho, apresentamos uma metodologia incremental de teste de estresse (MITE) que vai além dos objetivos dos benchmarks de desempenho comumente utilizados em testes de SGBD. Na realidade, a MITE tira proveito dos benchmarks como carga de trabalho para exercitar e estressar o SGBD sob teste. Além disso, nossa metodologia sugere testar com grandes cargas de trabalho para reproduzir ambientes de SGBD de grande escala.

Em um SGBD de grande escala, falhas acontecem constantemente, devido ao grande número de componentes. Estes sistemas são projetados para serem tolerantes a falhas e escaláveis quando chegam ao limite de desempenho. A MITE testa os SGBDs exatamente neste limite buscando erros relacionados às altas cargas de trabalho. Além disso, a MITE busca determinar a degradação do SGBD quando submetido a estas cargas de trabalho.

Esta degradação pode então gerar um índice que corresponde à perda acentuada de desempenho de algum serviço. Por exemplo, o número de transações perdidas por algum gargalo de desempenho. Acreditamos que este índice é uma ferramenta importante para planejar atualizações e melhorias nos SGBDs.

Este trabalho está organizado em seis capítulos. O Capítulo 2 introduz conceitos de bancos de dados transacionais de grande escala. O Capítulo 3 apresenta uma revisão da literatura no que diz respeito a teste de software e dos principais benchmarks já desenvolvidos. O Capítulo 4 descreve a MITE, proposta de metodologia para teste de estresse. O Capítulo 5 apresenta detalhes de implementação da metodologia proposta e sua aplicação. Finalmente o Capítulo 6 apresenta a conclusão e propostas de trabalhos futuros dessa pesquisa.

CAPÍTULO 2

BANCO DE DADOS TRANSACIONAIS DE GRANDE ESCALA

A grande quantidade de informações geradas atualmente por inúmeros tipos de aplicações requerem sistemas gerenciadores de banco de dados (SGBD) robustos e de alta escalabilidade a fim de suprir a necessidade dessas aplicações. Os SGBDs são conjuntos de programas que administram a criação, o uso e a manutenção de uma base de dados.

É cada vez mais comum que o SGBD fique exposto a uma carga elevada de trabalho. Esse tipo de ambiente requer um estudo de desempenho cuidadoso, pois em muitos casos é necessário admitir um percentual de falha que deve ser previsto a partir de testes.

Aplicações comerciais e de negócios, estruturam suas bases de dados para atender basicamente dois tipos de sistemas: Transacional ou de Suporte à decisão. Nesse trabalho iremos nos concentrar nos sistemas transacionais, que são o foco da MITE.

Sistemas transacionais, conhecidos como aplicações OLTP (Online Transaction Processing), são responsáveis por administrar sistemas que tratam das transações de uma determinada organização (e.g., transações bancárias, sistema de reservas de hotel *online*).

É comum a esses sistemas, várias ações de leitura e transformação de valores de seus registros e dispositivos. O conjunto de ações que fazem parte de uma transformação consistente de um estado do sistema podem ser agrupadas em uma transação. Toda e qualquer transação deve preservar restrições de consistência do sistema, que obedecem leis de transformação de estados consistentes em novos estados consistentes.

Diz-se então que, operações devem ser atômicas e duráveis, ou seja, se todas as operações de uma transação são realizadas com sucesso o novo estado é gravado, caso contrário, a transação deve ser abortada. Dessa forma uma transação ou é efetivada em sua completude ou é abortada. Uma vez que uma transação é efetivada, seus efeitos só podem ser alterados por outra transação[11].

Para fins de recuperação, o sistema precisa manter o controle de quando a transação começa, termina, é consolidada ou abortada, tal que uma transação segue os seguintes passos [9](Figura 2.1):

- *Estado consistente 1*: Marca o início de execução da transação.
- *Transições*: Operações de leitura e escrita sobre o BD executadas como parte da transação;
- *Commit*: Sinaliza fim bem sucedido, afirmando que as alterações executadas podem ser consolidadas;
- *Rollback*: Sinaliza fim mal sucedido. Dessa forma toda e qualquer operação realizada pela transação deve ser desfeita;
- *Estado consistente 2*: Operações de escrita e leitura terminaram indicando o fim da transação. Se o commit foi realizado as alterações são consolidadas.

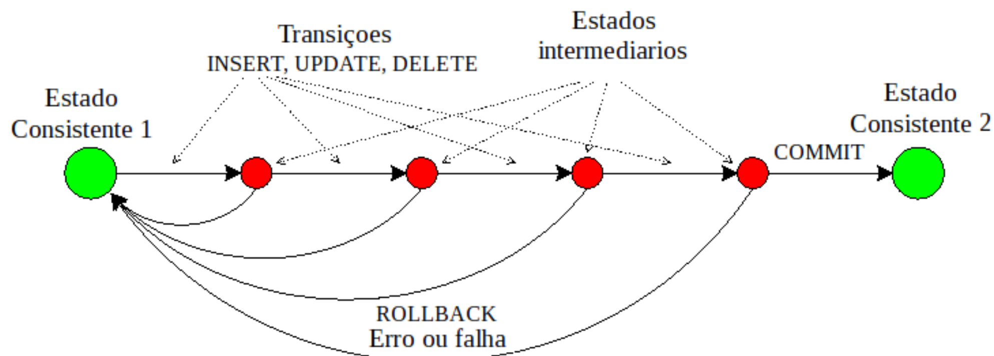

Figura 2.1: Passos de uma transação

As propriedades de uma transação, comumente chamadas de ACID, são [11]:

- Atomicidade: garante que ou toda a transação é realizada com sucesso ou nada é alterado;
- Consistência: regras de integridade dos dados devem ser asseguradas;
- Isolamento: o isolamento dos dados deve ser garantido. Transações podem ser concorrentes desde que não alterem os mesmos dados;

- Durabilidade: uma vez que uma transação é efetivada, as alterações são permanentes;

Podemos dividir as transações em dois tipos: simples e complexa. Uma transação simples é uma sequência de ações lineares sem interdependência de ações. Já uma transação complexa pode ter concorrência dentro dela mesmo, ou seja, possui uma ou mais ações que dependem do resultado de um grupo de outras ações. Exemplos desses dois tipos de transação podem ser observados na Figura 2.2, onde T1 é um exemplo de transação simples e T2 complexa.

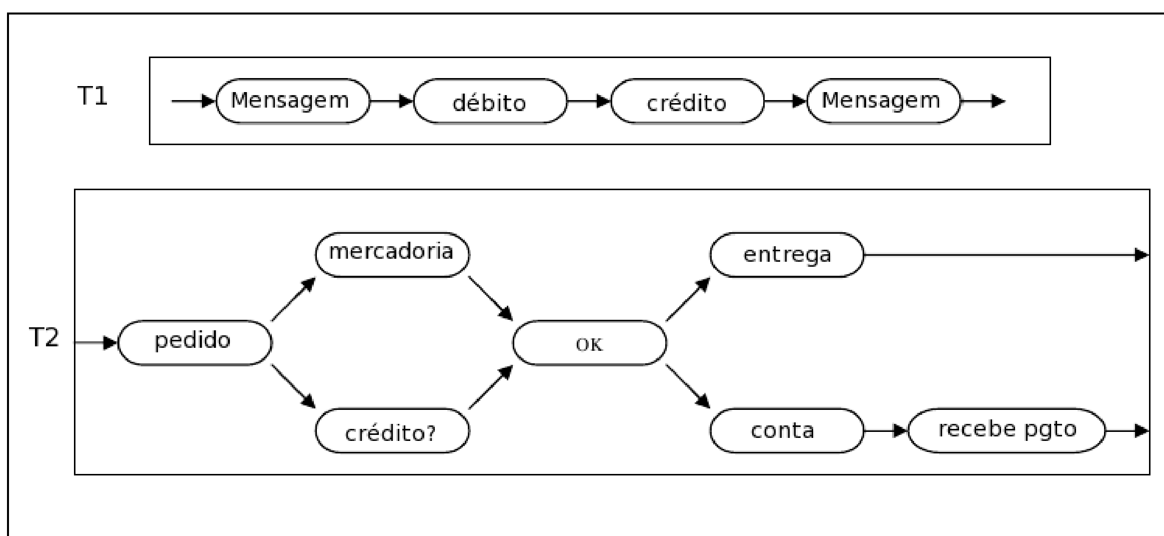

Figura 2.2: Tipos de transação

Stonebraker et al [20], ressaltam as limitações dos modelos e arquiteturas atualmente usados para sistemas OLTP de grande escala. Segundo os autores, o atual paradigma de Sistema Gerenciador de Banco de Dados para sistemas de comércio OLTP, podem ter seu desempenho ultrapassado em até duas ordens de magnitude, se comparados com novas arquiteturas de armazenamento, como por exemplo o *H-Store*[1] ou modelos híbridos. O autor se baseia em experimentos realizados com o benchmark TPC-C, exposto em detalhes na subseção 3.1.2.1, para sugerir a aposentadoria de 25 anos de estudos em bancos de dados relacionais por novas arquiteturas direcionadas a resolver problemas específicos, como por exemplo, sistemas OLTP.

[1]Banco de dados orientado a coluna

2.1 Testes em banco de dados de grande escala

Visto a importância de uma boa arquitetura de banco de dados, que atenda as necessidades de aplicações de grande escala, é necessário que testes sejam realizados para avaliar o comportamento dessas arquiteturas em ambientes semelhantes aos quais serão inseridos. No próximo capítulo veremos de que forma estes testes vem sendo realizados e quais são suas limitações.

CAPÍTULO 3

TESTE DE SOFTWARE

Avaliar a qualidade e o desempenho de um sistema computacional é uma preocupação comum entre usuários, técnicos e projetistas de sistemas. Os testes de sistemas são realizados a fim de garantir sua qualidade e encontrar o melhor desempenho em comparação a outros sistemas que executam a mesma tarefa [15].

A atividade de teste de software é um elemento crítico na garantia da qualidade de software sendo a última revisão de todos os requisitos de um projeto. Testes realizados sobre um determinado sistema estão compreendidos em técnicas de teste de software.

Ao contrário dos outros passos aplicados para a construção de um software, nos quais o objetivo é suprir os requisitos para construir o software, na fase de testes o objetivo é "demolir" o software. Podemos assim dizer que o teste de software pode ser entendido como uma busca por erros. Quando se testa um software, segundo Pressman [18], busca-se descobrir sistematicamente diferentes classes de erros com uma quantidade de tempo e esforço mínimos. Se a atividade de teste for conduzida com sucesso, ela descobrirá erros no software.

O teste de software jamais mostrará a ausência de erros, mas sim sua presença. As técnicas compreendidas para testar um software seguem basicamente duas abordagens distintas. A primeira é chamada de teste estático e a segunda, teste dinâmico.

O teste estático, valida o comportamento interno dos componentes do software. São testes estruturais que trabalham diretamente sobre o código-fonte, no qual busca-se avaliar fluxo de dados, testes de condição, ciclos e caminhos lógicos.

O teste dinâmico, valida o comportamento externo dos componentes do software ou até mesmo do software como um todo. Seu foco é a transformação dos dados de entrada nos dados de saída.

Ambos os testes possuem fases de aplicação dadas como teste de unidade, validação,

integração e sistema, representado em forma de espiral como na Figura 3.1. O teste de unidade valida partes isoladas do sistema, ou seja, pequenos componentes ou funções. Na integração é avaliado o comportamento da integração entre os vários componentes e funções unitárias e o comportamento das interfaces entre as unidades. O teste de validação busca validar o atendimento aos requisitos estabelecidos. Finalmente o teste de sistema avalia a funcionalidade e o desempenho do sistema como um todo.

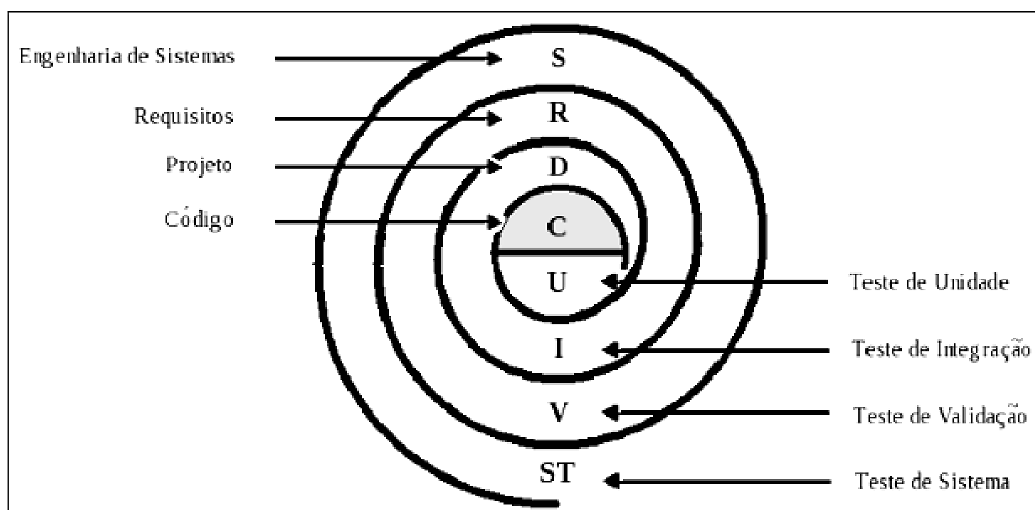

Figura 3.1: Espiral (teste de software) [18]

Neste trabalho o foco está no teste dinâmico, mais especificamente no teste de sistema. O teste de sistema é um conjunto de diferentes testes com o propósito de avaliar o sistema como um todo. Podemos subdividi-lo em quatro tipos: (i) O teste de recuperação que verifica a recuperação de um sistema diante uma falha. (ii)O teste de segurança que testa as medidas de segurança do software contra acessos indevidos. (iii) O teste de desempenho é idealizado para testar o desempenho do software em tempo de execução sob um cenário real de execução. (iv) Finalmente temos o teste de estresse que confronta o sistema com situações anormais de execução e analisa seu comportamento nessas circunstâncias [18].

Este trabalho está concentrado nos testes de desempenho e estresse que serão discutidos ao longo desse capítulo. Ambos os testes tem características próximas e por vezes o teste de desempenho acaba por ser uma etapa do teste de estresse. Essa combinação entre os dois testes possibilita encontrar situações de degradação e possíveis falhas do sistema [18].

3.1 Teste de desempenho e Benchmark

Assim como em outras áreas da ciência da computação testes de desempenho são amplamente usados em SGBDs. A aplicação desses testes beneficia-se das cargas de trabalho oferecidas pelos *benchmarks*. Ao longo de mais de 25 anos foram propostas várias abordagens para a realização de *benchmark*s a fim de encontrar uma carga de trabalho que verifique o ambiente real no qual o SGBD será inserido, possibilitando assim comparar vários SGBDs com o intuito de dar suporte à escolha do mais apropriado.

A comparação de desempenho de um ou mais sistemas através de medições é chamado de *benchmark*ing e analogamente o conjunto de cargas de trabalho usados para as medições são chamados de *benchmark*s. Geralmente a motivação é devido a questões de desempenho e custo [15].

Benchmarks podem ser divididos em dois grupos. O primeiro, que não diz respeito a SGBDs, considera basicamente questões de processamento numérico e desempenho de entrada e saída (E/S) do disco rígido, a exemplo do SPEC CPU2000 Benchmark[14] e do NAS Parallel Benchmark [10]. No segundo grupo enquadram-se *benchmarks* que visam avaliar SGBDs, como o DebitCredit[8] e de maneira geral os *benchmarks* criados pela *Transaction Processing Performance Council* (TPC)[21].

Embora diversas variações de *benchmarks* tenham sido usadas desde 1973 [15], o primeiro artigo registrado na literatura foi sobre o DebitCredit [8]. Esse *benchmark* busca reproduzir uma rede bancária distribuída e surgiu a partir da solicitação feita por um grande banco que pretendia adquirir equipamentos e sistemas para colocar todas as agências, caixas automáticos e clientes em um sistema *online*. Após seu lançamento, o DebitCredit se tornou o *benchmark* padrão do mercado e se manteve nesta posição até o lançamento do TPC em 1988.

O TPC refinou vários dos conceitos propostos pelo DebitCredit e padronizou algumas questões que na época fizeram com que os fornecedores de SGBD não autorizassem a realização de outros *benchmarks* em seus produtos.

3.1.1 DebitCredit

A especificação do DebitCredit é composta por um esquema de banco de dados, uma transação que conta com operações de inserção, seleção e atualização, e pelos componentes que simulam os clientes, os terminais e as agências. Sua arquitetura está representada na Figura 3.2.

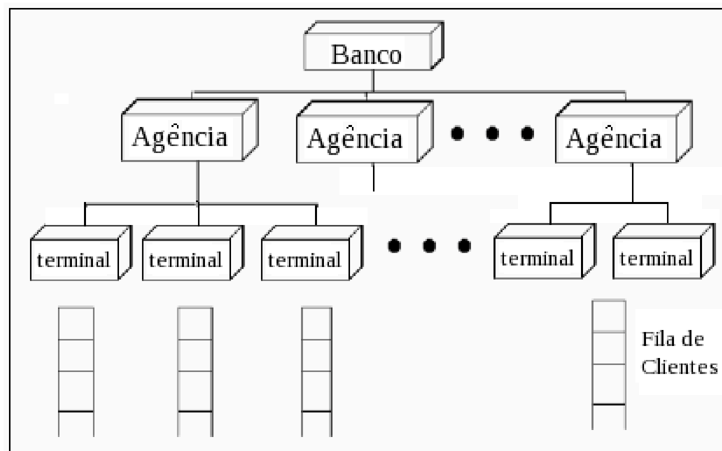

Figura 3.2: Arquitetura DebitCredit [8]

A base de dados para esse *benchmark* é bastante simples, sendo composta por três relações: agência, terminal e clientes. A transação utilizada também é simples. O seu pseudo código é apresentado na Figura 3.3 , com 4 tipos de registros no banco de dados: conta do cliente, terminal, agência e histórico.

Algoritmo 1: Transação DebitCredit	
Inicio transação	
Le mensagem do terminal	(100 bytes)
Atualiza conta do cliente	(100 bytes)
Escreve no histórico	(50 bytes, sequencial)
Atualiza terminal	(100 bytes)
Atualiza agência	(100 bytes)
Envia mensagem para o terminal	(200 bytes)
Encerra transação	

Figura 3.3: Transação DebitCredit [8]

O número de transações executadas por segundo (*tps*) é a métrica utilizada pelo *benchmark* para avaliar o desempenho do SGBD. Sistemas de tamanhos diferentes também

podem ser testados usando uma escala baseada no padrão proposto. Cada *tps* corresponde a 10 agências, 100 caixas e 100.000 clientes. Sendo assim, se por exemplo o sistema testado buscasse 50 *tps* deveria rodar o *benchmark* com 500 agências, 5.000 caixas e 5.000.000 de clientes. Para o sistema proposto o número esperado é de 100 *tps*.

O resultado da avaliação é dado por uma relação entre custo e desempenho ($/*tps*). O desempenho é medido através do *tps* desde que 95% de todas as transações tenham um tempo de resposta igual ou menor a 1 segundo. O tempo de resposta é o intervalo entre o último *bit* da linha de comunicação recebida e o primeiro *bit* da linha de comunicação enviada. O custo inclui todas as despesas em um período de 5 anos e inclui compra, instalação e manutenção de hardware e software [15][1]

Segundo DeWitt [8] , a partir da publicação desse *benchmark*, teve inicio uma disputa na especificação de *benchmarks* entre os grandes fornecedores de SGBDs que direcionou a indústria de SGBDs por mais de quinze anos.

3.1.2 TPC

A busca por um *benchmark* padrão culminou na criação do TPC no ano de 1988. O TPC foi fundado pela união dos esforços de 8 companhias. Os membros iniciais foram Control Data Corporation, Digital Equipment Corporation, ICL, Pyramid Technology, Stratus Computer, Sybase, Tandem Computers, Digital Equipment Corporation, ICLComputers e Wang Laboratories. Até o final do ano de criação esse número chegou a 26 companhias. Atualmente entre os membros do TPC estão os maiores fabricantes de SGBDs do mundo.

O TPC conta com 10 *benchmarks*, apresentados em ordem cronológica na Tabela 3.1, sendo que alguns são classificados como obsoletos.

Nesta seção falaremos brevemente das versões obsoletas e da versão em desenvolvimento e os *benchmarks* de versão corrente em uso serão abordados em seções individuais.

O primeiro *benchmark* lançado pelo TPC foi o TPC-A. O TPC-A é a versão TPC do *benchmark* DebitCredit com importantes modificações, como por exemplo:

[1] Os resultados publicados no artigo do DebitCredit não apresentavam o nome dos verdadeiros fornecedores dos SGBDs testados. Isso causou grande especulação na comunidade, porém os fornecedores sabiam qual era o seu resultado.

Tabela 3.1: Benchmarks TPC

Benchmark	Ano de criação	Aplicação	Estado
TPC-A	1989	OLTP	obsoleto
TPC-B	1990	Teste de Estresse	obsoleto
TPC-C	1992	OLTP	versão corrente
TPC-D	1995	Suporte a decisão	obsoleto
TPC-H	1999	Suporte a decisão	versão corrente
TPC-R	1999	Suporte a decisão	obsoleto
TPC-W	1999	Transações Web	obsoleto
TPC-APP	2004	Servidor de aplicações Web	obsoleto
TPC-E	2007	OLTP	versão corrente
TPC-DS	2008	Suporte a decisão	em desenvolvimento

- Nível de isolamento: A especificação do TPC-A exigia um controle extremamente rígido de nível de isolamento baseado nas propriedades ACID. No DebitCredit, o controle é baseado em log para manter a integridade das transações.

- Arquitetura: O DebitCredit se baseava em um sistema de Mainframe, enquanto o TPC-A propôs uma arquitetura baseada em um modelo cliente/servidor.

- Tempo de resposta (TR): O TR exigido pelo TPC-A foi mais flexível se comparado com DebitCredit. No DebitCredit é necessário que 95% das transações fossem executadas em 1 segundo, enquanto no TPC-A é suficiente que 90% das transações executassem em 2 segundos.

No ano seguinte ao lançamento do TPC-A foi lançado o TPC-B, que é considerado pelo TPC como um Teste de Estresse de banco de dados caracterizado por um número significativo de operações de entrada e saída (E/S) de disco rígido [21]. Este *benchmark*, por se tratar de um teste de estresse, será tratado com mais detalhes na seção 3.2.1.

O TPC-D, substituído posteriormente pelo TPC-H, é considerado obsoleto desde 1999. A proposta desse *benchmark* foi representar um sistema de apoio a decisão para sistemas que necessitam de consultas complexas e de longo tempo de execução em grandes estruturas de dados. Esse *benchmark* difere dos demais no sentido de que foi o primeiro *benchmark*, proposto pelo TPC, voltado para sistemas de suporte a decisão. Nele os requisitos empresariais de suporte a decisão foram traduzidos em 17 consultas complexas.

O TPC-R, o terceiro *benchmark* de suporte a decisão criado em 1999, é considerado obsoleto desde 2005. Esse *benchmark* é constituído por um grupo de regras de negócios orientado a consultas e modificações concorrentes de dados. As consultas e a base de dados são projetadas para dar suporte à simulação de grandes indústrias com características como análise de grandes quantidades de dados, consultas com alto grau de complexidade e de respostas a questões críticas e frequentes no mundo comercial [28].

Em 1999 foi criado o TPC-W, a primeira proposta de *benchmark* do TPC para simulação de transações Web. A carga de trabalho é executada em um ambiente de comércio eletrônico controlado que simula negócios orientados a um servidor de transações web.

O ambiente é caracterizado por múltiplas sessões de transações *online*, geração dinâmica de páginas web com acesso e atualização da base de dados, múltiplos tipos de transações, propriedades ACID, entre outras. A métrica de desempenho é baseada em transações web por segundo.

Assim como o TPC-R, o TPC-W é considerado obsoleto desde 2005 e esse foi substituído pelo TPC-APP.

O TPC-APP é um *benchmark* de comércio eletrônico criado em 2005 que simula uma livraria, na qual clientes fazem pedidos de livros que são entregues em endereço especificado. O banco de dados conta com 8 relações e uma carga de trabalho que simula transações *business-to-business* (transações eletrônicas realizadas entre empresas em um ambiente 24x7, ou seja, 24 horas por dia 7 dias por semana [22]).

A carga de trabalho simula servidores de aplicação comercial e bancos de dados associados a tais ambientes, que são caracterizados por: várias seções de negócios, utilização de documentos XML e protocolo SOAP [19] para troca de dados, gerenciamento de transações distribuídas, mensagens confiáveis e duráveis, banco de dados composto por tabelas com grande variedade de tamanho, atributos e relacionamentos, propriedades ACID e etc.

Este *benchmark* conta com duas métricas. A primeira é a SIPS (*Web Service Interaction per Second / Application Server System*, a qual mede a quantidade de serviços web que o sistema consegue processar por segundo. A segunda é a “Total SIPS”, que apresenta o número total de SIPS de todo o sistema testado associado ao preço do sistema.

O mais recente *benchmark* proposto pelo TPC é o TPC-DS, que simula uma grande empresa que gerencia, vende e distribui seus produtos. Esse *benchmark* ainda se encontra em desenvolvimento e a documentação disponibilizada pelo TPC é considerada parcial.

Assim como o TPC-H, o TPC-DS é um *benchmark* de suporte a decisão, porém o TPC-DS é composto por 99 modelos de consultas distintas e 12 operações de manutenção de dados. Apresenta uma carga de trabalho multiusuário e concorrente. O ambiente propõe as seguintes características [25]:

- Examinar grandes volumes de dados;
- Respostas a questões do mundo real de negócios;
- Executar consultas complexas;
- Gerar atividade intensa de processamento e acesso a disco;
- Prover sincronização com sistemas OLTP, por meio de funções de manutenção;

A carga de trabalho é dividida em 3 etapas: Carga de dados, execução de consultas e manutenção de dados, conforme a Figura 3.4.

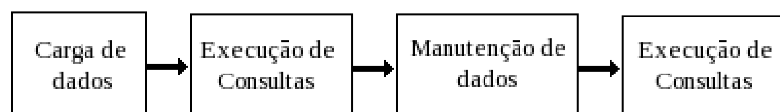

Figura 3.4: Etapas do TPC-DS [25]

De acordo com a documentação, os processos apresentados pelo TPC-DS são: Armazenar registros de aquisições de clientes, alterar preços de acordo com promoções, criar páginas web dinâmicas, manutenção de inventário e manutenção de perfil de clientes.

3.1.2.1 TPC-C

O TPC-C, aprovado em 1992, é a proposta de *benchmark* especificada pelo TPC para aplicações do tipo OLTP, sendo o modelo de referência para esse tipo aplicação. Se comparado com os *benchmarks* anteriores, ele é claramente mais complexo, contendo

transações variadas, que serão apresentadas a seguir, e uma estrutura de execução em geral mais complexa [24].

O TPC-C simula um sistema de entrada/entrega de pedidos, registro de pagamentos, verificação de estado de pedido e acompanhamento do nível do estoque. É formado por 5 transações diferentes de complexidade variada. O banco de dados é composto por nove tabelas e a métrica de desempenho é transações por minuto.

O TPC-C não se limita a testar o desempenho de um fornecedor de atacado, mas sim de toda indústria que deve administrar, vender ou distribuir um produto ou serviço qualquer que seja.

O TPC-C trabalha com transações de consulta, inserção e atualização. São 5 transações ao todo: *New-Order, Payment, Order-Status, Delivery* e *Stock-Level.*

- *New-Order*: Transação que representa as entradas no sistema. Transação executada com alta frequência e principal item de carga de trabalho submetida ao sistema.

- *Payment*: Transação que atualiza o crédito do cliente e estatísticas das tabelas DISTRICT e WAREHOUSE. Executada com grande frequência em sistemas *online.*

- *Order-Status*: Transação que verifica o estado do último pedido feito por um cliente. Transação de apenas leitura e de uso mediano de recursos computacionais. Tem baixa frequência de execução e tempo de resposta razoável, ou seja, 90% das transações respondem em até 5 segundos, requisito para operações /textitonline.

- *Delivery*: Faz o processamento das ordens de entrega, envolvendo operação de leitura e escrita no banco de dados. É executada com baixa frequência e assim como a transação order-status possui tempo de resposta razoável. Essa transação é executada em dois momentos: parte /textitonline e parte *batch.* O sistema agenda as ordens de entrega e grava em um arquivo. A parte *online* é responsável por gerar os dados da parte *batch.*

- *Stock-Level*: Essa transação seleciona itens que estão abaixo do limite no estoque e que foram vendidos recentemente. É uma transação considerada pesada. Possui

baixa frequência de execução e possui um tempo de resposta bem tolerante, ou seja, 90% das transações devem concluir em até 20 segundos.

As especificações do TPC-C definem um mínimo de 120 minutos de execução do *benchmark*, no entanto é recomendado que a execução seja de pelo menos 8 horas sem nenhum tipo de interrupção. Pelo menos 90% das transações do tipo *New-Order, Payment, Order-Status e Delivery* devem executar em 5 segundos, já para a Stock-Level o tempo é de 20 segundos. A métrica de desempenho do TPC-C é de produtividade de negócio, que considera o número de novas ordens inseridas, e é medida por transações por minuto (tpm).

3.1.2.2 TPC-H

Em contrapartida ao TPC-C citamos o TPC-H, *benchmark* para aplicações *Online Analytical Processing*(OLAP). Ele é o *benchmark* proposto pelo TPC para simular sistemas de Data Warehouse (DW). Simula um ambiente com grande volume de dados e consultas com alto grau de complexidade de um ambiente real [27].

A metodologia não modela um segmento de mercado específico, mas atividades que sejam comuns a vários segmentos. Ele prevê também requisitos quanto à transparência no acesso aos dados. Estes requisitos retiram da consulta qualquer necessidade de saber a localização dos dados.

O TPC disponibiliza um gerador de dados para povoar a base para o *benchmark* conforme a plataforma e a escala configuradas. O TPC-H possui vários fatores de escala (FS) divididas em 1 Gigabytes(GB), 10GB, 30GB, 100GB, 300GB, 1.000GB, 3.000GB, 10.000GB, 30.000GB, 100.000GB.

Para cada FS existe um número de sequências(S) de consultas que devem ser submetidas durante o teste como mostra a tabela 3.2.

Cada sequência é composta por 22 consultas classificadas pelo TPC com as seguintes características:

- Possuem alto grau de complexidade;

Tabela 3.2: Sequência de consultas por escala

FS	S
1	2
10	3
30	4
100	5
300	6
1000	7
3000	8
10000	9

- Possuem Vários tipos de acesso;
- Possuem natureza ad-hoc;
- Examinam grande porcentagem dos dados disponíveis;
- Diferem umas das outras;
- Possuem parâmetros que se alteram entre as execuções;

Existem 3 métricas principais definidas para esse *benchmark*: TPC-H *power*, produtividade, preço/desempenho.

A TPC-H *power* é usada para o cálculo de produtividade e é dada por:

$$Power@Size = \frac{3600 * FS}{\sqrt{\prod_{i=1}^{22} QI(i,0) * \prod_{j=1}^{2} RI(j,0)}}$$

Onde:

- FS é o fator de escala;
- QI é o intervalo em segundos de cada consulta de seleção;
- RI é o intervalo em segundos de cada consulta de atualização;
- variável 'i' corresponde as consultas de seleção;
- variável 'j' corresponde as consultas de atualização;

O Throughput@Size também é utilizado para o cálculo de produtividade e é dada por:

$$Throughput@Size = \frac{(S * 22 * 3600)}{T * FS}$$

Onde:

- S número de sequências dada em função do fator de escala;
- T é a medição do intervalo de tempo;
- FS é o fator de escala;

A métrica de produtividade é dada por:

$$QphH@Size = \sqrt{Power@Size * Throughput@Size}$$

E finalmente a métrica de preço/desempenho por:

$$Price_per_QphH = \frac{\$}{QphH@Size}$$

Onde $ é o custo de todo o sistema.

3.1.2.3 TPC-E

O TPC-E é uma proposta de *benchmark* OLTP que simula uma corretora. O foco do TPC-E é um banco de dados central que executa transações que realizam a interação entre cliente, mercado de ações e a bolsa de valores [26].

O banco de dados é formado por 33 relações classificadas em tabelas fixas, escalonadas ou progressivas. As tabelas fixas não alteram sua cardinalidade sendo assim independentes do tamanho da base de dados. As tabelas escalonadas tem sua cardinalidade em função do número de clientes, porém, durante a execução do *benchmark* seu tamanho fica inalterado, podendo apenas haver alteração de valores. As tabelas progressivas alteram de tamanho durante a execução do *benchmark* e tem sua cardinalidade inicial relacionada ao número de clientes.

O conjunto de transações do TPC-E é formado por 12 transações que efetuam inserção, atualização e exclusão de dados. Cada transação é descrita a seguir:

- *Broker-Volume*: Emula o processamento efetuado que apresenta o desempenho de cada corretor. É associada a geração de relatórios gerenciais, sendo uma transação que contem somente operações de leitura.

- *Customer-Position*: Obtém a situação atual de um cliente, tomando por base os valores de mercado de seus ativos. Transação do tipo somente leitura.

- *Market-Feed*: Monitora a atividade corrente do mercado para disponibilizar o preço atual dos ativos. Transação de escrita e leitura.

- *Market-Watch*: Monitora o estado global do sistema fornecendo ao cliente a tendência do mercado. Transação somente leitura.

- *Security-Detail*: Emula acesso a informações de um ativo em particular. Transação somente leitura.

- *Trade-Lookup*: Emula processo de recuperação de informações por um cliente ou corretor (e.g. análise geral do mercado, análise de desempenho de um ativo). Transação somente leitura.

- *Trade-Order*: Emula compra e venda de um ativo. Transação de escrita e leitura.

- *Trade-Result*: Emula a conclusão de um operação no mercado de ações e armazena no histórico para futuras consultas. Transação de escrita e leitura.

- *Trade-Status*: Efetua atualização do estado de um conjunto de operações. Transação somente leitura.

- *Trade-Update*: Emula pequenas atualização e/ou correções em um conjunto de operações. Transação de leitura e escrita.

- *Data-Maintenance*: Representa o processo periódico de manutenção dos dados estatísticos. Transação de leitura e escrita com baixa frequência.

- *Trade-Cleanup*: Emula o cancelamento de transações pendentes. Transação de leitura e escrita executada apenas uma vez durante o *benchmark*.

O TPC-E possui 3 métricas de desempenho: "Taxa de vazão"que representa a quantidade de transações do tipo *"Trade-Result"* executadas por segundo (*tpsE*). A segunda é "Preço/desempenho"que relaciona custo do sistema avaliado e *tpsE*. E a "data de disponibilidade", que informa quando os produtos avaliados estarão no mercado.

3.1.3 Set Query

O *Set Query* foi criado para dar suporte à tomada de decisões que necessitam de dados de desempenho relevantes para aplicações de dados de valores estratégicos [16]. Um exemplo é apresentado na figura 3.5. A consulta gera uma lista de emails para anúncio de uma nova revista de esporte direcionada ao público feminino que joga tênis.

```
SELECT NAME, ADDR FROM PROSPECTS
WHERE SEX = 'F'
AND FAMILYEARN > 40000
AND ZIPCODE BETWEEN 02100 AND 12200
AND EDUC = "COLLEGE"
AND HOBBY IN ( "TENNIS", "RACQUETBALL");
```

Figura 3.5: Exemplo Set Query [16]

O *Set Query* possui 4 características chave:

- Portabilidade: as consultas são definidas em SQL. No entanto é permitido o uso de outras linguagens sem influenciar nos resultados de desempenho. Os dados utilizados representam aplicações reais e são criados a partir de uma algoritmo portável que possibilita a criação do banco de dados em diferentes plataformas.

- Cobertura funcional: as consultas são escolhidas para suprir a necessidade de aplicações de dados com valor estratégico, possibilitando classificação de subconjuntos específicos para os sistemas que o usuário planeja usar.

- Cobertura de seletividade: As consultas variam em vários níveis de seletividade. A seletividade é dada pelo número de linhas em que uma consulta resulta. Uma consulta de baixa seletividade seria dada, por exemplo, por uma busca por clientes

do sexo feminino que resultaria em torno de 50%. Por outro lado uma consulta com alto grau de seletividade seria a busca por um cliente de CPF específico, a qual resultaria em uma única linha. O usuário pode concentrar suas medições baseadas em uma faixa de seletividade que lhe interessa.

- Escalabilidade: O banco de dados tem uma única tabela conhecida como *"tabela BENCH"*. Esta tabela pode conter múltiplos inteiros de um milhão de linhas com 200 *bytes* cada. O tamanho padrão da tabela corresponde a um milhão de linhas e segundo os autores [16] é o suficiente para que o *benchmark* simule o ambiente proposto.

A Tabela 3.6, apresenta um exemplo de tabela BENCH, que é composta por 13 colunas indexadas. No caso padrão, essas colunas são: KSEQ, K500K, K250K, K100K, K40K, K10K, K1K, K100, K25, K10, K5, K4 e K2. Doze colunas tem seus valores gerados randomicamente e variam de cardinalidade de 2 até 500.000. Cada coluna possui uma variação de cardinalidade que varia de 1 até a cardinalidade dada pelo nome da coluna. Por exemplo, a coluna K500K tem variação de 1 até 500.000, enquanto a coluna K2 varia entre 1 e 2, e assim de forma análoga, para todas as outras. A única coluna que não segue esse padrão é a KSEQ, chave primária da tabela, que possui seu valor sequencial de 1 até o número de linhas da tabela. A Figura 3.6 mostra as primeiras 10 linhas da tabela BENCH.

KSEQ	K500K	K250K	K100K	K40K	K10K	K1K	K100	K25	K10	K5	K4	K2
1	16808	225250	50074	23659	8931	273	45	4	4	5	1	2
2	484493	243043	7988	2504	2328	730	41	13	4	5	2	2
3	129561	70934	93100	279	1817	336	98	2	3	3	3	2
4	80980	129150	36580	38822	1968	673	94	12	6	1	1	2
5	140195	186358	35002	1154	6709	945	69	16	5	2	3	2
6	227723	204667	28550	38025	7802	854	78	9	9	4	3	2
7	28636	158014	23866	29815	9064	537	26	20	6	5	2	2
8	46518	184196	30106	10405	9452	299	89	24	6	3	1	1
9	436717	130338	54439	13145	1502	898	72	4	8	4	2	2
10	222295	227905	21610	26232	9746	176	36	24	3	5	1	1

Figura 3.6: Exemplo Tabela BENCH [16]

A métrica do *Set Query* é dada por *queries per second(QPS)* e por uma métrica de custo dada por *$PRICE/QPS*.

3.1.4 AS^{3}AP

O *ANSI SQL Standard Scalable and Portable (AS^{3}AP)* é um *benchmark* projetado para:

- Fornecer um conjunto abrangente para testes do poder de processamento do sistema sob teste (*SUT - System Under Test*);
- Criar um *benchmark* portável e escalável, de modo que possa testar uma ampla variedade de sistemas;
- Minimizar o esforço do usuário na implementação e execução do *benchmark*;
- Fornecer uma métrica uniforme, proporcional e que não dê margem à dupla interpretação;

O AS^{3}AP possui 2 módulos de teste: monousuário e multiusuário. O primeiro é destinado a testar os métodos de acesso e otimização de consultas básicas. O segundo, modela diferentes tipos de cargas de trabalho que simulam sistemas OLTP, cargas de trabalho de recuperação de informação e cargas de trabalho mistas [29].

A única medição executada por esse *benchmark* é o tempo decorrido de consulta e uma janela limite de 12 horas. Dados de desempenho de CPU e utilização de E/S devem ser coletados para uma análise mais profunda de desempenho. No entanto, eles não fazem parte da métrica de desempenho AS^{3}AP.

Todas as consultas são incorporadas em um programa juntamente com comandos de tempo necessários. Nos testes de multiusuários, os programas (processos) rodam concorrentemente, em scripts separados.

3.2 Teste de Estresse

O teste de estresse é uma variação do teste de desempenho. Este teste tem por objetivo colocar no sistema uma carga de trabalho além do esperado até o ponto de ruptura, ou seja, o ponto no qual o sistema começa a apresentar falhas ou deixa de responder novas requisições, sendo assim possível validar a corretude das funcionalidades e a degradação de desempenho do sistema.

3.2.1 TPC-B

O TPC-B é um de teste de estresse proposto pelo TPC, caracterizado por uma quantidade significativa de E/S em disco, moderado tempo de execução e integridade de transação. Os resultados obtidos pelo TPC-B não devem ser comparados com outros *benchmarks*, uma vez que ele não trata de sistemas OLTP ou similares, mas sim trata-se de um teste de estresse [23].

O ambiente de aplicação desse teste simula um sistema bancário que possui uma ou mais agências com múltiplos terminais e muitos clientes, cada um com uma conta. As transações simulam saques e depósitos realizados pelos clientes, e possuem o perfil apresentado na Figura 3.7.

A métrica utilizada pelo TPC-B é medida por transações por segundo ("*tps*"), respeitando um limite de tempo de residência, associado a uma relação preço/*tps*. A métrica é então chamada de "*tpsB*". Para seguir os padrões do TPC-B, os resultados devem conter tanto a taxa *tpsB* e a relação preço/*tpsB*.

Assim como em todos os *benchmarks* TPC, as propriedades ACID devem ser garantidas pelo sistema testado.

```
BEGIN TRANSACTION
   Update Account where Account_ID = Aid:
      Read Account_Balance from Account
      Set Account_Balance = Account_Balance + Delta
      Write Account_Balance to Account
   Write to History:
      Aid, Tid, Bid, Delta, Time_stamp
   Update Teller where Teller_ID = Tid:
      Set Teller_Balance = Teller_Balance + Delta
      Write Teller_Balance to Teller
   Update Branch where Branch_ID = Bid:
      Set Branch_Balance = Branch_Balance + Delta
      Write Branch_Balance to Branch
COMMIT TRANSACTION
Return Account_Balance to driver
```

Figura 3.7: Perfil de transação - TPC-B [23]

Segundo a definição do TPC-B, o teste deve ser realizado quando o sistema estiver no estado estável, onde o desempenho se mantém. Existem algumas exigências para a realização dos testes:

- Começar o teste após o sistema alcançar o *"steady state"*;
- Ser longo o bastante para que os resultados gerados possam ser reproduzidos;
- Executar de forma ininterrupta por pelo menos 15 minutos e não mais de uma hora;

O tempo de medição (TM) está compreendido durante o *"steady state"* e é durante este período que o *tps*B e o tempo de residência (RT) total e de cada transação são considerados. O tempo de residência de uma transação é dado por: $RT = T2 - T1$ onde T1 é o *timestamp* gravado antes da transação começar a sua execução e o T2 é o *timestamp* gravado depois que a transação terminou a execução.

Pelo menos 90% de todas as transações completadas durante o tempo de medição(TM) devem ter um tempo de residência menor que 2 segundos. Assim o "*tps*B"é dado por:

$$tpsB = \frac{\Sigma_transacoes}{TM}$$

A distribuição de frequência, em função do RT, das transações completadas durante o tempo de medição também deve ser relatada graficamente, como mostrado na Figura 3.8. O eixo Y corresponde ao número de transações e o eixo X o RT que varia de 0 a 5.

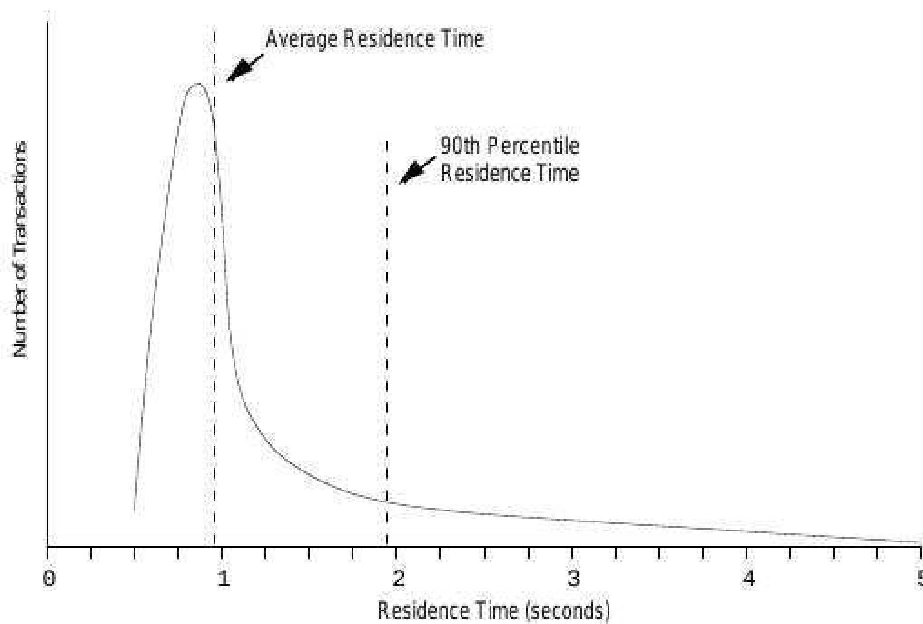

Figura 3.8: Distribuição de tempo de residência [23]

A métrica dada por preço/*tps*B, se baseiam no cálculo do preço do sistema sob 4 aspectos: Preço total da compra do sistema, preço de armazenamento, preço de manutenção e desenvolvimento do sistema. A própria documentação do TPC-B menciona que esses valores são muito relativos e variam muito sob vários fatores, fazendo com que essa métrica seja de difícil comparação.

3.3 Comparativo das Abordagens

A Tabela 3.3 é um resumo comparativo das principais abordagens de *benchmark* apresentadas, a forma em que são executados, os tipos de testes que podem derivar delas e suas métricas.

Tabela 3.3: Comparativo entre abordagens

Abordagem	distribuído	tipo de teste	métrica principal
Set Query	sim	desempenho	*qps*
AS^3AP	sim	desempenho	*qps*
TPC-B	não	estresse	*tps*
TPC-C	não	desempenho	*tpm*
TPC-H	não	desempenho	*tps*
TPC-E	sim	desempenho	*tps*

Observamos que em todas as abordagens apresentadas a métrica principal, de desempenho ou de estresse, é baseada na taxa de sucesso dos experimentos, ou seja, o número de transações e/ou consultas que o sistema consegue tratar em um determinado período de tempo. Métricas baseadas no número de transações tem maior aplicabilidade em sistemas transacionais, enquanto que as métricas que se baseiam em *consultas* são mais indicadas para sistemas de suporte a decisão.

Essa medição pode ser utilizada tanto em teste de desempenho quanto em um teste de estresse, contudo, o foco deve ser diferente, pois um teste de estresse o foco deve ser o ponto limite do sistema e como o mesmo se comporta frente a situações extremas de cargas de trabalho.

Outro fator importante é a necessidade do teste ser executado de forma distribuída, o que não é fato comum nas abordagens apresentadas, a execução de testes forma cen-

tralizada causa problemas de desempenho e contaminação de resultados causados pela própria arquitetura [7][5]. Segundo [4][30][1], testes de grande escala devem ser executados de forma distribuída.

Em sistemas de grande escala é difícil prever a carga de trabalho que o sistema irá enfrentar. Contudo, é possível avaliar o limite que este sistema possui de acordo com suas configurações de hardware e software. Logo, também é possível avaliar o seu comportamento quando exposto além desse limite. Geralmente, quando a carga de trabalho excede o limite do sistema, existe a degradação de desempenho (demonstrado no Capítulo 5).

As abordagens apresentadas não mostram como os sistemas se comportam neste contexto de estresse. Além disso, elas não apresentam uma metodologia que apóie a busca por problemas de degradação de desempenho que estes sistemas possam ter em ambientes de grande escala.

Podemos concluir, portanto, que não há uma metodologia de teste, que de fato, seja baseada em estresse e que possua métricas e resultados focados em ambientes transacionais de grande escala.

CAPÍTULO 4

METODOLOGIA INCREMENTAL DE TESTE DE ESTRESSE (MITE)

Neste capítulo apresentamos a Metodologia Incremental de Teste de Estresse (MITE), que tem por objetivo testar ambientes transacionais de grande escala. Esta metodologia busca ao mesmo tempo ser genérica e escalável, de forma que qualquer sistema que implemente o conceito de transação possa ser testado.

As métricas principais da MITE são a degradação de desempenho e os problemas (i.e., bugs) gerados pela carga de trabalho (i.e., estresse). A MITE possui um caráter incremental na execução dos testes. O uso de uma metodologia incremental possibilita testar o desempenho parcial do SUT (*System Under Test*) sobre uma grande variedade de cargas de trabalho.

Apresentamos então as especificações para implementação, a sequência de execução e as diferenças entre a nossa metodologia e a metodologia usada pelos benchmarks TPC-*like*.

4.1 Especificação do Banco de Dados

O esquema do banco de dados da MITE e as transações submetidas são baseadas no TPC-B, pois este é um teste de estresse e fornece uma estrutura de quatro tabelas suficiente para alcançar nosso objetivo. A Figura 4.1 apresenta o esquema, suas tabelas e relacionamentos.

Todo registro da tabela *branch* deve ter relação com um mesmo número de registros nas tabelas *teller* e *account*.

Para que cada transação represente uma quantidade similar de trabalho, é importante que todos os registros tenham o mesmo tamanho. Sendo assim, as tabelas são compostas de registros de mesmo tamanho e seguem a seguinte especificação.

Tabela *branch* deve ter registros de no mínimo 100 *bytes*, com os seguintes campos:

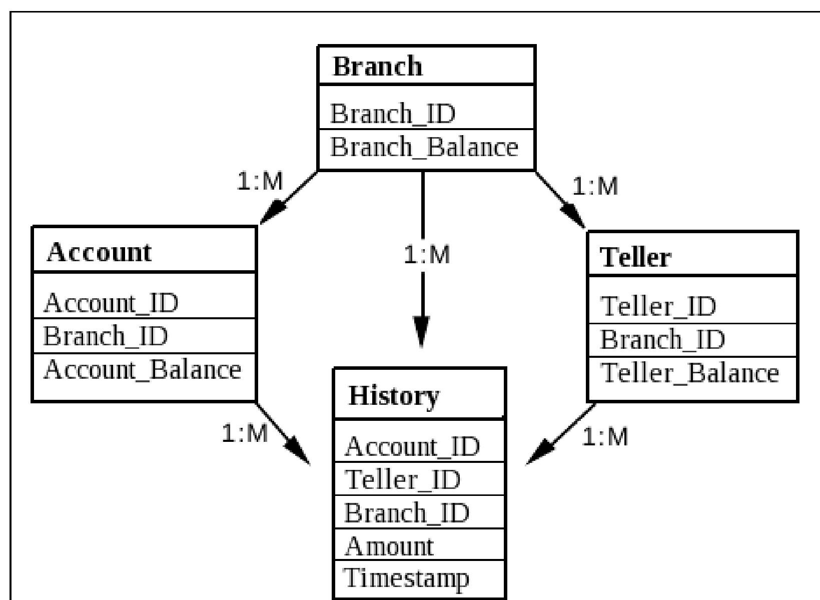

Figura 4.1: Esquema TPC-B [23]

- *Branch_ID*: Identificador único do registro.
- *Branch_Balance*: Campo numérico de 10 dígitos.

Tabela *teller* deve ter registros de no mínimo 100 *bytes*, com os seguintes campos:

- *Teller_ID*: Identificador único do registro.
- *Branch_ID*: Relacionamento com a tabela *branch*.
- *Teller_Balance*: Campo numérico de 10 dígitos.

Tabela *account* deve ter registros de no mínimo 100 *bytes*, com os seguintes campos:

- *Account_ID*: Identificador único do registro.
- *Branch_ID*: Relacionamento com a tabela *branch*.
- *Teller_ID*: Relacionamento com a tabela *teller*.
- *Account_Balance*: Campo numérico de 10 dígitos.

Tabela *history* deve ter registros de no mínimo 50 *bytes*, com os seguintes campos:

- *Account_ID*: Relacionamento com a tabela *account*.

- *Branch_ID*: Relacionamento com a tabela *branch*.
- *Teller_ID*: Relacionamento com a tabela *teller*.
- *Amount*: Campo numérico de 10 dígitos.
- *Time*: Campo do tipo *timestamp*.

A quantidade de registros por tabela segue a seguinte relação. Para cada registro na tabela *branch* são 10 registros em *teller* e 100000 em *account*. A tabela *history* tem relação direta com o número de transações executadas.[1]

As transações submetidas ao banco de dados seguem o padrão TPC-B e simulam saques e depósitos de clientes em um banco hipotético. Cada transação é composta de 5 ações:

- Atualizar conta do cliente
- Selecionar novo saldo
- Atualizar movimentação do caixa
- Atualizar movimentação da agência
- Gravar a operação de saque ou depósito no histórico.

4.2 Sequência de execução

A principal característica da MITE é o incremento de duas variáveis: complexidade e nível de criticidade do ambiente onde o SUT está inserido. A complexidade está relacionada às configurações de hardware, software e de arquitetura do sistema. Já o nível de criticidade diz respeito às variações do número de transações na carga de trabalho que o SUT deverá executar. A variação das configurações de complexidade e do tamanho da carga de trabalho é incremental, ou seja, executa inicialmente em um ambiente menos complexo até finalmente testar um mais complexo. O objetivo de executar de forma incremental é

[1] Os códigos de criação e preenchimento do banco de dados e da transação podem ser vistos na seção de anexos.

buscar erros relativos a diferentes escalas, mas principalmente os erros ligados às escalas nas quais começa a degradação do desempenho do SUT.

A avaliação do SUT é baseada na sua degradação, ou seja, quando o comportamento do SUT não corresponde à sua especificação e/ou configuração. A MITE possui 4 etapas, sequenciais, que são relativas à combinação das variáveis de complexidade e criticidade de forma incremental. As etapas são:

1. SUT executando com sua configuração padrão de instalação sob carga mínima de transações.

2. SUT executando com sua configuração padrão de instalação sob carga incremental de transações.

3. SUT otimizado até o limite de hardware e software sob carga mínima de transações.

4. SUT otimizado até o limite de hardware e software sob carga incremental de transações.

No primeiro passo, o objetivo é buscar erros de funcionalidade relacionados a instalação do SUT. No segundo passo, o objetivo é encontrar uma linha de base de desempenho com problemas de degradação sem a otimização do SUT. No terceiro passo, o SUT é otimizado e testado com um número mínimo de transações. Com isso buscamos erros típicos de degradação ligados a configurações errôneas. No quarto passo, o objetivo é encontrar a degradação que o SUT terá, e possíveis erros, relacionados à escala do teste.

Um exemplo de degradação pode ser dado por erros de conexão com o SGBD antes que o limite especificado na configuração seja alcançado. Os passos 2 e 4 devem ser executados de forma distribuída para, de fato, testar a escalabilidade do SUT. Portanto, o *driver* de teste não pode possuir limitações de escalabilidade.

Para alcançar o ponto de degradação, o número de transações submetidas ao SUT tende a ter uma grande variação e não tem propriamente um limite. Sugerimos que esse número varie em potências de 10.

4.3 MITE vs TPC-like

Os benchmarks aplicados em SGBDs, usados nos últimos 20 anos, tem como principal métrica de desempenho a relação entre o número de transações ou operações concluídas com sucesso em um determinado período de tempo (e.g., segundo, minuto, hora).

Essa métrica concentra-se em sistemas nos quais a carga de trabalho é conhecida ou já existe uma previsão de desempenho aceitável para incrementos dessa carga. No entanto, quando o objetivo é alcançar os limites do sistema, com tolerância a falhas e em cenários nos quais existe grande variação de carga, métricas baseadas em cargas fixas e em sucesso de transações ou operações não se mostram eficazes. Na realidade, estas métricas não consideram a possível degradação do SUT.

A análise de degradação é muito importante, principalmente em sistemas de grande escala, que geralmente estão expostos a cargas imprevisíveis de trabalho e com alta variação no número de transações. Por exemplo, cita-se o crescente uso do comércio eletrônico de compra e venda de produtos pela Internet. Esse tipo de comércio tem carga de trabalho com grande variação determinada por datas especiais (e.g., natal, dia das mães, dia dos pais), lançamento de novos produtos, dentre outros.

A MITE estende os benchmarks TPC-*Like* com grandes quantidades de transações na carga de trabalho sendo geradas de forma distribuída. Além disso, a MITE muda a abordagem de avaliação de desempenho (i.e., benchmarking) para teste de estresse, pois o objetivo é encontrar a degradação do sistema e não somente prover um índice de desempenho. A análise é realizada a partir do ponto onde o SGBD inicia a degradação, ou seja, a partir do ponto em que o sistema não consegue mais manter o seu comportamento padrão baseado em suas configurações.

4.4 Implementação

Nesta seção apresentamos a implementação da MITE aplicada em um SGBD de grande escala. Para a implementação da MITE sobre este SUT é necessário abordar duas características principais: o incremento de carga de trabalho e a sua execução distribuída.

O incremento da carga de trabalho deve ser executado de forma gradual e controlada. Para isso é empregado um conjunto de ações como sendo o caso de teste para implementação da MITE. Cada ação deste conjunto é responsável por executar um conjunto de transações simultâneas com o SUT, partindo de uma ação com uma carga mínima de transações até uma ação com uma carga extrema de transações.

Uma carga mínima de transações é facilmente implementada por um único nodo executando um conjunto pequeno de transações. Quando se deseja um número muito grande de transações simultâneas, é impossível sua implementação em um único nodo. Isto é devido ao tempo gasto para o gerenciamento e troca de contexto dos processos, ou *threads*, que executarão as transações. A solução então é uma execução distribuída desse conjunto de ações necessárias para empregar a MITE.

Com a utilização de um framework de teste de sistemas de grande escala, pode-se implementar e controlar esse conjunto de ações para serem executadas eficientemente de forma distribuída. Assim, foi empregado o PeerUnit que será apresentado em detalhes a seguir.

4.5 Uma visão geral do PeerUnit

PeerUnit [2] é um framework para teste de sistemas *peer-to-peer*(P2P), que combina testes de funcionalidade considerando parâmetros de escalabilidade e volatilidade do SUT. Consideramos como premissa que sistemas de grande escala possuem um comportamento similar ao de sistemas P2P e por isso a adoção do PeerUnit. Sistemas P2P são baseados em uma arquitetura distribuída, caracterizada pela descentralização das funções na rede, onde cada nodo realiza tanto funções de servidor quanto de cliente [3]. Como em P2P, pode-se verificar a corretude de um sistema de grande escala baseando-se em três dimensões: funcionalidade, escalabilidade e volatilidade [2].

A funcionalidade é o comportamento esperado do SUT mediante a aplicação de uma sequência de testes. A escalabilidade é a habilidade de manipular eficientemente o aumento da quantidade de nodos componentes do SUT. A volatilidade é a habilidade dos nodos saírem e entrarem no SUT sem interromperem seu funcionamento.

No PeerUnit, a dimensão "funcionalidade", é possível a partir da implementação da sequência de testes como um conjunto de ações bem definidas e controladas. A escalabilidade do SUT é acompanhada pelo PeerUnit pois ele possui uma implementação otimizada e simplificada. O PeerUnit ainda permite e controla a volatilidade dos nodos componentes do SUT. Esta característica é implementada no momento da definição da sequência de teste onde é possível especificar o momento do nodo entrar ou sair do SUT. Esta ação é especificada de forma determinística no caso do teste implementado.

O PeerUnit é implementado em Java e faz uso intensivo da reflexão dinâmica (*dynamic reflection*) e de anotações (*annotations*). Estas duas características são usadas para selecionar e executar as ações que compõem um caso de teste. A Figura 4.2 mostra os dois principais componentes da arquitetura do PeerUnit: o *testador* e o *coordenador*. O testador contém o conjunto de ações que são executados nos nodos componentes do SUT. O coordenador é executado em apenas um nodo, responsável pela sincronização da execução das ações dos casos de teste.

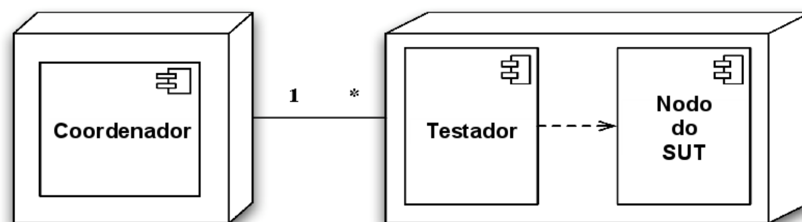

Figura 4.2: Arquitetura do PeerUnit [2]

O coordenador controla vários testadores e cada testador é executado sobre nodos distintos. O papel do testador é executar as ações do caso de teste e controlar a volatilidade de cada nodo. Após acabar a execução de todas as ações do caso de teste cada testador gera seu veredito local. O veredito local é um resultado conferido ao caso de teste aplicado ao SUT por nodo indicando a sua corretude ou não. O coordenador dispara as ações do caso de teste para os testadores e mantém uma lista de nodos não disponíveis. O coordenador então associa o veredito global baseado em todos vereditos locais informados pelos testadores. O veredito global é um resultado conferido ao SUT como um todo indicando a sua corretude.

De uma forma geral, o coordenador é o componente de controle responsável pelo gerenciamento dos casos de teste. Um caso de teste é implementado como uma classe, que é a classe principal da aplicação de teste. Um conjunto de teste pode conter vários casos de testes, que são implementados como um conjunto de ações. São utilizadas anotações para informar que um método é uma ação de um caso de teste, além de outras informações. As anotações disponíveis são:

- *TestStep*: especifica que o método é uma ação de um caso de teste;
- *BeforeClass*: especifica que o método é executado antes de cada caso de teste;
- *AfterClass*: especifica que o método é executado depois de cada caso de teste;

Todos os testadores recebem o mesmo caso de teste. Entretanto, eles executam apenas as ações designadas para eles, seguindo o que está definido pelas anotações. Cada ação é um ponto de sincronização: em um dado momento somente os métodos com a mesma definição serão executados nos testadores correspondentes. Assim as anotações possibilitam também determinar quais testadores executarão determinada ação do teste. Os testadores compartilham o mesmo caso de teste mas podem ter comportamentos diferentes. O objetivo é separar o teste de outros aspectos como a sincronização ou a execução condicional.

A execução das ações segue a idéia do protocolo de confirmação de duas fases (*two phase commit* - 2PC). Na primeira fase, o coordenador informa todos os testadores referenciados que uma ação pode ser executada, e aguarda. Assim que todos os nodos informam o fim da sua execução ao coordenador, é iniciada a execução da próxima ação. Se o tempo de espera (*timeout*) de uma execução é atingido, o caso de teste resulta em *inconclusivo*. Se qualquer *testador* tiver como resultado *falho*, o veredito global também resultará em *falho*. De outra forma, o resultado é analisado para decidir entre os vereditos: *sucesso* ou *inconclusivo*.

4.6 Implementação da MITE

Um caso de teste da MITE é implementado a partir de um conjunto de ações. Cada ação realiza um conjunto de transações que deve ser executada com o maior nível de concorrência possível. Para isso é necessário que vários nodos executem uma fração do número total de transações necessárias para gerar uma carga extrema de transações com o SUT. Em vez de uma ação com um grande número de transações em um único nodo, geramos uma mesma ação com um número menor de transações mas que será executada em vários nodos em paralelo, em um mesmo instante de tempo. Isso possibilita a execução de uma carga extrema de transações.

Assim foi definido o conjunto de ações apresentado na Tabela 4.1.

Tabela 4.1: Caso de teste da MITE

Sequência	Testadores	Ação
(a_1)	*	connect100();
(a_2)	*	connect200();
(a_3)	*	connect400();
(a_4)	*	connect500();
(a_5)	*	connect2000();
(a_6)	*	connect20000();

A primeira ação (a_1) a ser executada é a ação *connect100()* a qual realiza 100 transações com o SUT em cada nodo. Todas as ações serão executadas por todos os *testadores* de forma sequencial. é essencial que o *driver* de teste seja executado em máquinas clientes e não no servidor do SGBD, para evitar contaminação de desempenho [7]. Essa informação é apresentada pelo "*"presente na coluna testadores. Considerando então uma execução com 5 nodos obtêm-se 500 transações simultâneas com o SUT. Somente após a finalização dessa ação por todos os nodos é então executada a ação *connect200()* também de forma simultânea em todos os nodos. Têm-se então 1000 transações simultâneas com o SUT num exemplo com 5 nodos. Segue-se a execução até a última ação, *connect20000()*, que inicia 20 mil transações em cada nodo, totalizando 100 mil transações simultâneas.

Para executar várias transações em um mesmo nodo são utilizadas *threads*. Elas

permitem que um único programa execute várias tarefas diferentes ao mesmo tempo, de forma independente. Assim cada ação do caso de teste da MITE é um procedimento que criará um conjunto de *threads* de acordo com o estabelecido pela ação. Por exemplo na ação *connect100()* serão criadas 100 *threads*.

Para implementar esse caso de teste é apresentada a classe *TestMITE*, ilustrada na Figura 4.3. Essa classe é composta por seis métodos correspondentes a cada ação da tabela de ações do caso de teste da MITE. Acima da definição dos métodos existe a anotação *@TestStep* que significa que os métodos são identificados como ações do caso de teste.

```
public class TestMITE{

  @TestStep(order=1, range = "*", timeout = 100000)
  public void connect200(){
  ...
  }

  @TestStep(order=2, range = "*", timeout = 100000)
  public void connect400(){
  ...
  }

  ...

  @TestStep(order=6, range = "*", timeout = 1000000)
  public void connect20000(){
  ...
  }
}
```

Figura 4.3: ClasseTestMITE

O parâmetro *order* identifica qual a ordem de execução das ações. Como já dito, todas ações devem ser executadas por todos os testadores. Essa especificação é feita através do parâmetro *range* (i.e, range="*"), a partir do qual também é possível identificar apenas um ou alguns testadores que executarão determinada ação. O parâmetro *timeout* identifica o tempo de espera que o processo deve aguardar enquanto se executa a ação. Se esse parâmetro é alcançado aquela ação resulta como *inconclusiva*.

Dessa forma é possível implementar a MITE aplicando-a em um sistema de grande escala para sua avaliação. No capítulo seguinte é apresentado os experimentos realizados e os resultados obtidos com implementação da MITE.

CAPÍTULO 5

EXPERIMENTOS

Nesse capítulo são apresentadas as configurações de hardware e do SGBD, ou seja, o SUT que foi testado, além dos resultados alcançados com a aplicação da MITE. Com o objetivo de avaliar os limites do SUT, foram usadas configurações para alcançar o melhor desempenho de software e hardware.

A execução foi realizada baseando-se nas transações propostas pelo TPC-B, uma vez que o foco desse trabalho é teste de estresse. Para padronizar os experimentos o fator de escala do BD foi fixado em 10 a fim de limitar o número de variáveis.

5.1 Configuração do SGBD

O banco de dados foi implementado em um SGBD de código aberto, que nesse caso foi o PostgreSQL-8.4 [17]. Essa escolha foi baseada no dinamismo da comunidade. Esse SGBD possuí código aberto desde 1989 tendo um histórico sólido de respostas a *bugs*. Outro fato que também interferiu na escolha SGBD é que ele é usado pelo nosso grupo de pesquisa, a exemplo dos trabalhos [13][12].

Da configuração padrão do SGBD, apenas o parâmetro *max_connections* foi alterado. O valor padrão é tipicamente 100, e corresponde ao número máximo de conexões concorrentes que o SGBD deve aceitar. Para estes experimentos o valor foi fixado em 2000 conexões, limitado pela quantidade de memória compartilhada disponível [17]. Essa quantidade de memória é controlada pelo parâmetro SHMMAX do sistema operacional(SO).

O parâmetro SHMMAX determina o tamanho máximo de um segmento de memória compartilhada, e segundo a documentação do SGBD, recomenda-se o uso da seguinte fórmula para determinar um valor razoável:

$$SHMMAX = 250kB + 8.2kB * shared_buffers + 14.2kB * max_connections$$

- shares_buffers: define o espaço de memória alocado para o PostgreSQL armazenar as consultas SQL;
- max_connections: determina o número máximo de conexões concorrentes com o banco de dados;

5.2 Configuração de hardware

A máquina servidora na qual foi executado o SGBD durante os experimentos, que daqui em diante nominado como *chimay*, possui a seguinte configuração:

- Processador: Intel Xeon Processor W3570 Quad Core Processor
- Memória: 20GB
- Interface de rede Gigabit
- Disco rígido: 12 x 1 TB (RAID1+0)
- Sistema Operacional: Debian GNU/Linux, kernel 2.6.26-2-64

Para a simulação dos clientes foram usadas 6 máquinas de configurações diversas (Tabela 5.1), que a partir de agora serão tratadas como máquinas-cliente.

5.3 Comparação das Arquiteturas de Teste

O objetivo dessa comparação é mostrar a necessidade de uma arquitetura distribuída para testar SGBD de grande escala. Os primeiros testes realizados fazem um comparativo entre as arquiteturas de execução de teste baseadas na coordenação centralizada e distribuída. A Tabela 5.2 mostra a sequência incremental de carga de trabalho que foi submetida ao SGBD.

Tabela 5.1: Máquinas-cliente

Nome	Processador	Memória (GB)	SO
colombard	Dual-Core AMD Opteron(tm) Processor 2220	32	Debian GNU/Linux kernel 2.6.30
dalmore	Quad-Core AMD Opteron(tm) Processor 8387	56	Debian GNU/Linux kernel 2.6.34
bowmore	Quad-Core AMD Opteron(tm) Processor 8387	56	Debian GNU/Linux kernel 2.6.34
priorat	Quad-Core AMD Opteron(tm) Processor 8387	32	Debian GNU/Linux kernel 2.6.33
mumm	Intel(R) Xeon(R) Quad Core E5345 @ 2.33GHz	10	Debian GNU/Linux kernel 2.6.34
talisker	Intel(R) Core(TM) i7 CPU 975 @ 3.33GHz	5	Debian GNU/Linux kernel 2.6.34

Tabela 5.2: Teste de cargas incrementais

Teste	Número total de transações submetidas
1	500
2	1000
3	2000
4	3000
5	10000
6	100000

Esses testes foram realizados em um primeiro momento sobre uma arquitetura centralizada, na qual todas as transações são lançadas a partir de uma máquina-cliente, a *colombard*. Em seguida, após terminado o teste centralizado, distribuímos igualmente as transações entre as outras cinco máquinas-cliente para executar o teste de forma distribuída. É importante dizer que em todos os testes foram monitorados processos de *backend* e somente o postgreSQL estava rodando na máquina, portanto, processos do SO não tiveram influência nos resultados.

A Figura 5.1 mostra os resultados dessa primeira bateria de testes fazendo uma relação entre as transações submetidas e as finalizadas com sucesso de ambas as arquiteturas. Pode-se observar que o modelo centralizado tem uma limitação maior, com início de degradação a partir de cargas de trabalho superiores a 1000 transações. Isso acontece por uma limitação da arquitetura na criação de *threads* que simulam os clientes.

A arquitetura distribuída demonstra ser escalável e consegue enviar todas as transações

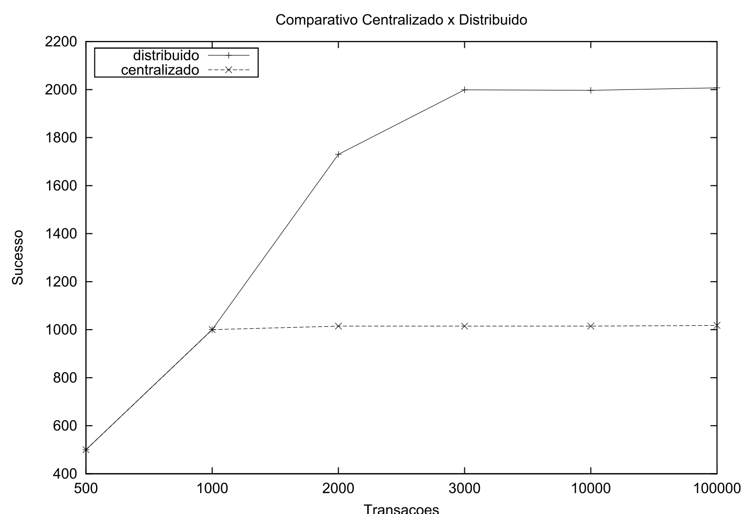

Figura 5.1: Teste comparativo de carga incremental

necessárias ao teste. A mesma arquitetura já foi utilizada e obteve ótimos resultados em ambientes P2P [6]. O SUT se estabiliza quando o número de transações é igual ou superior a 2000 transações. Esse comportamento é esperado, uma vez que o SGBD está configurado para aceitar 2000 conexões concorrentes. No entanto, veremos na próxima seção, em uma visão mais detalhada, que essa suposta estabilidade é uma falsa impressão.

Na próxima seção mostraremos testes executados somente sob a arquitetura distribuída, por ser a única arquitetura escalável que pode dar base para testes de grande escala.

5.4 Teste de Grande Escala

O objetivo dos testes realizados é mostrar a degradação do SUT e os erros gerados pelo estresse. Esta degradação será medida no número de transações executadas e no tempo decorrido para executar tais transações.

Os testes seguiram os passos da MITE onde incrementamos a carga de trabalho (número de transações) a cada nova execução do teste. Isto é, incrementamos o número de transações. Como parte da MITE o incremento foi realizado em potencia de 10 (i.e., 1000, 10000, 100000). A escolha desses valores possibilita uma fácil visualização da degradação do SUT.

A execução de testes do passo 1 da MITE não geraram erros, tendo o comportamento esperado quanto a funcionalidade padrão do SUT. O passo 2 foi importante para a definição da linha de base do número de transações para os experimentos de grande escala. Isto é, o número de transações que o SGBD executou com sucesso sem perdas ou erros. Esse número foi alcançado no teste de 500 transações.

A partir do teste com 1000 transações, ainda que com uma taxa pequena, começamos a observar erros de conexão como mostra a Figura 5.2. Esse teste está inserido no passo 3 da MITE.

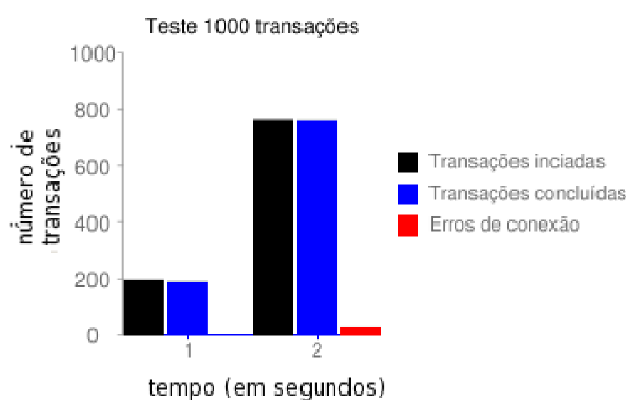

Figura 5.2: Teste 1000 transações

As 1000 transações foram submetidas ao BD em 2 segundos. O intervalo de tempo para análise é dividido em segundos. A distribuição das transações por intervalo de tempo é desigual devido ao *delay* de inicialização dos clientes. No primeiro intervalo foram submetidas 200 transações e todas foram atendidas e concluídas com sucesso. No intervalo subsequente foram 800 transações submetidas e dessas 33 não foram iniciadas devido a recusa de conexão por parte do BD. Essa taxa não era esperada visto que o BD foi configurado para atender 2000 conexões concorrentes. Um fato importante e que vale ser ressaltado é que não houve casos de transações abortadas, ou seja, todas as transações submetidas e aceitas pelo BD foram concluídas com sucesso com tempo de execução médio igual ou inferior a 1 segundo.

Os dois testes seguintes seguem o passo 4 da MITE, no qual o objetivo é realmente encontrar a degradação do sistema sob cargas extremas de trabalho. A Figura 5.3 mostra

o teste de 10000 transações.

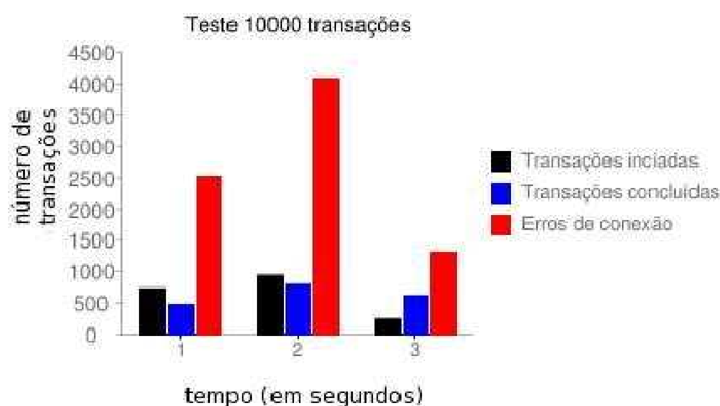

Figura 5.3: Teste 10000 transações

Nesse teste é possível observar um aumento no tempo de execução das transações. Na média, o tempo foi superior a 1 segundo chegando, no pior caso, a 2 segundos. Assim como no teste anterior não houveram transações abortadas. O número de conexões recusadas pelo BD teve níveis muito altos como mostra a Tabela 5.3. A taxa de erro, calculada para cada segundo de execução, se baseia no parâmetro *max_connections* do SGBD e no número de transações submetidas, aceitas e as ainda ativas. O cálculo é dado pela seguinte fórmula:

$$Taxa_de_erro = [submetidas - (aceitas + ativas)]/max_connections$$

- Submetidas: Transações submetidas em um determinado instante de tempo. (o valor máximo a ser considerado não deve ser superior a *max_connections*)
- Aceitas: Transações que iniciaram sua execução.
- Ativas: Transações que ainda não terminaram sua execução.

A soma das transações aceitas, e consequentemente de conexões aceitas pelo BD, é muito próxima de 2000, como observado na Tabela 5.3, o que dá a falsa impressão de que o parâmetro *max_connections* foi respeitado. No entanto, essas conexões não foram aceitas de forma concorrente, e é esse problema que a taxa de erro tenta mensurar. O esperado

Tabela 5.3: Taxa de erro

Tempo	Transações submetidas	Transações aceitas	Transações finalizadas	Taxa de erro
1	3304	761	508	0,61
2	5081	965	833	0,39
3	1615	271	656	0,48

é que o BD sempre atenda 2000 conexões concorrentes, o que resultaria em uma taxa de erro igual a 0, mas isso não acontece em nenhum momento, mostrando uma limitação do SGBD.

No teste de 100000 transações mostrado na figura 5.4 a degradação do BD fica mais evidente. O número de *"Erros de conexão"* chegou a ser superior a 10000, mas para facilitar a visualização no gráfico fixamos o teto em 2500 erros.

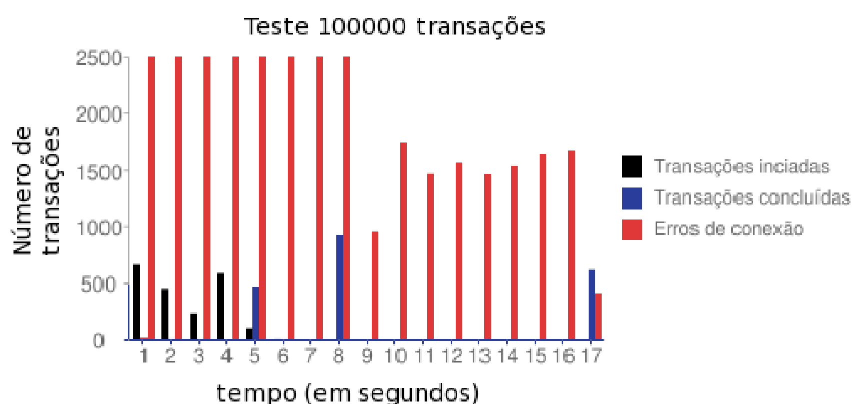

Figura 5.4: Teste 100000 transações

Neste teste a taxa de erro variou entre 0,001 e 0,69. A pior taxa acontece no segundo 9, que é o momento de início de um estado de *pane* do BD, no qual ele mantém 618 transações ativas e passa a recusar qualquer nova tentativa de conexão.

A Figura 5.5 mostra a degradação do tempo de execução das transações nos testes, que nos testes de 100000 conexões chegou a 11 segundos, tempo 5 vezes maior se comparado com às condições estáveis do SUT.

Esta comparação mostra que a carga de trabalho se mantém em um nível muito alto, o SUT passa a entrar em uma espécie de *thrashing*, situação em que grandes quantidades de recursos computacionais são usados para fazer uma quantidade de trabalho, com o sistema

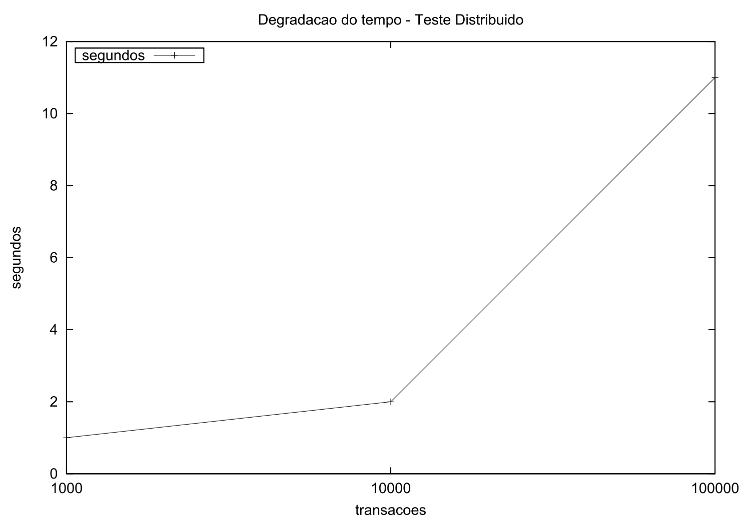

Figura 5.5: Degradação do tempo para concluir uma transação

em um estado contínuo de contenção de recursos. Porém não chega a ficar inutilizável, voltando ao estado normal quando a submissão das 100000 transações, por parte dos clientes, terminam de maneira normal.

Esse comportamento é prejudicial e não aceitável em sistemas de grande escala, onde a carga de trabalho não pode ser prevista de fato e se espera uma mínima estabilidade condizente com a configuração do SUT.

Os experimentos mostraram problemas enfrentados pelo SGBD quando exposto a ambientes extremos, ressaltando a importância de testes que busquem encontrar os limites do sistemas e possibilitem a prevenção diante destes cenários.

CAPÍTULO 6

CONCLUSÃO

Há uma deficiência dentre os benchmarks atualmente propostos em encontrar os limites e a degradação do SUT. Atualmente não existe uma técnica amplamente usada de teste de estresse que, em sistemas de grande escala, são muito úteis para descobrir erros ligados a grande carga de trabalho que esses sistemas são submetidos.

Neste trabalho apresentamos a MITE, uma metodologia incremental de teste de estresse que busca ser efetiva na avaliação de sistemas de grande escala.

O método incremental de carga de trabalho que a MITE propõe possibilita análises parciais de desempenho do SUT até que seus limites sejam alcançados, podendo ser usada por benchmarks que buscam ser efetivos em teste de estresse.

Os testes mostraram que a MITE foi eficaz em encontrar erros ligados ao estresse de um SGBD popular de código aberto, o PostgreSQL. Os quatro passos propostos foram essenciais na descoberta dos erros. Inicialmente, os 2 primeiros passos provaram ser importantes na definição da linha base dos experimentos, ou seja, qual o comportamento do SGBD com sua configuração padrão.

O passo 3 ajudou no processo de otimização do SGBD além de verificar configurações errôneas que possam surgir neste processo.

Finalmente, o passo 4 mostrou o estado de falha do SGBD quando seu limite de desempenho foi atingido. Uma vez atingido o limite, o mesmo recusa novas conexões o que sugere um comportamento normal. Contudo, após alguns segundos o SGBD conclui parte das transações, o que normalmente possibilitaria aceitar novas transações, mas o estado de *thrashing* não se altera, e impede a aceitação de novas transações.

Sistemas de grande escala são submetidos a ambientes com grandes variações de cargas de trabalho que dificultam ou até tornam impossível sua previsão. Dessa forma é necessário estar preparado para o limite. Sabendo as limitações do sistema é possível

se prevenir para casos extremos realizando, por exemplo, *upgrades de hardware* e/ou de tecnologia com antecedência.

6.1 Trabalho Futuro

Nesse trabalho apresentamos e testamos uma nova metodologia de teste de estresse em um SGBD relacional para fins transacionais. Acreditamos que a sua aplicação pode ser estendida a outras arquiteturas e a SGBDs que suportem outros outros modelos de dado, como orientação a coluna, por exemplo. Isto possibilitará a escolha mais adequada para cada aplicação.

Fica em aberto, a partir dos resultados, a geração automática de um índice, baseado em uma métrica de degradação do sistema, que possibilite a comparação de vários sistemas submetidos a estes testes.

O uso de um maior número de computadores nos testes, por exemplo execução do SUT em um cluster, pode ser planejado, no intuito de aumentar a simultaneidade e diminuir a sobrecarga das máquinas-cliente na criação de *threads*.

ANEXOS

```
/*
 *     This is a sample implementation of the Transaction Processing Performance
 *     Council Benchmark B coded in ANSI C and ANSI SQL2.
 *     Any equivalent implementation is equally acceptable.
 *
 *     Exceptions:
 *            1.     ANSI/ISO SQL has no explicit BEGIN WORK (begin transaction).
 *                   To show that the driver is outside the transaction,
 *                   explicit BEGIN WORK statements are included
 *            2      The C language has only integer and float numerics - it does not
 *                   support precision or scale.  So, in this implementation, money is
 *                   represented as integer pennies (pence, pfennig, centimes,...)
 *            3.     To clarify the schema, the following SQL2 features are used:
 *                          Primary Key
 *                          Foreign Key
 *                          DateTime datatype
 *                          Default values (to simplify handling of pad chars).
 *            4.     For simplicity, the program does no error checking or handling.
 */

/* Global declarations                                */
exec sql BEGIN DECLARE SECTION;

/* tpc bm b scaling rules
                                               */
long tps            =       1;   /* the tps scaling factor: here it is 1 */
long nbranches      =       1;   /* number of branches in 1 tps db       */
long ntellers       =      10;   /* number of tellers in  1 tps db       */
long naccounts      =  100000;   /* number of accounts in 1 tps db       */
long nhistory       =  864000;   /* number of history recs in 1 tps db   */

/* working storage                                    */

long i,sqlcode, Bid, Tid, Aid, delta, Abalance;

exec sql END DECLARE SECTION;

void CreateDatabase();
long DoOne(long Bid, long Tid,  long  Aid, long delta);
#include <stdio.h>
/* main program,    creates a 1-tps database:  i.e. 1 branch, 10 tellers,...
 *                    runs one TPC BM B transaction
 */
main()
{
       CreateDatabase();
       Abalance = DoOne(1,1,1,100);     /* add 100 to account 1 by teller 1 at branch 1,
                                           return new balance      */
}
```

```
/*
 *  CreateDatabase - Creates and Initializes a scaled database.    */
void CreateDatabase()
{

exec sql BEGIN WORK;             /* start trans to cover DDL ops  */
exec sql CREATE TABLE branches (
             Bid                 NUMERIC(9),  PRIMARY KEY(Bid),
             Bbalance            NUMERIC(10),
             filler              CHAR(88) DEFAULT SYSTEM
                                 );            /* pad to 100 bytes */
exec sql CREATE TABLE tellers (
             Tid                 NUMERIC(9), PRIMARY KEY(Tid),
             Bid                 NUMERIC(9)  FOREIGN KEY REFERENCES branches,
             Tbalance            NUMERIC(10),
             filler              CHAR(84) DEFAULT SYSTEM
                                 );            /* pad to 100 bytes */
exec sql CREATE TABLE accounts (
             Aid                 NUMERIC(9), PRIMARY KEY(Aid),
             Bid                 NUMERIC(9) FOREIGN KEY REFERENCES branches,
             Abalance            NUMERIC(10),
             filler              CHAR(84) DEFAULT SYSTEM
                                 );            /* pad to 100 bytes */
exec sql CREATE TABLE history (
             Tid                 NUMERIC(9) FOREIGN KEY REFERENCES tellers,
             Bid                 NUMERIC(9) FOREIGN KEY REFERENCES branches,
             Aid                 NUMERIC(9) FOREIGN KEY REFERENCES accounts,
             delta               NUMERIC(10),
             time                TIMESTAMP,
             filler              CHAR(22) DEFAULT SYSTEM
                                 );            /* pad to 50 bytes  */

/* prime database using TPC BM B scaling rules.
 *      Note that for each branch and teller:
 *            branch_id = teller_id  / ntellers
 *            branch_id = account_id / naccounts
 */
for (i = 0; i < nbranches*tps; i++)
      exec sql INSERT INTO branches(Bid,Bbalance) VALUES (:i,0);
for (i = 0; i < ntellers*tps; i++)
      exec sql INSERT INTO tellers(Tid,Bid,Tbalance) VALUES (:i,:i/:ntellers,0);
for (i = 0; i < naccounts*tps; i++)
      exec sql INSERT INTO accounts(Aid,Bid,Abalance) VALUES (:i,:i/:naccounts,0);
exec sql COMMIT WORK;
}                   /* end of CreateDatabase   */
```

```
/*
 *  DoOne - Executes a single TPC BM B transaction.
 */

long DoOne(long Bid, long Tid,  long  Aid, long delta)
{

exec sql BEGIN WORK;

exec sql UPDATE accounts
       SET          Abalance = Abalance + :delta
       WHERE        Aid = :Aid;

exec sql SELECT Abalance INTO :Abalance
       FROM         accounts
       WHERE        Aid = :Aid;

exec sql UPDATE tellers
       SET          Tbalance = Tbalance + :delta
       WHERE        Tid = :Tid;

exec sql UPDATE branches
       SET          Bbalance = Bbalance + :delta
       WHERE        Bid = :Bid;

exec sql INSERT INTO history(Tid, Bid, Aid, delta, time)
       VALUES (:Tid, :Bid, :Aid, :delta, CURRENT);

exec sql COMMIT WORK;

return ( Abalance);

}                   /* end of DoOne                  */
```

BIBLIOGRAFIA

[1] Eduardo Cunha Almeida, Gerson Sunye, e Patrick Valduriez. High performance computing for computational science - vecpar 2008. capitulo Testing Architectures for Large Scale Systems, páginas 555–566. Springer-Verlag, Berlin, Heidelberg, 2008.

[2] Eduardo Cunha de Almeida, Gerson Sunye, Yves Le Traon, e Patrick Valduriez. A framework for testing peer-to-peer systems. *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, páginas 167–176, Washington, DC, USA, 2008. IEEE Computer Society.

[3] Stephanos Androutsellis-theotokis e Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36:335–371, 2004.

[4] Francisco Brasileiro, Patricia Machado, Walfredo Cirne, e Alexandre Duarte. Gridunit: software testing on the grid. *Software Engineering, International Conference on*, 0:779–782, 2006.

[5] Eduardo Cunha de Almeida. *Testing and Validation of Peer-to-Peer Systems*. Tese de Doutorado, Université de Nantes, 2009.

[6] Eduardo Cunha De Almeida, João Eugenio Marynowski, Gerson Sunye, Yves Le Traon, e Patrick Valduriez. Efficient distributed test architectures for large-scale systems. *Proceedings of the 22nd IFIP WG 6.1 international conference on Testing software and systems*, ICTSS'10, páginas 174–187, Berlin, Heidelberg, 2010. Springer-Verlag.

[7] Murilo Rodrigues de Lima. Execução distribuída de benchmarks em sistemas de banco de dados relacionais. Dissertação de Mestrado, Universidade Federal do Paraná, 2008.

[8] David J. DeWitt e Charles Levine. Not just correct, but correct and fast: a look at one of jim gray's contributions to database system performance. *SIGMOD Rec.*, 37:45–49, June de 2008.

[9] Ramez Elmasri e Shamkant B. Navathe. Fundamentals of database systems, 2003.

[10] Adam Ferrari, Adrian Filipi-Martin, e Soumya Viswanathan. The nas parallel benchmark kernels in mpl. 1995.

[11] Jim Gray. The transaction concept: virtues and limitations (invited paper). *Proceedings of the seventh international conference on Very Large Data Bases - Volume 7*, VLDB '1981, páginas 144–154. VLDB Endowment, 1981.

[12] Pryscila Barvick Guttoski. Otimização de consultas no postgresql utilizando o algoritmo de kruskal. Dissertação de Mestrado, Universidade Federal do Paraná, 2006.

[13] Pryscila Barvik Guttoski, Marcos Sfair Sunye, e Fabiano Silva. Kruskal's algorithm for query tree optimization. *Database Engineering and Applications Symposium, International*, 0:296–302, 2007.

[14] John L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33:28–35, 2000.

[15] Raj Jain. *The Art of Computer Systems Performance Analysis*. 1991.

[16] Patrick E. O neil. The set query benchmark. *The Benchamark Handbook (2nd edition)*. Morgan Kaufmann, 1993.

[17] PostgreSQL. http://www.postgresql.org/about/, agosto de 2010.

[18] Roger S. Pressman. *Engenharia de Software*. Pearson Makron Books, 1995.

[19] Protocolo SOAP. Protocolo soap. http://www.w3.org/tr/soap12-part1/, agosto de 2010.

[20] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, e Pat Helland. The end of an architectural era: (it's time for a complete

rewrite). *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, páginas 1150–1160. VLDB Endowment, 2007.

[21] TPC. Transaction processing performance council, http://www.tpc.org/, agosto de 2010.

[22] TPC-APP. Transaction processing performance council. http://www.tpc.org/tpc_app/, agosto de 2010.

[23] TPC-B. Transaction processing performance council. http://www.tpc.org/tpcb/, agosto de 2010.

[24] TPC-C. Transaction processing performance council. http://www.tpc.org/tpc_c/, agosto de 2010.

[25] TPC-DS. Transaction processing performance council. http://www.tpc.org/tpcds/, agosto de 2010.

[26] TPC-E. Transaction processing performance council. http://www.tpc.org/tpce/, agosto de 2010.

[27] TPC-H. Transaction processing performance council. http://www.tpc.org/tpch/, agosto de 2010.

[28] TPC-R. Transaction processing performance council. http://www.tpc.org/tpcr/, agosto de 2010.

[29] Carolyn Turbyfill, Cyril U. Orji, e Dina Bitton. As3ap - an ansi sql standard scaleable and portable benchmark for relational database systems. *The Benchmark Handbook*, 1993.

[30] Thomas Walter, Ina Schieferdecker, e Jens Grabowski. Test architectures for distributed systems - state of the art and beyond. *Invited talk in: Testing of Communicating Systems (Editors: A. Petrenko, N. Yevtuschenko), volume 11, Kluwer Academic Publishers, 1998*, 1998.