MURILO RODRIGUES DE LIMA

EXECUÇÃO DISTRIBUÍDA DE BENCHMARKS EM SISTEMAS DE BANCOS DE DADOS RELACIONAIS.

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. Marcos Sfair Sunyé

CURITIBA

2008

MURILO RODRIGUES DE LIMA

EXECUÇÃO DISTRIBUÍDA DE BENCHMARKS EM SISTEMAS DE BANCOS DE DADOS RELACIONAIS.

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. Marcos Sfair Sunyé

CURITIBA

2008

MURILO RODRIGUES DE LIMA

EXECUÇÃO DISTRIBUÍDA DE BENCHMARKS EM SISTEMAS DE BANCOS DE DADOS RELACIONAIS.

Dissertação aprovada como requisito parcial à obtenção do grau de Mestre no Programa de Pós-Graduação em Informática da Universidade Federal do Paraná, pela Comisso formada pelos professores:

Orientador: Prof. Dr. Marcos Sfair Sunyé
Departamento de Informática, UFPR

Prof. Dr. Gerson Sunyé
Université de Nantes, França

Prof. Dr. Fabiano Silva
Departamento de Informática, UFPR

Prof. Dr. Marcos Alexandre Castilho
Departamento de Informática, UFPR

Curitiba, 14 de Agosto de 2008

AGRADECIMENTOS

A minha esposa Jacheline. Seu companheirismo, amor e apoio incondicionais foram de extrema importância.

Aos professores do Departamento de Informática, em especial ao meu orientador Marcos Sfair Sunyé, que trabalham a cada dia tornam melhores os cursos da área de Informática da Universidade Federal do Paraná.

A grande amigo Eduardo Cunha de Almeida. Suas dicas e conselhos foram cruciais.

A minha família que sempre acreditou em mim.

Aos meus amigos (do time de futebol, dos encontros dos finais de semana e todos os outros), que ajudaram a balancear o trabalho e a dedicação com a diversão.

Ao pessoal do laboratório de BD pela ajuda com a configuração dos equipamentos.

Aos outros companheiros mestrandos que da mesma forma que eu, batalharam muito para conseguir concluir este objetivo.

Ao pessoal da GVT que durante a época de aulas presenciais, permitiram que eu me ausentá-se do trabalho para estudar.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

As avaliações de desempenho, ou benchmark, de SGBDs são executadas a mais de 25 anos. Ao longo desses anos diversas metodologias foram propostas.

Os bancos de dados modelados para aplicações do tipo OLTP, em especial, foram alvo dos mais diversos benchmarks, entre eles os benchmarks DebitCredit e TPC-C.

A maioria dos benchmarks incluem cláusulas para a simulação de ambientes multi-usários, no qual diferentes usuários submetem transações, de maneira concorrente, contra o SGBD. Da mesma maneira, as ferramentas para a execução de benchmarks incluem funcionalidades para a simulação de ambientes multi-usuários.

Analisando algumas ferramentas de benchmarks, observamos que todas utilizam uma abordagem *centralizada* na simulação de ambientes multi-usuários, ou seja, todos os usuários simulados são executados em uma mesma máquina.

A execução centralizada, além de utilizar uma abordagem equivocada, introduz distorções na resultado final do benchmarck, uma vez que a simulação de vários usuários pode facilmente esgotar os recursos computacionais da máquina na qual o benchmark está sendo executado.

Neste trabalho, apresentamos uma alternativa à execução centralizada de benchmarks. O TPCC-C3SL é uma ferramenta para execução de benchmarks em SGBDs baseada na especificação do TPC-C. Com ela podemos executar benchmarks centralizados e distribuídos e comparar os resultados obtidos.

Com base no resultado dos testes, pudemos concluir que em uma simulação de até 30 usuários concorrentes, a execução centralizada apresenta um desepenho significativamente melhor em relação a execução distribuída. Porém, quando a simulação atinge 100 usuários concorrentes a abordagem distribuída apresenta um resultado muito melhor em relação a abordagem centralizada.

ABSTRACT

Performance analysis, or in other words, benchmark, of DBMS have been executed for more than 25 years. Along these years several methodologies were proposed.

Databases designed for OLPT applications were target of several benchmarks like DebitCredit and TPC-C.

Most of benchmark specification includes clauses for multi-user environment simulation, where different users submit transactions, in a concurrent way, against the database. In the same way, the benchmark tools include modules for simulating multi-user environments.

Analyzing some benchmarks tools we noticed that all of them use a centralized approach (execution) for multi-user environment simulation. In this approach all simulated users are executed in the same machine.

Beside the fact that the centralized execution evaluates the DBMS in a wrong way, it also introduces deviations in the benchmark's final result, by the fact that simulating several users can easily run out the resources of the machine in which the benchmark is running.

In this work, we present an alternative to the centralized benchmark execution. The TPCC-C3SL is benchmark tool based on the TPC-C specification. It enables the execution of both centralized and distributed benchmarks and also compares the results.

Based on the test's results, we concluded that in a simulation with 30 users (or less), the centralized execution shows results far better than compared to the results of the distributed execution. However, when as the number of uses grows and reach 100 concurrent users, the distributed execution shows a performance better than the centralized execution.

CAPÍTULO 1

INTRODUÇÃO

A análise de desempenho e mensuração de carga de trabalho em sistemas computacionais aparece como um tema importante dentro da Ciência da Computação, incluindo os ambientes em que se utiliza um sistema gerenciador de bancos de dados (SGBD).

Ao longo dos anos, pesquisadores propuseram diferentes formas de se analisar o desempenho de um SGBD. Um estudo bem elaborado em cima dessa questão, ajuda na tarefa de escolher o melhor SGBD para uma determinada necessidade.

De modo geral, é possível dividir as avaliações de desempenho de SGBDs, a qual nos referire-mos a partir desse momento como *benchmark*, em dois grupos. No primeiro grupo podemos enquadrar os benchmarks que consideram basicamente questões como processamento numérico e desempenho em operações de entrada e saída (E/S) dos discos rígidos, como o SPEC CPU2000 Benchmark [14] e NAS Parallel Benchmark [20]. No segundo grupo, se enquadram os benchmarks que procuram executar uma carga de trabalho mais abrangente do que simplesmente processamento numérico ou operações de E/. Na verdade, este grupo de benchmarks é baseado em algum nicho específico. Por exemplo, TPC-C executa uma carga de trabalho para ambientes transacionais; TPC-H [29] executa uma carga de trabalho para ambientes de apoio à decisão.

O primeiro benchmark proposto que se baseava nas exigências de sistemas comerciais foi o DebitCredit [1]. Este benchmark surgiu a partir da solicitação feita por um grande banco que pretendia adquirir equipamentos e sistemas para colocar todas as suas agências, caixas automáticos e clientes em um sistema *on-line*.

Após seu lançamento, o DebitCredit se tornou o benchmark padrão do mercado e manteve esta posição por mais de quinze quando o Transaction Processing Performance Council (TPC) [30] foi fundado.

O TPC refinou vários dos conceitos propostos pelo benchmark DebitCredit e padroni-

zou algumas questões que na época, fizeram com que os fornecedores de SGBD não autorizassem a realização de benchmarks em seus produtos, a não ser quando realizadas por eles próprios.

Dos esforços iniciais surgiu o TPC-A, uma especificação para benchmark de sistemas de bancos de relacionais baseada totalmente no DebitCredit. Um ano após o lançamento do TPC-A, o TPC fez algumas modificações e simplificações no TPC-A e disponibilizou uma nova versão do benchmark para SGBDs relacionais: o TPC-B.

Os dois benchmarks do TPC tomaram o lugar do DebitCredit como o benchmark padrão de mercado, e se mantiveram dessa forma até que foram substituídos pelo novo benchmark do TPC: o TPC-C.

Na pratica, cada vez que uma nova metodologia de benchmark é apresentada, existe também a necessidade de desenvolver a ferramenta que às executa [5, 21, 31].

Inicialmente, as ferramentas para execução de benchmarks eram desenvolvidas e utilizadas somente pelo autor do benchmark.

Após o lançamento do TPC-C, diversas ferramentas para execução desse benchmark começaram a ser desenvolvidos e disponibilizados para a comunidade. Entretanto, essas ferramentas tinham algum tipo de limitação em relação a sua utilização em diversas configurações de arquiteturas.

Em muitos casos essas ferramentas são desenvolvidas especificamente para um determinado SGBD. Isso significa que a lógica do benchmark é adaptada às características do SGBD escolhido e em alguns casos o benchmark é implementado dentro do próprio SGBD (e.g., através de *stored procedures*).

O fato de implementar uma ferramenta direcionada a um determinado SGBD nos apresenta dois problemas: (i) é impossível utilizá-la na avaliação de uma gama maior de SGBDs. (ii) a carga de trabalho tende a ser irreal. Otimizações podem ser implementadas para atender certos requisitos do próprio SGBD. Por exemplo, no caso de *stored procedures* o código seria compilado, otimizado e guardado na memória do SGBD e isso poderia distorcer os resultados da avaliação.

Todas essas ferramentas possuem uma funcionalidade que permite simular cargas de

trabalho em ambientes multi-usuário, porém, todas elas simulam os usuários através de *Threads*. [1]

Nestas ferramentas, cada usuário simulado - representado por um Terminal Remoto Emulado (RTE) - executa em uma nova Thread dentro do mesmo processador. Durante a execução de um benchmark esta funcionalidade introduz um grande problema: ela passa a idéia de que os usuários envolvidos na simulação submetem as transações contra o SGBD a partir de um mesmo computador. A utilização de muitas Threads exige muita memória e obriga o processador a realizar diversas trocas de contexto. Uma melhor abordagem para esse tipo de funcionalidade seria um cenário em que os RTEs submetessem as transações a partir de diferentes máquinas.

A partir dessas considerações, a execução do benchmark de forma distribuída se torna essencial, visto que a execução centralizada insere distorções na avaliação. A execução de um benchmarck de forma centralizada e com uma alta carga de trabalho (diversos RTEs por exemplo) fará com que o SGBD apresente uma desempenho extremamente baixo devido ao esgotamento dos recursos computacionais da máquina que executa o SGBD.

A execução distribuída elimina essas distorções pelo fato de a execução benchmark como um todo (RTEs e SGBD) fica espalhada por diversas máquinas na rede. Essa abordagem se aproxima mais da realidade encontrada em ambientes reais e ainda da especificação do TPC-C, a qual considera os RTEs como máquinas com capacidade de processamento que submetem as transações contra o SGBD. O TPC-C não considera mais o cenário em que um grande servidor serve diversos "terminais burros".

O objetivo deste trabalho é apresentar o TPCC-C3SL, um sistema distribuído que foi desenvolvido em linguagem de programação Java, para a especificação do TPC-C.

O TPCC-C3SL possui como diferencial a capacidade de ser executado de forma distribuída. Desta forma um SGBD sob avaliação poderá executar sem o impacto causado pelo gerenciamento de múltiplas Threads (i.e., troca de contexto, alocação de memória compartilhada, etc.). Ainda, esta arquitetura distribuída permite aumentar facilmente a

[1] Na programação concorrente, existem duas unidades básicas de execução: *Processos* e *Threads*. Os processos são unidades de execução que possuem uma área própria de memória. Threads são algumas vezes chamadas *processos leves*. Tanto os processos como as Threads proporcionam um ambiente de execução, porém, criar uma Thread exige menos recursos do que criar um novo processo [27].

escala do benchmark tornando possível, diferentes avaliações e cenários (e.g., sobrecarga, falhas de rede, etc.).

Além disso, esta ferramenta foi desenvolvida como software livre com o objetivo de ser facilmente mantida. Para atingir este objetivo, o desenvolvimento foi focado no uso de padrões e técnicas avançadas de programação a objetos. Foram utilizados também diversos *frameworks* consagrados para as tarefas de persistência de objetos em bancos de dados, e geração de trilhas de auditoria e logs.

Este trabalho está dividido em cinco capítulos. O capítulo 2 relata uma revisão da literatura de alguns dos benchmarks mais importantes já desenvolvidos. O capítulo 3 apresenta um sumário da especificação do TPC-C. O capítulo 4 apresenta algumas ferramentas desenvolvidas para a execução de benchmarks. O capítulo 5 descreve o TCC-C3S;. nossa proposta de ferramenta para execução de benchmarks baseada na especificação do TPC-C. O capítulo 6 descreve a configuração do ambientes de testes e a análise dos resultados obtidos. Finalmente, no capítulo 7 é apresentada uma conclusão do trabalho.

CAPÍTULO 2

BENCHMARKS

De acordo com o dicionário para a língua inglesa *The American Heritage*, a expressão *benchmark* significa mensurar algo de acordo com padrões especificados com o intuito de comparar e melhorar um determinado produto, teoria, entre outros. Ou seja, realizar um benchmark em um SGBD, consiste em executar uma série de cargas de trabalho com características bem definidas a fim de avaliar o desempenho do SGBD em questão.

Os benchmarks de SGBDs têm despertado os interesses de pesquisadores há mais de 20 anos. Ao longo de todos esses anos, diferentes benchmarks foram propostos, cada um procurando corrigir as deficiências e erros dos anteriores.

Nas próximas seções apresentaremos alguns dos principais benchmarks para sistemas de bancos de dados relacionais, procurando traçar uma linha cronológica entre as metodologias até chegar ao benchmark TPC-C.

2.1 Wisconsin Benchmark

Em 1983, foi proposta a primeira tentativa de desenvolver uma metodologia científica para a avaliação de desempenho de SGBDs [4].

O objetivo principal do benchmark Wisconsin era avaliar o desempenho dos operadores relacionais básicos, presentes em todas as consultas, desde as simples até as mais complexas.

O benchmark era composto da construção de base de dados sintética e da execução de um conjunto de consultas com finalidades variadas.

Existiam quatro relações básicas com diferentes cardinalidades: 1000, 2000, 5000 e 10000 tuplas. Todas as tuplas tinham 182 bytes de comprimento, assim, as quatro relações exigiam 4MB de espaço em disco. Frequentemente estas relações eram duplicadas, a fim proporcionar testes como a junção de duas relações com 10000 tuplas cada.

O conjunto de consultas selecionadas, nominadas de Wisconsin Benchmark, tinha por objetivo mensurar o custo das seguintes operações:

1. Seleção com diferentes níveis de seletividade
2. Projeção com diferentes percentuais de atributos repetidos
3. Junção simples e múltipla
4. Agregação simples e funções de agregação
5. Atualizações: inserção, exclusão e alteração.

Para avaliar o benchmark, foram selecionados três SGBDs convencionais: as versões comercial e acadêmica do *Ingres* e *Oracle*. Também foram avaliadas duas máquinas para sistemas de bancos de dados: *DIRECT* e Britton-Lee IDM/500. De acordo com DeWitt [10], os resultados produzidos pelo Wisconsin benchmark eram interessantes, porém controversos. Para DeWitt [9], o benchmark Wisconsin estava longe da realidade dos sistemas reais da época.

Segundo DeWitt [9], uma das maiores contribuições do benchmark Wisconsin foi acelerar o desenvolvimento do benchmark *Datamation*, o qual viria mais tarde a ser chamado de benchmark *DebitCredit*.

2.2 DebitCredit Benchmark

Dois anos após a criação do benchmark Wisconsin, o benchmark DebitCredit foi proposto no artigo *A Measure of Transaction Processing Power* [1].

O propósito do benchmark DebitCredit era imitar as operações que ocorrem quando um cliente de um banco faz um saque ou realiza um depósito. A construção desse benchmark foi motivada a partir de um *Requisição de Propostas* emitida por um grande banco que planejava colocar suas 1000 agencias, 10.000 caixas automáticos (ATMs) e seus dez milhões de clientes em um sistema on-line.

A base de dados para esse benchmark era bastante simples, sendo composta por três relações: agência, ATMs e clientes. A transação utilizada também era simples e compreendia os passos encontrados no algoritmo 1.

Algorithm 1: Transação do benchmark DebitCredit

Inicia uma transação.
Lê a mensagem proveniente do ATM.
Lê e atualiza o registro da conta do cliente.
Insere um registro de auditoria no arquivo de histórico.
Lê e atualiza o registro do ATM.
Lê e atualiza o registro da agência.
Envia a mensagem de retorno para o ATM.
Finaliza a transação.

O número destas transações executadas por segundo (tps) é a métrica utilizada pelo benchmark para avaliar o desempenho dos SGBDs.

Os autores propuseram ainda regras de escala para o tamanho do banco de dados. Se o desempenho esperado fosse de 100 tps, o banco de dados deveria ter 1.000 tuplas na relação de agências, 100.000 tuplas de ATMs e dez milhões de registros de clientes. Se algum fornecedor desejá-se atingir um desempenho de 1.000 tps, então o tamanho da base de dados deveria aumentar utilizando um fator de 10. Contudo, se o objetivo fosse atingir apenas 10 tps, então o tamanho da base de dados seria reduzido utilizando um fator de 10. As regras de escalas foram um dos fatores decisivos para o sucesso do benchmark, pois permitiam que todos os fornecedores participassem.

Diferentemente do benchmark Wisconsin, os resultados publicados no artigo não apresentavam os nomes verdadeiros dos fornecedores dos SGBDs avaliados. Isso causou grande especulação na comunidade, porém os fornecedores sabiam qual era o seu resultado.

Segundo DeWitt [10], a partir da publicação desse benchmark, teve inicio uma guerra de benchmark entre os grandes fornecedores de SGBDs que direcionou a indústria de sistemas de bancos de dados por mais de quinze anos.

2.3 Os benchmarks TPC-A e TPC-B

A partir dos resultados apresentados pelo benchmark Wisconsin, todos os grandes fornecedores de SGBD (com exceção da IBM), inseriram uma cláusula de "produto não passível de benchmark". Se um resultado de desempenho fosse necessário, esse deveria ser produzido pelo próprio fornecedor do SGBD.

O desejo pela existência de um benchmark padrão e que pudesse ter resultados publicados de maneira independente, levou a criação do Transaction Performance Council (TPC) [30]. No momento da formação (Agosto de 1988), oito empresas entre fornecedores de *software* e *hardware* aderiram ao conselho. No final do mesmo ano, o grupo já somava 26 representantes [23].

Uma das primeiras ações do TPC foi definir um consenso a respeito de uma série de regras que deveriam ser seguidas para que os resultados pudessem ser publicados. Além disso, o TPC refinou vários dos conceitos introduzidos pelo benchmark DebitCredit. O resultado disso foi o lançamento do TPC *Benchmark* A (TPC-A).

As grandes diferenças encontradas entre as especificações do DebitCredit e TPC-A são:

- **Nível de isolamento**: O DebitCredit utilizava um simples arquivo de log para manter a integridade da transação em caso de falhas. A especificação do TPC-A exigia um controle extremamente rígido se baseando nas propriedades de Atomicidade, Consistência, Isolamento e Durabilidade (ACID) [2]. Além disso, foi desenvolvido um conjunto de testes para comprovar cada uma destas propriedades.

- **Arquitetura do sistema**: os testes formulados pelo DebitCredit consideravam uma arquitetura na qual ATMs eram servidas por um *Mainframe* através de uma *Wide Area Network* (WAN). Já os integrantes do TPC estavam mais interessados em sistemas menores, como aqueles encontrados em *Local Area Network*s, ou até mesmo arquiteturas cliente/servidor.

- **Tempo de Resposta (RT)**: O TPC-A relaxou o RT exigido pelo DebitCredit. Enquanto que o DebitCredit exigia que 95% das transações deviam ser executadas

em até um segundo, para o TPC-A era suficiente um RT de dois segundos para 90% das transações.

Um ano após o lançamento do TPC-A, o conselho disponibilizou uma nova versão que continha algumas simplificações. Essa nova metologia de avaliação de SGBDs que recebeu o nome de TPC *Benchmark* B (TPC-B).

O TPC-B manteve o perfil das transações executadas e as mesmas exigências a respeito das propriedades ACID. A grande diferença foi a introdução de transações executadas em modo *batch*, processadas em paralelo às transações *on line*.

Em 1992 o conselho disponibilizou TPC-C [28], o primeiro benchmark produzido pelo conselho que não possuía ligação alguma com o benchmark DebitCredit. Decorridos três anos do lançamento do TPC-C, os dois primeiros benchmarks propostos pelo conselho foram totalmente aposentados.

2.4 Outros benchmarks

Além do benchmark DebitCredit, outras propostas se seguiram após a publicação do benchmark Wisconsin. O AS^{3}AP [31], por exemplo, é uma delas. Entre seus principais objetivos, o AS^{3}AP procura ser genérico o bastante a ponto de poder ser utilizado para avaliar uma ampla lista de SGBDs.

Além disso, o AS^{3}AP utiliza um conjunto de consultas baseadas apenas em instruções encontradas na especificação ANSI. O objetivo por traz dessa decisão era tornar o benchmark "portável", ou seja, capaz de avaliar diversas arquiteturas de sistemas.

O período após o lançamento do benchmark Wisconsin, tornou os benchmarks bastante populares. Até modelos de SGBD diferentes do OLTP, como o modelo OLAP (também conhecido como *Decision Support System* - DSS), possuía metodologias para avaliação desempenho. O benchmark *Set Query* [21], é um exemplo de benchmark para DSS.

CAPÍTULO 3

TPC-C

O TPC *benchmark*-C (TPC-C) [28] é a especificação proposta pelo TPC para a mensuração de carga de trabalho para aplicações do tipo OLTP.

Este benchmark mistura transações de apenas leitura com transações que fazem intensivas atualizações.

A métrica de desempenho do TPC-C é uma métrica de produtividade de negócio, considerando o número de novas ordens inseridas. Nas próximas seções apresentaremos algumas características do TPC-C.

3.1 Estrutura do banco de dados

O *benchmark* TPC-C é composto de uma série de transações projetadas para simular um típico sistema de aquisição de produtos, com as funcionalidades comunmente encontradas nessas aplicações, tais como criação de novos pedidos, realização de pagamentos, registro de saída de produtos, verificação do estado de um pedido, entre outras.

A empresa representada no *benchmark* é um fornecedor atacadista que possui determinado número de distritos de vendas e lojas associadas geograficamente distribuídas. Conforme a empresa cresce, novas lojas e distritos são criados. Cada distrito tem capacidade para atender 3000 clientes. Existem também os armazéns regionais, sendo que cada armazém atende 10 distritos. Todas as lojas mantêm estoque de 100K produtos que são vendidos pela empresa. A figura 3.1 ilustra essa hierarquia.

De maneira geral, a cardinalidade das tabelas varia em função do número de registros na tabela `WAREHOUSE`. Apenas a tabela `ITEM` possui um número fixo de registros. O relacionamento e a cardinalidade entre as tabelas é representada pela figura 3.2.

O TPC-C determina que um conjunto mínimo de dados deve ser carregado nas tabelas antes do início da execução do benchmark. Essa etapa é chamada de carregamento inicial

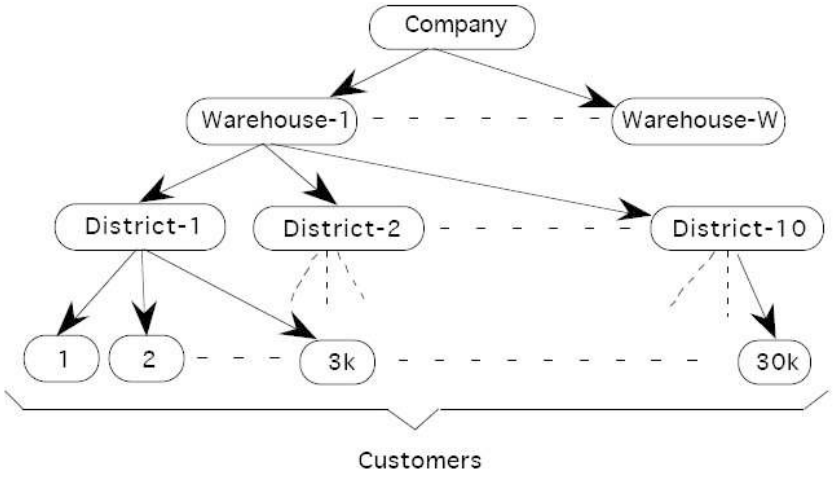

Figura 3.1: Hierarquia das relações do TPC-C

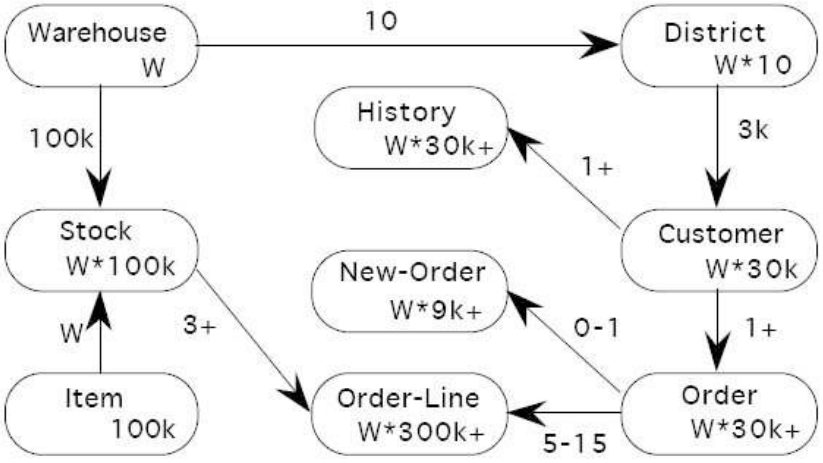

Figura 3.2: Diagrama de entidade-relacionamento do TPC-C.

do banco de dados.

A especificação define a maneira como cada coluna de cada tabela deve ser carregada e o número de registros que devem ser inseridos. A tabela 3.1 apresenta a quantidade de tuplas que cada tabela deve conter para cada `WAREHOUSE` configurado.

Tabela	**Tuplas**	**Tamanho da tupla (em bytes)**	**Tamanho da tabela (em 1K bytes)**
`WAREHOUSE`	1	89	0,089
`DISTRICT`	10	95	0.950
`CUSTOMER`	30K	655	19.650
`HISTORY`	30K	46	1.380
`ORDER`	30K	24	720
`NEW-ORDER`	9K	8	72
`ORDER-LINE`[a]	300K	54	16.200
`STOCK`	100K	306	30.600
`ITEM`	100K	82	8.200
Total de tuplas	610K		
Tamanho do banco de dados			76823,048

[a]É permitida uma diferença de 1% em função da variação aleatória encontrada no carregamento dos dados.

Tabela 3.1: Cardinalidade inicial das tabelas do TPC-C

3.2 Consistência e Isolamento

Jim Gray afirma em seu artigo *The Transaction Concept: Virtues and Limitations* [11] que uma transação é a transformação de um estado que possui as propriedades de atomicidade (tudo ou nada), durabilidade (efeitos resistem a falhas) e consistência (transformação correta).

Gray e Reuter mencionam no livro *Transaction Processing: Concepts and Techniques* [12] outro conceito: isolamento (concorrência na transformação). Essas quatro propriedades formam a sigla ACID (Atomicidade, Consistência, Isolamento e Durabilidade).

Segundo Gray [11], a consistência é a propriedade que uma aplicação exige que, ao se executar uma transação contra um SGBD, este banco de dados migra de um estado consistente para um outro estado consistente, considerando que inicialmente o SGBD

estava em um estado consistente.

A metodologia do TPC-C define doze testes para identificar o nível de consistência do SGBD. Após o carregamento inicial dos dados, o qual deve seguir a especificação de população inicial do SGBD, todos os doze testes devem apresentar resultado de que o banco de dados está num estado consistente.

O isolamento define quando e como as alterações feitas por uma operação em um registro se tornarão visíveis para outras operações concorrentes [11, 24, 22].

De acordo com especificação ANSI-SQL92 [2], o isolamento, ou nível de isolamento, é definido em termos de fenômenos que podem acontecer em transações concorrentes. Os níveis de isolamento são:

- **F0** (*Dirty Write*) Uma transação T_1 lê um elemento e o modifica. Uma transação T_2 modifica ou apaga o mesmo elemento e realiza um *commit*. Se T_1 tentar ler novamente este elemento, ela pode receber o valor alterado por T_2 ou descobrir que o elemento não existe mais.

- **F1** (*Dirty read*) Uma transação T_1 modifica um elemento. Uma transação T_2 lê o mesmo elemento antes de T_1 realizar *commit*. Se T_1 realizar um *rollback*, T_2 terá lido uma valor que nunca foi gravado e dessa maneira, nunca existiu.

- **F2** (*Non-repeatable Read*) Uma transação T_1 lê um elemento. Uma transação T_2 modifica ou apaga o mesmo elemento e realiza um *commit*. Se T_1 tentar reler este elemento, ele poderá receber o valor modificado ou descobrir que o elemento foi removido.

- **F3** (*Phantom*) Uma transação T_1 lê um conjunto de valores N a partir de um critério de busca. Uma transação T_2 executa uma série de comandos e produz um ou mais elementos que satisfazem a mesma condição de busca utilizada por T_1. Se a transação T_1 refizer a leitura inicial como o mesmo critério de busca, ela receberá um conjunto de valores diferentes.

O nível de isolamento define quais fenômenos pode acontecer em determinado nível. A tabela 3.2 apresenta uma matriz com os fenômenos permitidos para cada nível. Por

exemplo, o nível de isolamento dois determina que o único fenômeno que pode ocorrer é o *Phantom*.

Nível de isolamento	**F0 (*Dirty Write*)**	**F1 (*Dirty read*)**	**F2 (*Non-repeatable Read*)**	**F3 (*Phantom*)**
0	Impossível	Possível	Possível	Possível
1	Impossível	Impossível	Possível	Possível
2	Impossível	Impossível	Impossível	Possível
3	Impossível	Impossível	Impossível	Impossível

Tabela 3.2: Níveis de isolamento

Com estes conceitos definidos, a especificação do TPC-C exige os níveis de isolamento encontrados na tabela 3.3.

Req. #	**Descrição**
1.	Isolamento nível 3 entre as transações *New-Order*, *Payment*, *Delivery* e *Order-Status*.
2.	Isolamento nível 2 para as transações *New-Order*, *Payment*, *Delivery* e *Order-Status* relativas a qualquer outra transação.
3.	Isolamento nível 1 para a transação *Stock-Level* relativa às transações do TPC-C e qualquer transação arbitrária.

Tabela 3.3: Níveis de isolamento exigidos para as transações do TPC-C

O TPC-C sugere ainda uma lista de nove testes que devem ser realizados a fim de que as exigências definidas na tabela 3.3 sejam cumpridas pela aplicação.

3.3 Terminais Remotos

A especificação do TPC-C define um terminal como um dispositivo de interface que permite tanto a entrada de dados no sistema como a apresentação de informações para o usuário. A apresentação das informações para o usuário acontece por meio de formulários, sendo um para cada tipo de transação do TPC-C.

No contexto do *benchmark* o terminal remoto emulado (RTE) simula um usuário real que submete transações para o SGBD. Cada RTE está *conectado* a um *Warehouse*, o qual é chamado de *homeWarehouse*. Cada *Warehouse* pode aceitar conexões de no máximo dez RTEs. Uma vez que o RTE se "conecta" no *Warehouse* ele executa um ciclo de

operações composto de apresentações de telas, tempos de espera e de resposta (RT). A figura 3.3 ilustra o ciclo de operações realizado pelo RTE.

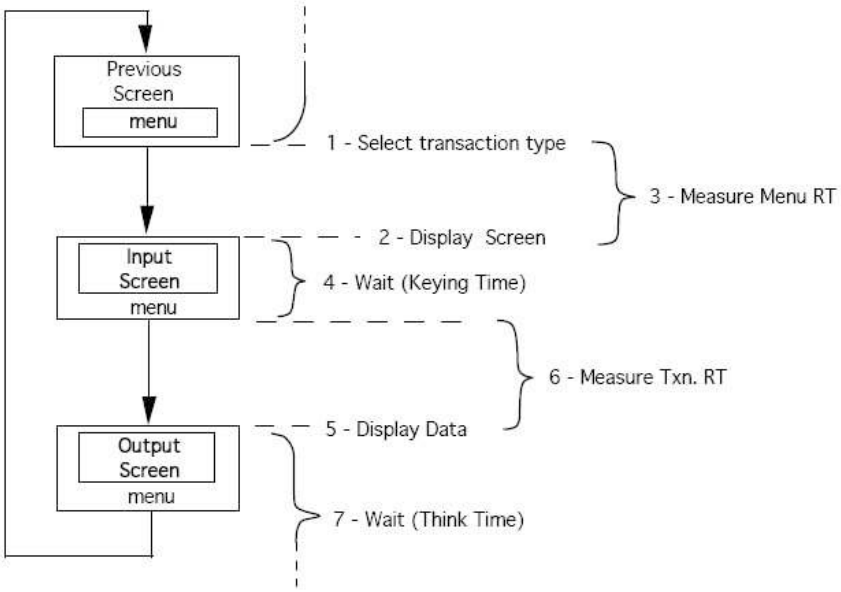

Figura 3.3: Ciclo de operações executadas pelo RTE.

Observando a figura 3.3 podemos identificar dois passos (#2 e #5) em que uma tela é apresentada pelo RTE para o usuário final. Cada tipo de transação possui uma tela padrão para apresentação dos dados. A especificação do TPC-C exige que as telas sejam apresentadas da maneira definida. A especificação aceita pequenas diferenças do formato original, porém estas devem ser justificadas. No apêndice A é possível visualizar o formato de cada uma das telas de E/S.

No passo #2 o RTE deve selecionar de maneira uniforme e aleatória, um dos cinco tipos de transações do TPC-C. A especificação determina que os RTEs devam manter um equilíbrio mínimo entre as transações executadas. A tabela 3.4 relaciona o percentual mínimo de execução para cada uma das transações.

O objetivo da aplicação de percentuais mínimos para cada transação é fazer com que a cada transação *New-Order* executada, uma transação *Payment* também seja executada, e para cada dez transações *New-Order* executadas, uma *Order-Status*, uma *Delivery* e uma *Stock-Level* sejam executadas.

Transação	Percentual
New-Order[a]	NA
Payment	43
Order-Status	4
Delivery	4
Stock-Level	4

[a]não existe um percentual mínimo para esta transação

Tabela 3.4: Percentual mínimo de execução das transações do TPC-C

3.4 Transações

O TPC-C define cinco tipos de transações que devem ser submetidas contra o SGBD durante o período de avaliação. A transação define uma unidade de trabalho na qual um conjunto de operações deve ser realizado de forma atômica, ou seja, todas as operações dentro da transação devem executar com sucesso para que um *commit* seja realizado.

Para explicar o funcionamento das transações, precisamos definir alguns termos:

- **Selecionar** significa identificar, ou obter referencia, para um ou mais registros de uma tabela, porém sem recuperar valores.
- **Recuperar** se refere ao fato de acessar o dado de um atributo e o enviar para a aplicação.

Nas subseções seguintes, descreveremos o funcionamento de cada transação.

3.4.1 Transação *New-Order*

A transação *New-Order* representa a entrada de uma nova ordem no sistema. Esta é uma transação considerada leve, não consumindo muitos recursos computacionais. Ela possui alta freqüência de execução e é o item principal de carga de trabalho submetida ao SGBD. As operações executadas nesta transação podem ser observadas no algoritmo 2.

3.4.2 Transação *Payment*

A transação *Payment* atualiza o crédito de um cliente e reflete este pagamento nas estatísticas das tabelas `DISTRICT` e `WAREHOUSE`.

Algorithm 2: Transação *New-Order*

Input: $homeWarehouse$, valor que indica a qual *Warehouse* o RTE está conectado
Output: $NewOrderResponse$, objeto de retorno da transação *New-Order*
Inicia uma transação.
Seleciona um registro da tabela `WAREHOUSE` com recuperação de dados.
Seleciona um registro da tabela `DISTRICT` com recuperação e atualização de dados.
Seleciona um registro da tabela `CUSTOMER` com recuperação de dados.
Calcula *olCnt* como o número de itens que a ordem terá.
Insere um registro na tabela `ORDER`.
Insere um registro na tabela `NEW-ORDER`.
for $i <= 1$ *to olCnt* **do**
  Seleciona um registro na tabela `ITEM` com recuperação de dados.
  Seleciona um registro na tabela `STOCK` com recuperação e atualização de dados.
  Insere um registro na tabela `ORDER_LINE`.
end
Finaliza a transação.

É uma transação leve (não consumindo muitos recursos computacionais) e com alto índice de execução. Além disso, o RT deve ser pequeno para atender requisições *on-line*[1].

Esta transação realiza acesso na tabela `CUSTOMER` através de uma chave não primária[2].

As operações executadas nesta transação podem ser observadas no algoritmo 3.

Algorithm 3: Transação *Payment*

Input: $homeWarehouse$, valor que indica a qual *Warehouse* o RTE está conectado
Output: $PaymentResponse$, objeto de retorno da transação *New-Order*
Inicia uma transação.
Seleciona um registro da tabela `WAREHOUSE` com recuperação e atualização de dados.
Seleciona um registro da tabela `DISTRICT` com recuperação e atualização de dados.
Seleciona um registro da tabela `CUSTOMER` com recuperação e atualização de dados[3].
Insere um registro na tabela `HISTORY`.
Finaliza a transação.

3.4.3 Transação *Order-Status*

A transação *Order-Status* verifica o estado da última *Order* criada por um *Customer*.

É uma transação de apenas leitura e mediana em relação ao consumo de recursos computacionais, pelo fato de que para obter a última *Order*, a aplicação obriga o SGBD a

[1]Para cada tipo de transação, pelo menos 90% delas deve ter um RT de cinco segundos, com exceção da transação *Stock-Level*, na qual o RT é de vinte segundos

[2]Em 40% das execuções dessa transação, o *Customer* será selecionado através de uma combinação de atributos que não compõe uma chave primária, estrangeira ou um índice. Dessa forma o SGBD será obrigado a realizar um *full scan* na tabela `CUSTOMER`

selecionar um conjunto de registros ordenados decrescentemente pela data de criação do registro e recuperar apenas a primeira ocorrência do conjunto selecionado.

Além disso, essa transação realiza acesso na tabela `CUSTOMER` através de uma chave não primária[4].

Ela possui baixa freqüência de execução e tempo de resposta razoável, ou seja, 90% das transações devem concluir em até cinco segundos, a fim de atender requisições *on line*.

As operações relacionadas nesta transação podem ser observadas no algoritmo 4.

Algorithm 4: Transação *Order-Status*

Input: $homeWarehouse$, valor que indica a qual *Warehouse* o RTE está conectado
Output: $OrderStatusResponse$, objeto de retorno da transação *Order-Status*
Inicia uma transação.
Seleciona um registro da tabela `DISTRICT` com recuperação de dados.
Seleciona um registro da tabela `ORDER` com recuperação de dados.
Seleciona um ou mais registros da tabela `ORDER_LINE` com recuperação de dados.
Finaliza a transação.

3.4.4 Transação *Delivery*

A transação *Delivery* envolve o processamento em lote de dez novas ordens (que não foram entregues ainda). Cada ordem é processada por completo envolvendo operações de leitura e escrita na tabelas do banco de dados.

Ela possui baixa freqüência de execução e tempo de resposta razoável, ou seja, 90% das transações devem concluir em até cinco segundos, a fim de atender requisições *on line*.

Essa transação deve ser executada através de um sistema de agendamento, utilizando uma estrutura de dados como a fila. O resultado da execução deve ser armazenado em um arquivo.

Esta transação possui dois momentos de execução: a parte *on line* e a parte *batch*. Durante a parte *on line*, apenas são gerados os dados que serão utilizados na parte *batch*.

[4]Em 60% dos casos o *Customer* será recuperado através de seu sobrenome. Nesta situação, mais de um *Customer* será recuperado. Seja n o tamanho do conjunto de *Customer*s recuperados, o *Customer* escolhido será o elemento na posição $n/2$ (arredondado para cima). Para o restante dos casos, o *Customer* será recuperado através de seu código. O *Warehouse* utilizado na será o $homeWarehouse$ e o *District* será escolhido aleatoriamente.

Uma vez que os valores são gerados e a transação é agendada, o sistema devolve o resultado para o RTE independente de a transação ter terminado ou não.

Durante a parte *batch*, o sistema retira a transação da fila e a executa. As operações executadas nesta transação podem ser observadas no algoritmo 5.

Algorithm 5: Transação *Delivery*

Input: $homeWarehouse$, valor que indica a qual *Warehouse* o RTE está conectado
Output: $OrderStatusResponse$, objeto de retorno da transação *Order-Status*
Inicia uma transação.
$oCarrierId \leftarrow valor$.
$districts \leftarrow$, Seleciona todos os registros da tabela `DISTRICT` vinculados ao homeWarehouse com recuperacao de dados.
for $district \in districts$ **do**
    Seleciona o registro com menor `order_id` da tabela `NEW_ORDER` com recuperação de dados.
    $numOrder \leftarrow$ `order_id`
    Exclui o registro selecionado na tabela `NEW_ORDER`.
    Seleciona um registro da tabela `ORDER` com recuperação e atualização de dados.
    Seleciona um ou mais registros da tabela `ORDER_LINE` com recuperação e atualização de dados.
    Seleciona um registro da tabela `CUSTOMER` com recuperação e atualização de dados.
end
Finaliza a transação.

3.4.5 Transação *Stock-Level*

A transação *Stock-Level* fornece o número de itens vendidos recentemente e que possuem quantidade em estoque abaixo do limite especificado.

Está transação representa uma operação pesada, ou seja, que demanda grande quantidade de recursos computacionais. Ela tem baixa freqüência de execução e tempo de resposta bastante tolerante, ou seja, 90% das transações devem concluir em até 20 segundos.

As operações executadas nesta transação podem ser observadas no algoritmo 6.

Algorithm 6: Transação *Stock-Level*

Input: $homeWarehouse$, valor que indica a qual *Warehouse* o RTE está conectado; *district*, indica a qual *District* o RTE está conectado
Output: $StockLevelResponse$, objeto de retorno da transação *Stock-Level*
Inicia uma transação.
Seleciona um registro na tabela `DISTRICT` com recuperação de dados.
Seleciona-se um ou mais registros na tabela `ORDER_LINE` com recuperação de dados.
Seleciona-se um ou mais registros na tabela `STOCK` com recuperação de dados.
Finaliza a transação.

3.5 Métricas de desempenho

A especificação do TPC-C define que intervalo mínimo de execução do teste é de 120 minutos, porém ela recomenda que o benchmark seja executado por pelo menos oito horas sem qualquer tipo de interrupção.

Para que o resultado do benchmark seja válido perante o TPC, um conjunto de métricas deve ser apresentado. Os resultados devem ser avaliados por um auditor certificado pelo TPC. O relatório deve conter as informações apresentadas nas seções seguintes:

3.5.1 Distribuição de freqüência dos Tempos de Resposta

A distribuição de freqüência dos Tempos de Resposta (FDRT) deve ser reportada independentemente para cada uma das transações do TPC-C.

Essa métrica mostra a mensuração de todos os FDRTs iniciados e completados durante o intervalo de avaliação. Ou seja, a transação deve ter realizado *commit* com sucesso e o resultado foi apresentado pelo RTE, no caso das transações *New-Order*, *Payment*, *Order-Status* e *Stock-Level*.

Transações que armazenaram o resultado por completo em um arquivo também são consideradas transações que foram realizadas com sucesso.

A figura 3.4 apresenta um exemplo de gráfico de como essa métrica pode ser apresentada.

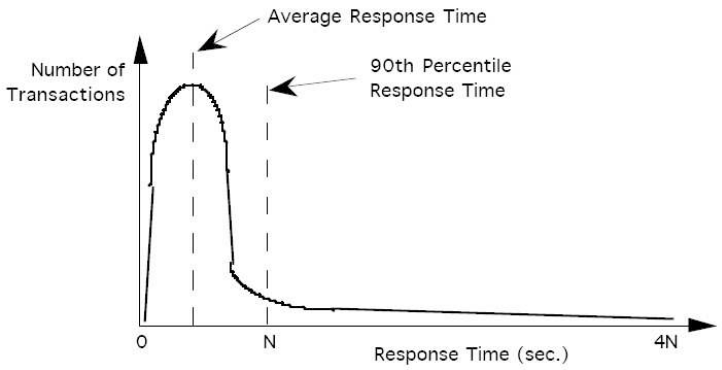

Figura 3.4: Gráfico da FDRT.

3.5.2 Tempo de Resposta vs. a produtividade da transação *New-Order*

Essa métrica deve ser apresentada na forma de gráfico. O eixo x representa a produtividade da transação *New-Order*. O eixo y representa o nonagésimo percentil dos tempos de resposta (RT).

O gráfico deve ser construído de modo que a produtividade possa ser visualizada nos intervalos de 50%, 80% e 100%. Um exemplo desse gráfico pode ser visualizado na figura 3.5.

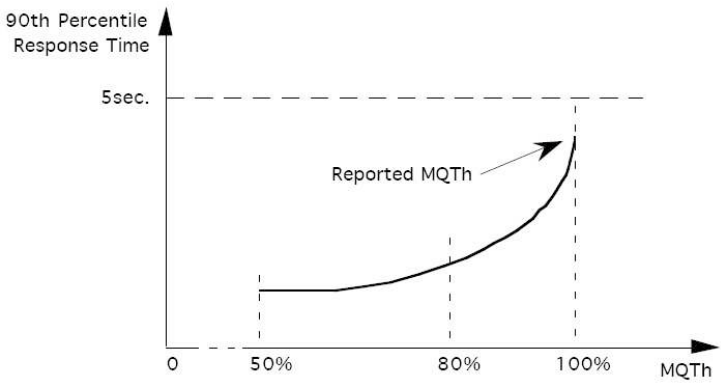

Figura 3.5: Gráfico do RT vs. a produtividade da transação *New-Order*.

3.5.3 Distribuição de freqüência dos *Think Times* (FDTT)

Como pode ser visto na figura 3.3, cada RTE executa um ciclo composto de passos. Entre estes passos estão a apresentação de tela, os tempos de digitação e espera (*Think Times*).

Os *Think Times* representam o tempo que o RTE (considerando um usuário real) leva para observar o resultado das transações na tela.

Um exemplo desse gráfico pode ser visualizado na figura 3.6.

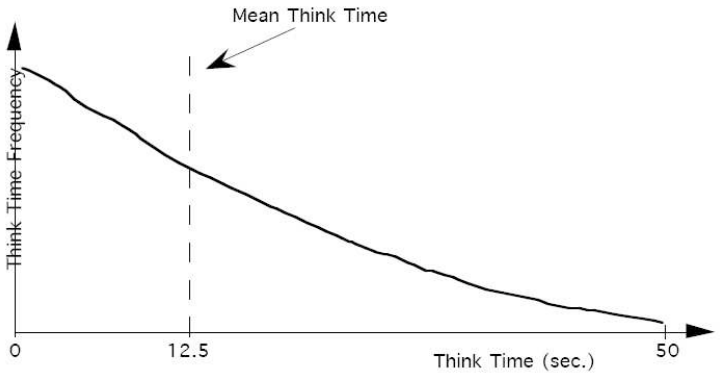

Figura 3.6: Gráfico do FDTT.

3.5.4 Produtividade da transação *New-Order* versus o tempo decorrido

Essa métrica também deve ser apresentada em forma de gráfico, o qual deve mostrar tanto o tempo de aceleração como o tempo total do benchmark. O eixo x representa o tempo decorrido desde o início do benchmark. O eixo y representa a produtividade em tpmC.

Um mínimo de 240 intervalos deve ser utilizado, cada um com um intervalo máximo de 30 segundos. A figura 3.7 apresenta um exemplo de como essa métrica pode ser apresentada.

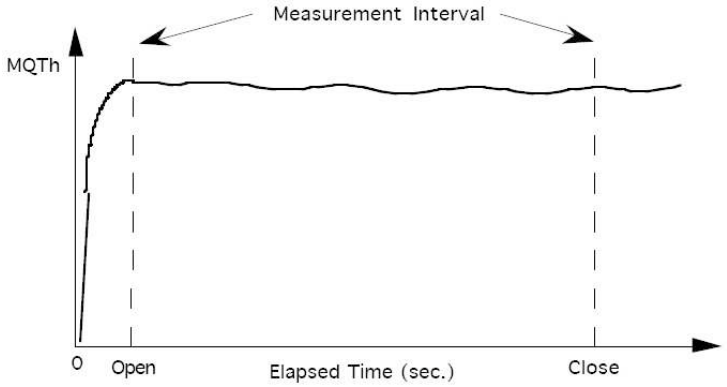

Figura 3.7: tpmC vs. tempo decorrido.

CAPÍTULO 4

FERRAMENTAS PARA EXECUÇÃO DE BENCHMARKS

A maioria das ferramentas desenvolvidas para a realização de benchmarks não é disponibilizada para a comunidade. Especialmente para os testes do TPC-C, em que o responsável pela construção do benchmark é o próprio interessado na divulgação dos resultados, que nesse caso são grandes fornecedores de SGBDs comerciais e empresas da área de *hardware*.

Esse comportamento motivou grupos de desenvolvedores e pesquisadores a construir versões de código aberto para a especificação do TPC-C.

Nas subseções seguintes, apresentaremos algumas das ferramentas mais importantes para a realização de benchmarks.

4.1 OSDL-DBT2

O *Open Source Development Labs* (OSDL) foi uma organização sem fins lucrativos cujos principais objetivos incluíam a melhoria e o aumento da velocidade na adoção do sistema operacional Linux [7].

O OSDL desenvolveu uma série de ferramentas baseadas nas especificações do TPC, entre elas o DBT2 (que era a implementação do TPC-C) e o DBT3, que era baseado no TPC-H, especificação para sistemas de apoio a tomada de decisão.

Em 2007 o OSDL se uniu a outro grupo (*Free Standards Group*) e formaram a *The Linux Foundation* cujo objetivo passou a ser o aumento da competição do sistema operacional Linux em relação ao Microsoft Windows.

4.2 TPCC-UVa

Desenvolvido na Universidade de Valladolid, o TPCC-UVa [17] foi codificado em linguagem de programação C e utiliza o núcleo do PostgreSQL [13] como base de dados.

Entre os pontos fortes do TPCC-UVa, podemos citar, por exemplo, a fidelidade que ele mantém em relação aos requisitos definidos na especificação do TPC-C. A única não conformidade que ele apresenta, é a não utilização de um monitor de transação (TM) comercial. Na verdade, eles optaram por desenvolver seu próprio TM.

Esta ferramenta possui algumas restrições quanto ao seu uso em avaliações de SGBD. A maneira como ela foi projetada a tornou completamente amarrada ao sistema operacional Linux e ao SGBD PostgreSQL. Além disso, não é possível replicá-la em diversos computadores de uma rede local e executar diversas cópias dela em máquinas distintas.

Independente de todas essas questões, a utilização de um TM parece ser o fator de maior impacto na simulação de um ambiente multi-usuário. O TM recebe as transações submetidas pelos usuários remotos e as enfilera. Um processo contínuo remove essas transações da fila e as executa. Esta abordagem conduz a simulação de volta a um ambiente mono-usuário e assim não existirá concorrência dentro do SGBD.

4.3 BenchmarkSQL

No maior sitio de desenvolvimento de sistemas de código aberto da internet, uma das ferramentas relativas à implementação do TPC-C mais bem classificada é o BenchmarkSQL [18].

O BenchmarkSQL é uma ferramenta desenvolvida em linguagem de programação Java e que reflete bastante a especificação do TPC-C. Essa ferramenta utiliza *drivers* JDBC para se comunicar com os SGBDs. Isto a torna compatível com uma ampla lista de sistemas de bancos de dados [26].

A interface com o usuário é uma janela similar às encontradas em sistemas Windows, a partir da qual as funcionalidades da ferramenta são executadas através de cliques em botões.

O BenchmarkSQL prove quase todas as características que propusemos para o TPCC-C3SL, embora ela apresente algumas questões que acreditamos que podem ser melhoradas.

Uma dessas questões é o carregamento inicial dos dados. O BenchmarkSQL realiza essa tarefa primeiramente criando as tabelas através de comandos SQL. Após isso, ela

carrega os dados iniciais com base no número de *Warehouses* configurados. Por fim, outro conjunto de comandos SQL deve ser executado a fim de que os índices das tabelas sejam criados. Essa estratégia influência o tempo de carregamento inicial dos dados, uma vez que a existência de índices nas tabelas reduz o desempenho de operações de inserção.

Outra questão é a simulação de ambiente multi-usuário. A maneira padrão de utilizar a ferramenta, é executar todos os RTEs em um mesmo processador, utilizando dessa forma uma abordagem *multi-thread*. É possível espalhar cópias do programa em outras máquinas de uma mesma rede local e executar o benchmark para todas essas cópias. O ponto negativo é que os usuários precisam executar a ação a partir da interface gráfica. Dessa forma, é muito improvável que o usuário consiga ir a cada uma das cópias do programa e executar a operação sem afetar o tempo de inicio entre cada aplicativo.

4.4 OSDB

Outra ferramenta de *benchmark* que pode ser encontrada na internet é o *Open Source Database Benchmark* (OSDB) [3]. Os seus desenvolvedores resolveram não seguir a especificação definida pelo TPC-C e construíram uma ferramenta de benchmark baseada no AS^{3}AP.

Além disso, a última atualização disponibilizada para esta ferramenta aconteceu em Outubro de 2004.

4.5 Comparativo entre as ferramentas de benchmark

A tabela 4.1 apresenta um resumo das características das ferramentas analisadas e as compara com a aplicação proposta por nós, o TPCC-C3SL.

As principais diferenças que podemos observar nas ferramentas analisadas são: (i) a forma com que os benchmarks simulando ambientes multi-usuários são realizados e (ii) e o alto acoplamento com determinado(s) SGBDS.

Identificamos nessas deficiências a possibilidade de construção de uma ferramenta superior. O resultado disso é o TPCC-C3SL, nossa proposta de ferramenta para execução

Características	**TPCC-C3SL**	**TPCC-UVa**	**OSDB**	**BenchmarkSQL**
Metodologia	TPC-C	TPC-C	AS^{3}AP	TPC-C
Linguagem[a]	Java	C	C	Java
SO[b]	Independente[c]	GNU/Linux	GNU/Linux	Independente
Simulação[d]	Sim	Sim	Sim	Sim
Testes distribuídos[e]	Sim	Não	Não	Não
Criação das tabelas[f]	Automática	script SQL	script SQL	script SQL
SGBDs compatíveis	Qualquer um que forneça um *driver JDBC*[g]	PostgreSQL	MySQL, PostgreSQL e Informix	Vários[h]

[a]Linguagem de programação utilizada

[b]Sistema operacional compatível

[c]A linguagem de programação Java é tanto compilada como interpretada, assim uma aplicação Java já compilada pode ser executada em qualquer sistema operacional que possua uma máquina virtual Java (JVM)

[d]Simulação de ambiente multi-usuário

[e]Múltiplas instâncias do *benchmarks* executando em diferentes computadores

[f]Forma de criação das tabelas do banco de dados

[g]No momento da escrita deste trabalho, existiam mais de 100 SGBDs compatíveis (por exemplo, Oracle, Sybase, IBM DB2 e Microsoft SQL Server) e 29 dialetos disponíveis para o *framework* Hibernate

[h]Uma ampla lista, incluindo os maiores SGBDs como Oracle, Sybase e Microsoft SQl Server

Tabela 4.1: Análise das características das ferramentas de *benchmark*.

de benchmarks baseados na especificação do TPC-C. O TPCC-C3SL é apresentado com maiores detalhes no capítulo 5.

CAPÍTULO 5

TPCC-C3SL

Neste capítulo descrevemos as principais características de nossa implementação para a especificação do TPC-C. Apesar de procurarmos seguir fielmente as diretivas definidas na especificação, nossa ferramenta não está em conformidade com seus requerimentos pelo fato de que não utilizamos um monitor de transação (TM) comercial.

Entre as funcionalidades que competem ao TM está o uso de uma estrutura de dados baseada nos conceitos de fila, na qual as transações a serem disparadas contra o SGBD são inseridas. Ele também pode ordenar as transações nessa fila caso seja necessário.

Na arquitetura do TPCC-C3SL não existe o conceito do TM. As transações são submetidas contra o SGBD diretamente pelos RTEs .

5.1 Independência de banco de dados

Durante o planejamento da arquitetura do TPCC-C3SL, um dos principais objetivos imaginados para o TPCC-C3SL foi que a ferramenta fosse capaz de realizar benchmarks em uma ampla lista de SGBDS.

Também era primordial que nenhuma modificação deveria ser realizada no código fonte quando o SGBD sob avaliação fosse substituído por outro sistema de banco de dados.

Para realizar essa tarefa optamos por utilizar o *framework Hibernate*.

O *Hibernate* é uma implementação para a *Java Persistence API* (JPA) [25], a qual faz parte da especificação *Enterprise Java Beans 3.0* [16]. O objetivo da especificação JPA é definir um padrão para as operações de armazenamento, recuperação, atualização e exclusão (CRUD) de objetos Java para SGBDs relacionais.

A partir da versão 5, a linguagem de programação Java oferece uma característica chamada *Annotations*. Uma *Annotation* é um tipo de instrução de programação que pode ser utilizada em conjunto com a JPA para orientar o SGBD em determinadas operações.

```
package br.ufpr.tpccufpr.vo;

@Entity
@Table(name = "WAREHOUSE")

public class Warehouse implements Serializable {

   private Long id;

   @Id
   @Column(name = "W_ID")
   public Long getId() {
      return id;
   }

   public void setId(Long id) {
      this.id = id;
   }
}
```

Figura 5.1: Classe Java com *Annotations* JPA.

A *Annotation* é codificada no nível de uma classe Java. A figura 5.1 ilustra uma classe Java com Annotations. A *Annotation* `@Entity` cria um vínculo entre o objeto Java e uma tabela do banco de dados. A *Annotation* `@Table` estipula que o nome desta tabela vinculada. A classe Java ilustrada na figura 5.1, por exemplo, estará vinculada a tabela `WAREHOUSE`. Toda as operações CRUD executadas em objeto Java *Warehouse* serão executadas diretamente na tabela `WAREHOUSE`.

Caso uma *Annotation* `@Table` não seja fornecida, o Hibernate utilizará o mesmo nome da própria classe Java.

As *Annotations* localizadas logo acima do método `getId()` se referem ao atributo `id`. `@Column` indica que uma coluna será criada nessa tabela e o nome para essa coluna será `W_ID`. A *Annotation* `@Id` determina que esta coluna seja uma chave primária (PK).

Através das *Annotations*, a aplicação é capaz de executar comandos do tipo *Data Definition Language* (DDL) contra o SGBD sem utilizar qualquer instrução explícita de *Structured Query Language* (SQL).

O provedor da implementação da JPA, no caso o Hibernate, fornece dialetos de comunicação com uma ampla lista de SGBDs. Os dialetos traduzem as operações sobre os objetos Java em comandos SQL que são disparados contra o SGBD.

Por exemplo, no início da execução do benchmark, o TPCC-C3SL identificará que a classe *Warehouse* ilustrada na figura 5.1 possui uma *Annotation* `@Entity`. Isso significa que uma tabela `WAREHOUSE` deve existir no banco de dados. Caso a tabela não exista a aplicação criará essa tabela automaticamente.

A figura 5.2 ilustra como o Hibernate traduz as *Annotations* encontradas na classe Java *Warehouse* 5.1 em um comando SQL `CREATE TABLE`.

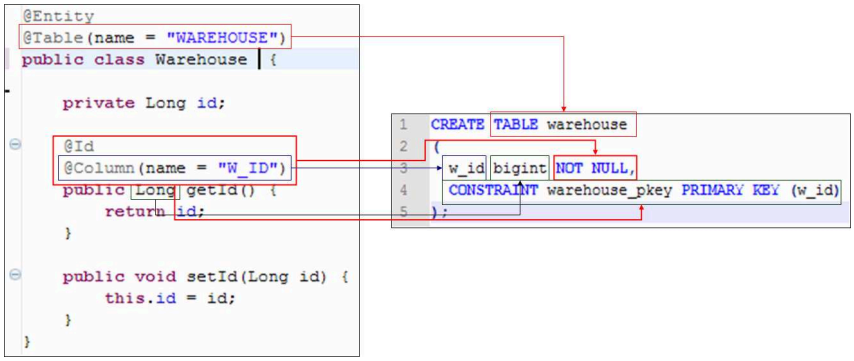

Figura 5.2: Conversão de uma classe Java com *Annotations* em um comando SQL `CREATE TABLE`.

O grande benefício da utilização das *Annotations* em conjunto com a JPA é que a aplicação praticamente não tem acoplamento com o SGBD, dessa forma uma vez que seja necessário mudar a camada de persistência de um SGBD `X` para `Y`, não existe necessidade de modificar o código da aplicação para este novo SGBD. A única modificação necessária é no arquivo de propriedades da conexão entre o aplicativo e o banco de dados.

5.2 Arquitetura

Como uma implementação da especificação do TPC-C, a principal funcionalidade de nossa aplicação é a execução das cinco transações definidas na especificação.

O TPC-C determina, através de um valor fixo, a freqüência com que cada transação deve ser executada. Como funcionalidade adicional, inserimos no sistema a possibilidade de se alterar esses valores. Esta abordagem proporciona ao usuário a capacidade de alterar a freqüência de distribuição das transações de maneira que ele possa adaptar o comportamento do *benchmark* às suas necessidades. Entretanto, reforçamos que a

modificação da freqüência de distribuição das transações viola a especificação do TPC-C.

O componente mais importante para a execução do *benchmark* é o *Terminal Remoto Emulado* (RTE), o qual representa um usuário do sistema submetendo transações contra o SGBD. Uma vez que o RTE inicia sua execução, ele entra em um ciclo até que outro componente do sistema o instrua a encerrar sua execução.

Dentro do ciclo, o RTE executa uma série de passos. Entre estes passos está a seleção aleatória de uma transação. A figura 5.3 apresenta o diagrama de colaboração em linguagem de modelagem unificada (UML) entre os componentes envolvidos no ciclo.

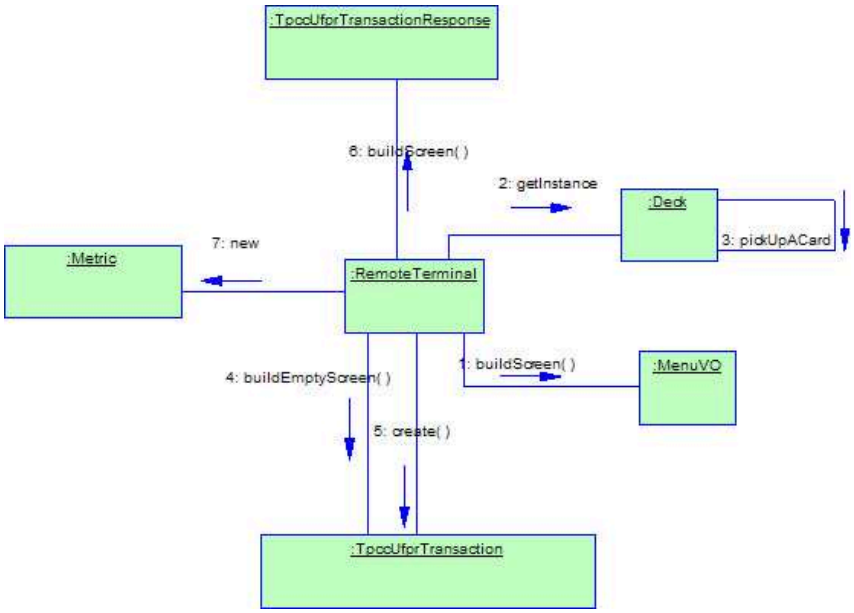

Figura 5.3: Diagrama de colaboração dos componentes envolvidos no ciclo de execução do RTE.

Em nosso projeto, codificamos uma espécie de "repositório" das cinco transações que o RTE deve executar. No contexto da aplicação o componente responsável por gerenciar o repositório é chamado de *Deck*, o qual carrega o repositório no início da execução do *benchmark*.

A distribuição de freqüência das transações no repositório reflete a configuração determinada pelo usuário.

No passo em que o RTE escolhe qual transação deve executar, ele solicita para o *Deck* um objeto que representa a transação a ser executada. O *Deck* por sua vez, remove de maneira aleatória uma transação de seu repositório e a envia para o RTE. Se no mo-

mento da remoção do transação o repositório estiver vazio, o *Deck* carregará o repositório automaticamente.

A transação selecionada pelo *Deck* é enviada para o RTE como uma interface (objeto que encapsula a codificação de fato). A figura 5.4 apresenta o diagrama de classes dos objetos envolvidos na seleção das transações a serem executadas.

Por exemplo, considere que o RTE solicitou ao *Deck* uma transação. O *Deck* selecionou aleatóriamente uma transação *Payment* do repositório. A transação *Paymente* está codificado no objeto Java *PaymentTransaction*. Quando o *Deck* envia a transação para o RTE, está será enviada como uma *TpccufprTransaction*, que é a interface do *PaymentTransaction*.

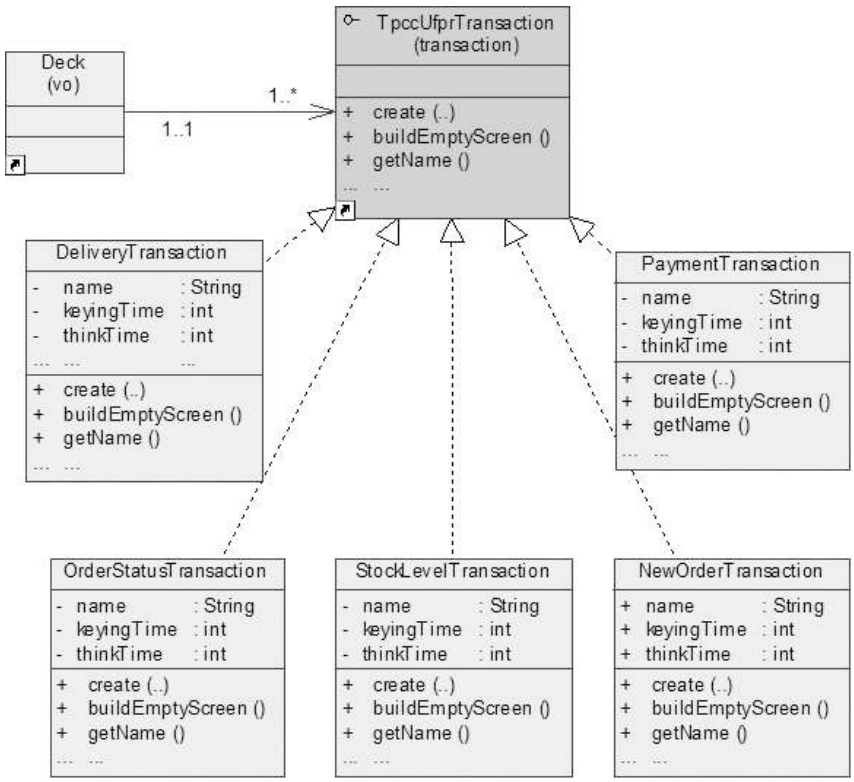

Figura 5.4: Relação entre os objetos *Deck* e as transações.

Logo após receber o objeto que representa a transação, o RTE a executa. Apesar do fato de que ele não sabe qual transação de fato recebeu, quando ele invoca o método `create()`, o método que é realmente invocado é o da classe com a implementação real. Isto é possível em função de um conceito existente nas linguagens orientadas a objetos,

chamado Polimorfismo.[1]

Durante a execução da transação, vários comandos são submetidos contra o SGBD. Os tempos de execução são armazenados em memória e o resultado da transação como um todo é verificado. Um resultado `INCOMPLETO` significa que ocorreu algum problema durante a execução da transação e um comando *rollback* foi executado.

O último passo executado dentro do ciclo do RTE é o armazenamento dos valores que serão utilizados na construção do relatório de desempenho do *benchmark*, os quais envolvem basicamente os tempos de respostas e o resultado geral da transação.

5.3 Métricas

Os tempos de resposta e estado geral da transação compõe um objeto Java chamado *Metric*, o qual também é armazenado no SGBD em avaliação.

Pelo fato de utilizamos drivers JDBC como meio de comunicação entre o SGBD e o TPCC-C3SL, o banco de dados em que as métricas serão armazenadas pode estar em uma máquina diferente da máquina que executará o benchmark. Isso proporciona ao TPCC-C3SL a capacidade de simular tanto ambientes mono-usuário como ambientes multi-usuário.

Para simular um ambiente mono-usuário, é necessário configurar o número de RTEs para 1. Isso inicializará apenas um RTE que se comportará como um usuário real submetendo transações contra o SGBD.

A simulação de um ambiente multi-usuário pode ser executada de três diferentes maneiras: (i) pode-se configurar o sistema para executar mais de um RTE na mesma máquina, (ii) executar cada RTE em uma máquina diferente ou (iii) mesmo uma combinação dessas duas opções. Isso significa que cada máquina selecionada para executar o *benchmark* terá mais de um RTE em execução. Estes cenários serão descritos mais detalhadamento no capítulo 6.

[1]Segundo [15], com o polimorfismo dois ou mais objetos podem responder à mesma mensagem. Isso significa que duas classes, `c1` e `c2`, por exemplo, podem ter um método `getName()`. A assinatura do método é a mesma para ambas as classes, porém a implementação deles pode ser completamente diferente.

CAPÍTULO 6

ANÁLISE DOS TESTES

Neste capítulo descreveremos a metodologia utilizada bem como os resultados dos testes realizados com o TPCC-C3SL.. Primeiramente, descreveremos a arquitetura de rede e máquinas utilizadas. Após isso apresentamos as configurações e parâmetros usados no benchmark e por fim descrevemos as características de cada teste e os resultados obtidos.

Para avaliar o TPCC-C3SL utilizamos três SGBDs diferentes. Os SGBDs utilizados incluem produtos de código aberto como PostgreSQL [13] e MySQL [19]. Também realizamos os testes em um SGBD desenvolvido em linguagem Java, o Apache Derby [8].

6.1 Ambiente de testes

Para executar os testes, utilizamos um servidor Intel Quad Core com dois discos rígidos SCSI de 250GB (configurados em RAID 0) e um total de 2GB de memória RAM. O sistema operacional (SO) do servidor utilizado foi o Ubuntu 8.04 LTS Server Edition para processadores com arquitetura de 64 bits. Deste ponto em diante, nos referiremos a esta máquina como `ribeira`.

Durante os testes de ambiente multi-usuário, utilizamos outras cinco máquinas com configurações diversas. Essas máquinas serão tratadas a partir deste ponto como máquinas-cliente. Tanto a máquina `ribeira` como as máquinas cliente estavam dentro da mesma rede local. A tabela 6.1 apresenta as características das máquinas utilizadas.

A máquina escolhida para executar os SGBDs foi a `ribeira`. Os SGBDs instalados foram:

- PostgreSQL 8.3.1
- MySQL 5

Nome	Processador	Memória (em GB)	HD (em TB)[a]	SO
`ribeira`	Intel(R) Xeon(TM) Quad-Core (2 GHz)	2	0.5	Ubuntu 8.04 LTS Server Edition (Linux version 2.6.24-16-server)
`macalan`	Dual-Core AMD Opteron(tm) Processor 2220 (2800 MHz)	20	5	Debian 4.1.1-21 (Linux version 2.6.24.7)
`caporal`	AMD Opteron(tm) Processor 242 (1600 MHz)	4	5	Debian 4.1.1-21 (Linux version 2.6.24.7)
`talisker`	AMD Opteron(tm) Processor 242 (1600 MHz)	4	5	Debian 4.1.1-21 (Linux version 2.6.24.7)
`cohiba`	Intel(R) Xeon(TM) 3.60GHz	4	5	Debian 4.1.1-21 (Linux version 2.6.24.7)
`bolwmore`	Dual-Core AMD Opteron(tm) Processor 2220 (2800 MHz)	32	5	Debian 4.1.1-21 (Linux version 2.6.24.7)

[a]Com exceção da `ribeira`, todas as outras máquinas compartilhavam o mesmo disco.

Tabela 6.1: Configuração das máquinas utilizadas

O SGBD Derby não precisa ser instalado. Pelo fato de ser desenvolvido inteiramente em Java, basta apenas descompactar o arquivo e iniciar o servidor do Derby.

É importante ressaltar que nenhum ajuste com intenção de melhorar o desempenho do SGBD foi realizado. Cada SGBD foi instalado utilizando sua configuração padrão. Da mesma forma, também não foi utilizado nenhuma características do *framework* Hibernate que pudessem proporcionar um ganho de desempenho, como os *caches* de segundo nível.

Outro aplicativo que foi instalado em todas as máquinas é o ambiente de execução Java (JRE), mais conhecido como Máquina Virtual Java (JVM). A versão da JVM utilizada foi a versão 6.

A JVM permite que alguns parâmetros lhe sejam fornecidos de modo que as aplicações em execução nela obtenham melhor desempenho. A tabela 6.2 apresenta a lista dos argumentos utilizados, seus respectivos valores e descrições.

Com todos os SGBDs necessários e a correta versão da JVM instalados, uma vez que seja necessário mudar o SGBD sob avaliação, é necessário modificar apenas alguns parâmetros no arquivo de configuração `tpccufpr_conn.properties`. A figura 6.1 ilustra quais parâmetros devem ser modificados.

Pode-se notar que a figura 6.1 ilustra as propriedades de conexão para os SGBDs

Argumento	Valor	Descrição
`-XX:+UseParNewGC`		Auxilia aplicativos que instanciam massivamente novos objetos Java.
`-Xms<size>`	1GB[a]	Define o tamanho inicial da memória Heap utilizado pelo Java.
`-Xmx<size>`	1GB[b]	Define o tamanho máximo da memória Heap.
`-Xmn<size>`	100MB	Tamanho da área de memória *Young*[c].

[a]Para as máquinas clientes foi utilizado 256MB

[b]Para as máquinas clientes foi utilizado 256MB

[c]Basicamente, a memória Heap utilizada pela JVM é divida em duas áreas: *Young* e *Old*. Todos os objetos Java quando instanciados ocupam a área Young. Quando essa área se esgota, um processo chamado *Garbage Collection* (GC) é executado. O objetivo do GC é liberar o espaço utilizado pelos objetos "mortos" (objetos que não precisam mais existir durante o contexto da aplicação.) Durante GC na área Young, os objetos que ainda estão "vivos" são movidos para a área Old. A documentação da JVM diz que para as aplicações obterem melhor desempenho, a maioria dos objetos na memória Heap deveriam ser removidos enquanto eles estão na área Young, uma vez que o GC consume menos recursos quando ele executa nesta área.

Tabela 6.2: Argumentos da JVM

```
# Hibernate connection configuration
# Modify these properties in order to match your database configuration

# PostgreSQL
hibernate.connection.driver_class=org.postgresql.Driver
hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
hibernate.connection.url=jdbc:postgresql:/\ /localhost/tpccufpr
hibernate.connection.username=tpccufpr
hibernate.connection.password=tpccufpr

# MySQL 5
#hibernate.connection.driver_class=com.mysql.jdbc.Driver
#hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
#hibernate.connection.url=jdbc:mysql:/\ /localhost/tpccufpr
#hibernate.connection.username=root
#hibernate.connection.password=1234
```

Figura 6.1: Arquivo de conexão do TPCC-C3SL

PostgreSQL e MySQL. O caracter # significa que tudo que vier após ele é um comentário e não será considerado pelo programa. Neste exemplo, o TPCC-C3SL considerará apenas a linhas relativas à conexão com o PostgreSQL. Para mudar a conexão para o MySQL, basta comentar as linhas referentes à conexão com o PostgreSQL e descomentar a linhas referentes ao MySQL.

Basicamente, cinco parâmetros devem ser alterados para modificar a conexão com o SGBD.[1]

1. ***Driver class*** Classe Java responsável por gerenciar a conexão com o SGBD. Essa clase também é responsável por manter o nível de isolamento selecionado.

2. ***Hibernate dialect*** Responsável por traduzir as operações JPA entre a aplicação e o SGBD.

3. ***Connection URL*** Indica para a classe Java como se conectar com o SGBD.

4. ***Username*** *login* de um usuário existente no SGBD.

5. ***Password*** Senha do usuário.

Os drivers JDBC são desenvolvidos pelos próprios fornecedores dos SGBDs. Apesar disso, as propriedades de conexão são bastante similares. As diferenças ocorrem principalmente nas propriedades *Connection URL* e *Hibernate Dialect*. Alguns SGBDs, por exemplo o Apache Derby, não utilizam por padrão uma senha para conexão com o SGBD. Nesse caso, é necessário comentar a linha inteira da propriedade *Password* ao invés de fornecer uma senha em branco.

6.2 Especificação geral do benchmark

Para executar a simulação, um conjunto mínimo de dados deve ser carregado nas tabelas. A tabela 3.1 apresenta o número de tuplas que cada tabela deve conter caso a tabela `WAREHOUSE` seja carregada com um registro.

[1] Verifique a documentação do *driver* JDBC que se pretende utilizar.

A especificação do TPC-C define ainda a maneira como cada tabela será carregada. Por exemplo, o sobrenome de cada *Customer* inserido na tabela `CUSTOMER` deve ser formado a partir da junção de três palavras. A tabela 6.3 apresenta o conjunto de palavras disponíveis para os sobrenomes. Para formar o sobrenome, deve-se gerar um número aleatório entre 0 e 999. Cada caracter individual do número gerado representa uma palavra da tabela 6.3. Por exemplo, o número 371 resulta no sobrenome PRICALLY-OUGHT.

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
BAR	OUGHT	ABLE	PRI	PRESS	ESE	ANTI	CALLY	ATION	EING

Tabela 6.3: Palavras disponíveis para a formação do sobrenome do *Customer*.

A quantidade de registros em cada tabela varia em função do número de `WAREHOUSES` definidos. Para facilitar o entendimento, utilizaremos partir desse ponto, a expressão fator de escala (FS), como o número de `WAREHOUSES` configurados. A tabela 6.4 apresenta a quantidade de registros que foram inseridos nas tabelas para os testes realizados. A exceção se dá no teste que avalia o desempenho dos SGBDs no carregamento das tabelas em função do FS configurado. Nesse teste, a cada execução do benchmark o FS era incrementado em 1.

Tabela	**Registros**	**Tamanho da tabela (em 1K bytes)**
Warehouse	1	0,089
Item	100K	8.200
Stock	100K	30.600
District	10	0.950
Customer	30K	19.650
History	30K	1.380
Order	30K	720
Order-Line[a]	300K	16.200
New-Order	9K	72

[a] 1% de variação é permitido em função da variação encontrada durante o carregamento dos dados iniciais.

Tabela 6.4: Cardinalidade inicial para a execução dos testes.

Para realizar o carregamento inicial dos dados, utilizamos o TPCC-C3SL. Mesmo que as tabelas não existam no banco de dados, no início do carregamento dos dados, a aplicação

irá verificar se as tabelas necessárias existem ou não. Se as tabelas já existirem no banco de dados, o TPCC-C3SL primeiramente as removerá do SGBD e logo em seqüência criará as tabelas novamente. Após isso começa o carregamento dos dados.

Caso as tabelas não existam no banco de dados, o TPCC-C3SL simplesmente cria as tabelas no SGBD e inicia o carregamento dos dados.

A razão pela qual optamos por remover as tabelas do SGBD em vez de apagar seus conteúdos, se dá pelo fato que de recriar a tabela (através dos comandos `DROP` e `CREATE`) consome consideravelmente menos tempo e recursos do que um comando `delete from <table_name>`[2].

A maioria das tabelas é carregada em paralelo com outras tabelas. Por exemplo, as tabelas `WAREHOUSE` e `DISTRICT`. Porém, outras tabelas são carregadas de maneira isolada, como é o caso da tabela `ITEM`.

Além do tamanho inicial do banco de dados (baseado no FS), deve ser definida também a distribuição de freqüência (FD) das transações do TPC-C. A especificação do TPC-C determina que para cada dez transações *New-Order* executadas, devem ser executadas dez transações *Payment*, uma transação *Order-Status*, uma transação *Delivery* e uma transação *Stock-Level*.

A figura 6.2 ilustra a parte do arquivo de configuração `tpccufpr.properties` onde é possível ajustar a FD das transações que serão executadas durante o benchmark. A soma dos pesos deve sempre resultar no valor vinte e três (23). Caso seja necessário modificar o comportamento do benchmark para que execute apenas transações *New-Order*, basta configurar a propriedade `tpcc.ufpr.txn.weight.new.order` para vinte e três (23) e as outras quatro propriedades para zero (0).

Outros dois parâmetros que devem ser ajustados para a realização do benchmark são `tpcc.ufpr.bechmark.time` e `tpcc.ufpr.rampup.time`. O primeiro parâmetro define o tempo de duração do benchmark e o segundo parâmetro define o tempo de *aceleração*.

O tempo de aceleração representa um intervalo no qual os RTEs se conectam ao

[2]Cada comando `delete` executado gera um log. Um instrução `delete from <table_name>` obriga o SGBD a gerar uma log para cada linhada tabela. Comandos DDL como `CREATE` e `DROP` não geram log e por isso exigem menos recursos.

```
# defines the amount that each transaction will be executed.
# The sum of weights must match 23 or the applications will
# throws an exception and quit
tpcc.ufpr.txn.weight.new.order=10
tpcc.ufpr.txn.weight.payment=10
tpcc.ufpr.txn.weight.order.status=1
tpcc.ufpr.txn.weight.delivery=1
tpcc.ufpr.txn.weight.stock.level=1
```

Figura 6.2: Configuração da distribuição de freqüência das transações no TPCC-C3SL.

sistema. Esse intervalo garante uma margem de segurança até que todos os RTEs tenham se conectado e o benchmark estre num estado "estável" até que as métricas começem a ser armazenadas.

Considerando que o tempo total do *benchmark* foi configurado para trinta minutos e o tempo de aceleração para cinco minutos, significa que durante os cinco primeiros minutos, não será armazenado o resultado das transações executadas. Portanto, as transações executados dentro do tempo de aceleração não são contabilizadas no relatório de desempenho do benchmark.

6.3 Avaliação dos resultados

6.3.1 Carregamento inicial

Neste teste preliminar o objetivo foi verificar como cada SGBD se comportava em uma situação de inserção massiva de dados. É importante ressaltar que neste cenário não existe concorrência no SGBD. A máquina utilizada foi a `ribeira` e o FS foi um (1). O resultado do pode ser observado na figura 6.3

Com base no resultado apresentado na figura 6.3 podemos observar que tanto o PostgreSQL como MySQL obtiveram desempenho semelhante. Considerando o tempo total de carregamento, o PostgreSQL foi um pouco mais lento que o MySQL, concluindo o carregamento inicial com um tempo 13% maior que o tempo consumido pelo MySQL.

O SGBD com o pior desempenho foi o Apache Derby, finalizando o carregamento com um tempo total 33% maior que o tempo total do SGBD mais rápido (MySQL).

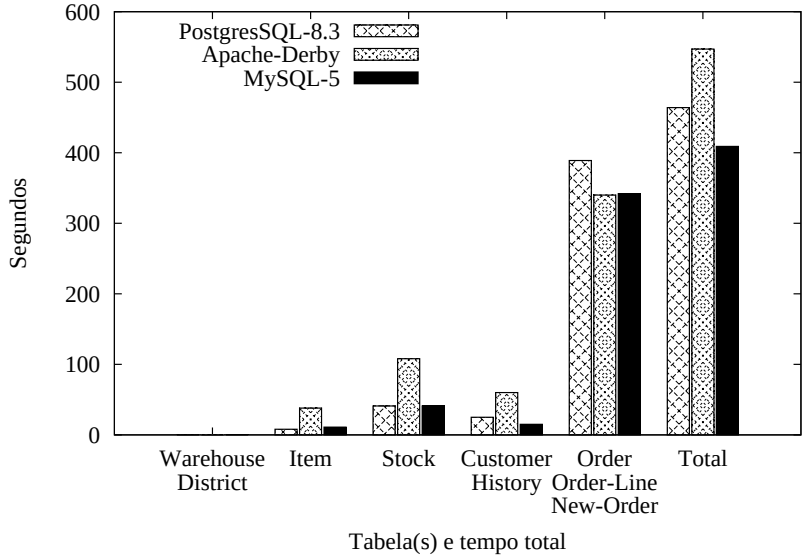

Figura 6.3: Tempo de carregamento dos dados iniciais.

Observando ainda o comportamento dos SGBDs nas operações de inserção de registros, executamos dez testes de carregamento para cada SGBD. A cada teste o fator de escala foi elevado em ponto. Isso que dizer que ao final do teste, cada SGBD foi carregando com um número aproximado de seis milhões de registros. A figura 6.4 ilustra o desempenho de cada um dos SGBDs.

Podemos notar pelos resultados apresentados na figura 6.4 que tanto o PostgreSQL como o MySQL mantiveram um tempo médio de carregamento por *Warehouse* independentemente do FS utilizado. Já o Apache Derby apresentou perda de desempenho cada vez que o FS foi incrementado.

6.3.2 Teste de desempenho - modo *server*

Nos testes de desempenho utilizamos a principal métrica definida pelo TPC-C: tpmC. Como já foi mencionada anteriormente, essa métrica considera o número de transações *New-Order* executadas com sucesso durante o benchmark. Este número é divido pelo tempo de execução (em minutos) do benchmark resultando no número de transações *New-Order* processadas por minuto.

O objetivo desse teste era observar o comportamento dos SGBDs conforme a aplicação

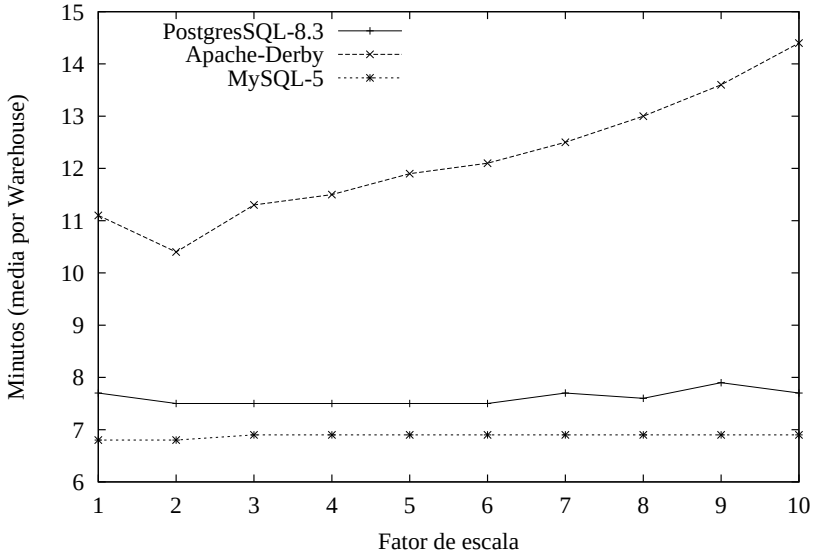

Figura 6.4: Tempo médio de carregamento por *Warehouse* em função de FS.

modifica sua característica.

A especificação do TPC-C define que 43% de todas das transações executadas devem ser de transações *New-Order*. Neste teste, para cada SGBD avaliado, executamos o benchmark oito vezes. A primeira execução utilizou a FD padrão, ou seja, 43% de transações executadas eram transações *New-Order*. A cada nova execução do benchmark, aumentamos a participação da transação *New-Order* sobre o total de transações executadas em nove pontos percentuais. Dessa forma, a cada nova execução aumentamos o nível de concorrência no SGBD pelo fato de que a transação *New-Order*, além das várias inserções que realiza, ela também necessita de acesso exclusivo em um determinado registro da tabela `DISTRICT`.

Este teste foi executado na mesma máquina em que os SGBDs estavam instalados (`ribeira`), assim não existe sobrecarga de transmissão de dados pela rede. Definimos um total de dez RTEs que executaram por dez minutos. O tempo de aceleração foi estipulado em um minuto. O resultado do teste pode ser observado no gráfico da figura 6.5

Como a métrica tpmC é baseada somente na transação *New-Order*, é natural que conforme o número de transações *New-Order* executadas cresce em relação ao total de transações, o valor de tpmC também aumente.

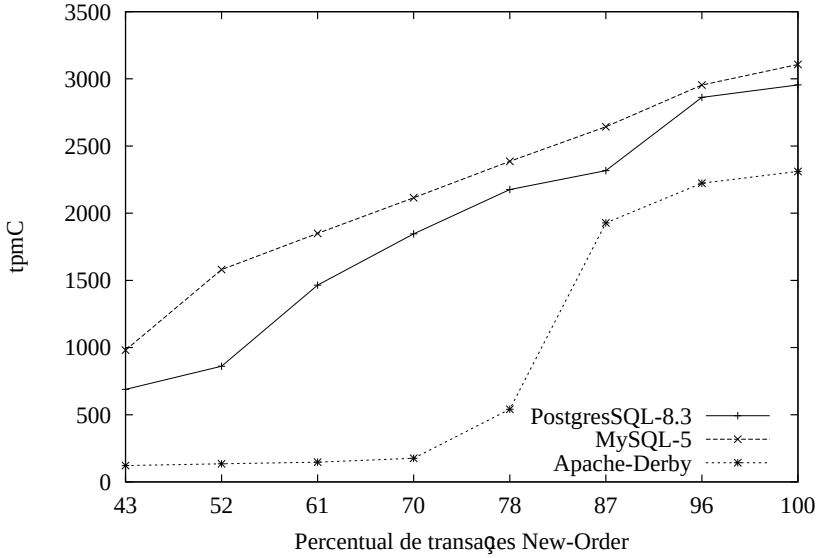

Figura 6.5: Resultado do teste *server*.

Pode-se observar pelo resultado do teste que, novamente, o PostgreSQL e o MySQL apresentaram comportamentos bastante similares. Suas curvas de desempenho mostram um padrão de crescimento conforme cresçe o percentual de transações *New-Order*.

A curva de desempenho do Apache Derby propõe uma suposição: o SGBD apresenta bom desempenho e começa a se equiparar em relação aos outros SGBDs, a partir do momento em que a transação *New-Order* representa 87% de todas as transações executadas durante o benchmark. Nesta situação, não existem mais transações do tipo *Payment* sendo executadas.

Tanto a transação *New-Order* como *Payment*, no começo do bloco de operações demarcadas pelo início e fim da transação, recuperam de forma aleatória uma tupla para modificação da tabela `DISTRICT`. Além disso, a transação *Payment*, também recupera para modificação uma tupla da tabela `WAREHOUSE`, referente ao *homeWarehouse* do RTE que submeteu a transação. Na maioria das aplicações, essas tuplas seriam recuperadas com uma instrução do tipo `SELECT ...FOR UPDATE`. Isto causa um bloqueio do registro e faz com que outras transações que precisam utilizar essa tupla, aguardem até que o bloqueio seja encerrado. Essa técnica é conhecida como *Bloqueio Programático* [6].

O *framework* Hibernate utiliza um técnica de bloqueio diferente, chamada *Bloqueio*

Otimista. Esta técnica, apesar do nome, não faz bloqueio de tupla durante a transação. O controle de concorrência é realizado através do versionamento do registro. Um dos atributos do objeto que representa a tabela recebe a *Annotation* `@Version`. Quando uma transação realiza o *commit*, a coluna referente à versão do registro é incrementada.

Considere duas transações concorrentes T_1 e T_2 que recuperam o mesmo registro R_1. Ambas recebem a mesma versão do registro. Assumindo que T_1 completa antes de T_2, T_1 atualizará a versão de R_1, fazendo com que a versão atual da tupla seja maior do que a versão que T_2 possui. Quando T_2 realizar *commit*, o *framework* perceberá que T_2 possui uma versão desatualizada do registro. Isso causa o lançamento de uma exceção e faz com que todas as modificações realizadas por T_2 sejam revertidas através de um comando *rollback*.

Podemos concluir com isso que a concorrência entre transações tem grande impacto sobre o desempenho do Apache Derby.

6.3.3 Teste de desempenho - modo *multi-client*

No segundo teste adotamos uma metodologia um pouco diferente em relação ao teste anterior. No teste descrito em 6.3.2, todos os RTEs foram executados na mesma máquina em que os SGBDs estavam instalados, utilizando assim uma abordagem *multi-thread*.

Neste teste (chamado de *multi-client*), utilizamos praticamente os mesmos parâmetros do teste anterior. A diferença é que a máquina `ribeira` executou apenas os SGBDs. Os RTEs foram distribuídos pelas máquinas cliente, descritas na tabela 6.1. Cada máquina cliente executou apenas dois RTEs. A figura 6.6 apresenta o resultado do teste.

Avaliando o resultado apresentado na figura 6.6 podemos observar que o maior tpmC encontrando no teste 6.3.2 foi 3,6 vezes maior que o melhor resultado encontrado no cenário que os RTEs foram distribuídos em diversas máquinas. Muito dessa diferença se deve ao fato de que a máquina `ribeira` possui uma configuração muito superior em relação às máquinas clientes. Outro ponto que pode contribuir para isso é o fato de não existir tráfego de rede no teste 6.3.2.

Entretanto, é possível constatar que o MySQL não obteve o mesmo comportamento

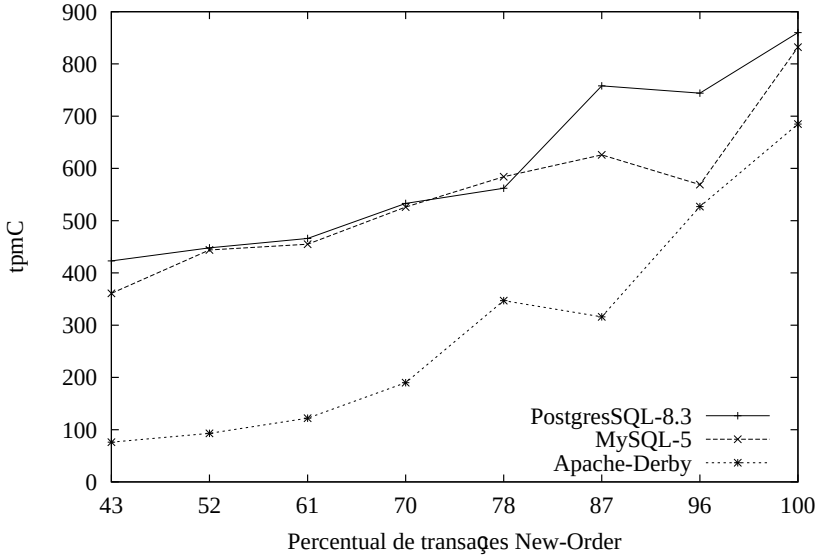

Figura 6.6: Resultado do teste *multi-client*.

apresentado no teste anterior. Ele começou com desempenho superior ao do PostgreSQL mas conforme o percentual de *New-Order* foi crescendo, sua curva de desempenho apresentou uma saturação não encontrada no teste anterior, e chegou a ser ultrapassada pela curva de desempenho do PostgreSQL. O Apache Derby por sua vez, repetiu o comportamento identificado no teste anterior, e só apresentou melhora de desempenho a partir do momento em que a transação *Payment* deixou de ser executada.

6.3.4 Teste de desempenho - modo *mono-client*

Com o intuito de avaliar como a abordagem *multi-thread* influencia o desempenho do benchmark, executamos o teste anterior, porém com uma diferença. Apenas uma máquina cliente foi responsável por executar os dez RTEs. Escolhemos a máquina com menor configuração entre todas as máquinas cliente. No caso a máquina escolhida foi a `caporal`.

A máquina ribeira mais uma vez foi responsável por executar apenas os SGBDs. O resultado do teste pode ser observado na figura 6.7.

Analisando o gráfico da figura 6.7 é possível identificar um comportamento similar ao encontrado no teste 6.3.2. Em ambos os testes, apenas um máquina foi utilizada para executar todos os RTEs.

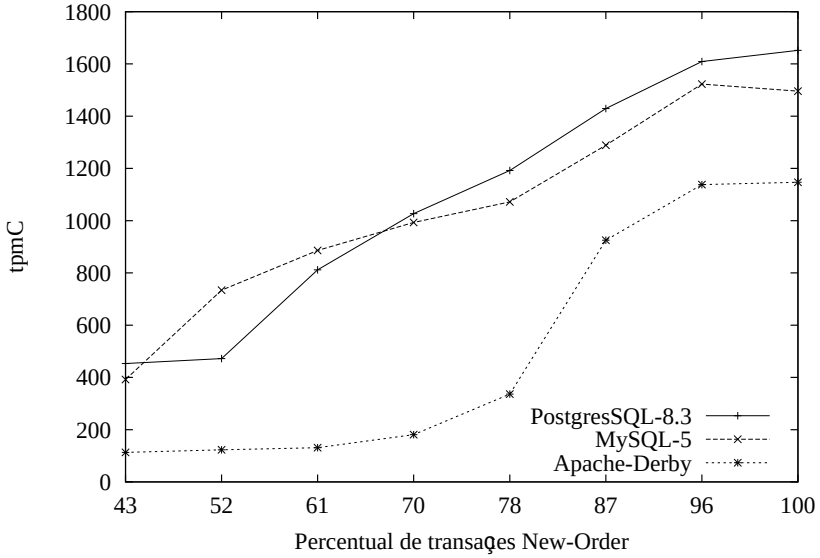

Figura 6.7: Resultado do teste *mono-client*.

A constatação de que a abordagem *multi-thread* influencia o desempenho do benchmark pode ser verificada quando se compara os resultados dos testes 6.3.3 e 6.3.4. Mesmo utilizando a máquina com menor capacidade computacional (`caporal`) dentre todas as máquinas clientes, o desempenho médio foi aproximadamente duas vezes maior do que o desempenho encontrado quando os RTEs foram espalhados em diversas máquinas cliente.

Os resultados ficam mais evidentes quando comparamos os resultados de cada teste para um mesmo SGBD. Estas comparações podem ser observadas na figuras 6.8, 6.9 e 6.10.

A figura 6.8 apresenta o resultado dos três testes de desempenho descritos anteriormente, considerando apenas o SGBD Apache Derby. Nesse gráfico fica claro a diferença entre um arquitetura centralizada (todos os RTEs executados na mesma máquina) e uma arquitura distribuída, na qual os RTEs são espalhados por diversas máquinas cliente.

A figura 6.9 ilustra o resultado dos testes de desempenho para o SGBD PostgreSQL. Novamente é possível observar um comportamento muito semelhante ao do Apache Derby. A arquitetura centralizada, mesmo utilizando a máquina com menor capacidade de processamento apresentado resultado aproximadamente 50% superior à arquitetura distribuída.

Na figura 6.10 observamos os resultados de desempenho apenas para o SGBD MySQL.

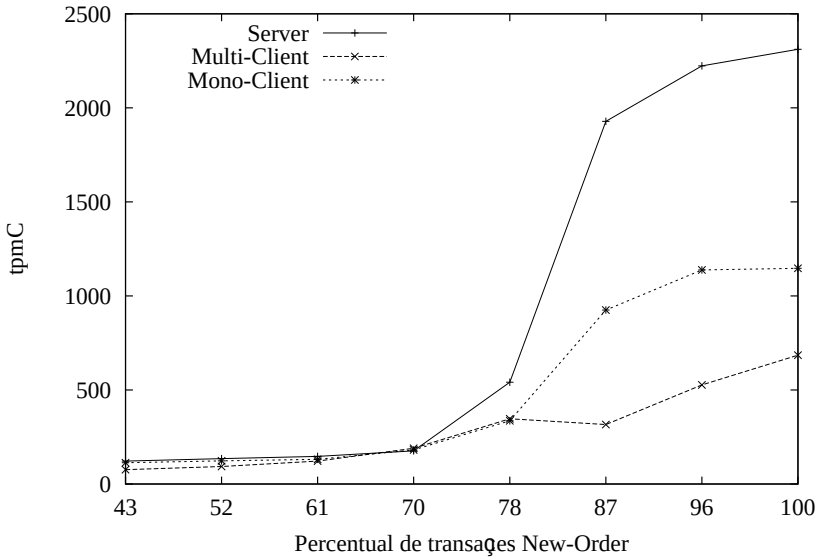

Figura 6.8: Resultado dos testes do SGBD Apache Derby.

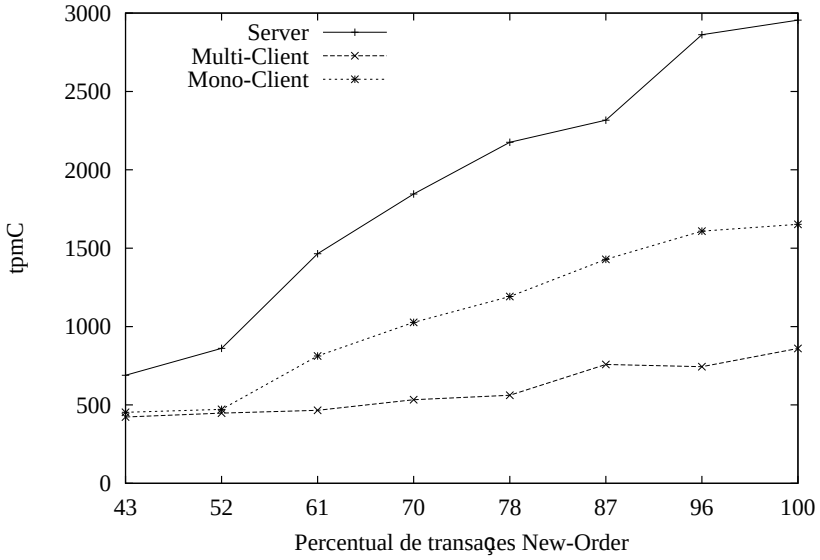

Figura 6.9: Resultado dos testes do SGBD PostgreSQL.

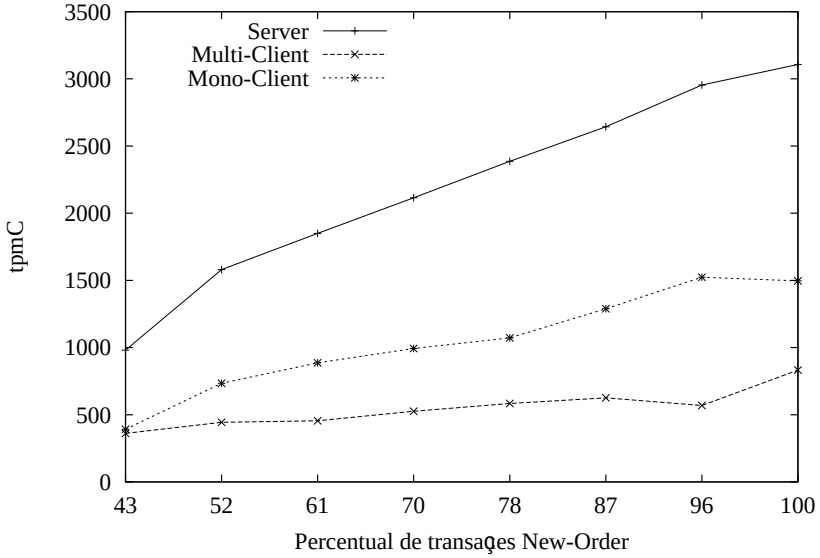

Figura 6.10: Resultado dos testes do SGBD MySQL.

Novamente, o mesmo padrão de desempenho encontrado nos dois SGBDs anteriores, aparece no MySQL.

6.3.5 Teste de desempenho vs. concorrência

Nos testes anteriores foi possível verificar como o TPCC-C3SL permite executar benchmarks para avaliar diversas situações. Arquiteturas centralizadas (utilizando uma abordagem *multi-thread*) e distribuídas, diferentes SGBDS e modificações no comportamento do benchmark. Porém, os testes realizados envolviam uma carga de trabalho mínima, ou seja, o banco de dados foi carregado com apenas um *Warehouse* e apenas dez RTE foram executados.

Neste teste procuramos avaliar como o SGBD se comporta com uma carga de trabalho bastante superior.

Para cada SGBD foram executados quatro benchmarks. A cada execução o número de RTEs executados era incrementado. Na primeira execução do benchmark foram executados 10 RTEs. Nas execuções seguinte foram executados, respectivamente, 30, 50, 100 e 250 RTEs.

Como cada *Warehouse* atende no máximo dez RTEs, o tamanho do banco de dados

também aumentou a cada execução. Na primeira execução o FS foi um (1). Já na última execução o FS foi 25. Considerando o número de registros que cada tabela deve conter para cada *Warehouse* e o tamanho aproximado de cada tabela (conforme a tabela 6.4), na última execução o SGBD foi carregado com aproximadamente 15 milhões de registros. O tamanho estimado do banco de dados para um FS 25 é de 1.875 MB.

A FD utilizada foi a recomendada pela especificação do TPC-C (conforme tabela 3.4), na qual 43% da transações executadas são transações *New-Order*.

O tempo de execução para cada benchmark foi de dez minutos, considerando um tempo de aceleração de um (1) minuto.

Os cenários selecionados para a execução do teste foram os mesmo utilizados no testes de desempenho realizados anteriormente. Utilizamos uma abordadem centralizada, executando todos os RTEs na máquinas `ribeira` e `caporal`. Também utilizamos a abordagem distribuída, na qual os RTEs foram espalhados pelas máquinas clientes.

Teste de concorrência - modo *server*. A figura 6.11 apresenta o gráfico de desempenho do teste de concorrência versus tpmC (modo *server*) para os três SGBDs analisados.

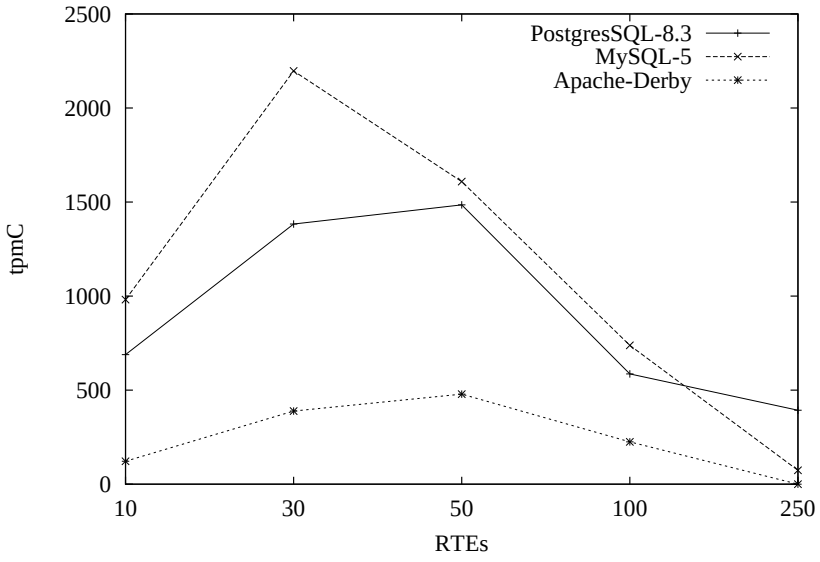

Figura 6.11: Resultado do teste de concorrência - modo *server*.

Observando a curva do PostgreSQL, é possível perceber uma queda de desempenho

extremamente acentuada a partir de 50 RTEs, apesar do número máximo de conexões, por padrão, ser definido como 100. Quando são executados 250 RTEs o desempenho é tão baixo que chega a ser praticamente o mesmo encontrado na execução com apenas 10 RTEs.

Outra métrica bastante impactada pelo nível de concorrência é o RT. Na execução com 30 RTEs o RT médio da transação *New-Order* era de 0,1 segundo. Quando foram executados 250 RTEs, o RT médio subiu para 3,8 segundos. O índice de reversão (*rollback* executados) da transação *New-Order* também foi alterado significativamente. Comparando a execução com 30 e com 100 RTEs, o índice passou de 15% para 62%.

O SGBD MySQL apresenta uma degradação de desempenho extremamente alta quando o número de RTEs passa de 30 para 50. O tmpC cai de 2.197 para 1.609, uma queda de 27%. Essa tendência de queda permanece conforme o número de RTEs em execução aumenta, chegando a 74 tpmC na execução de 250 RTEs. O RT médio alcançou a marca de 39 segundos, em média, para a execução de cada transação *New-Order*.

A curva do SGBD Apache Derby apresenta um desempenho extremamente baixo em relação aos outros SGBDs. O maior tpmC encontrado foi 479 quando 50 RTEs foram executados. O RT médio para 100 RTEs foi de 6,7 segundos. Valor muito acima do determinado na especificação do TPC-C, que diz que 90% das transações *New-Order* devem concluir em menos de 5 segundos.

Ainda sobre o Apache Derby, o último benchmark não pode ser executado porque o SGBD não conseguiu completar o carregamento inicial dos dados dentro do intervalo máximo de dez horas.

Teste de concorrência - modo *multi-client*. Para o teste em modo *multi-client* utilizamos as cinco máquinas clientes descritas na tabela 6.1. A fim de evitar a abordagem *multi-thread*, os RTEs foram espalhados uniformemente. Cada uma das máquinas executou 2, 6, 10, 20 e 50 RTEs, respectivamente, para os testes com 10, 30, 50, 100 e 250 RTEs.

A figura 6.12 apresenta o gráfico de desempenho do teste de concorrência versus tpmC

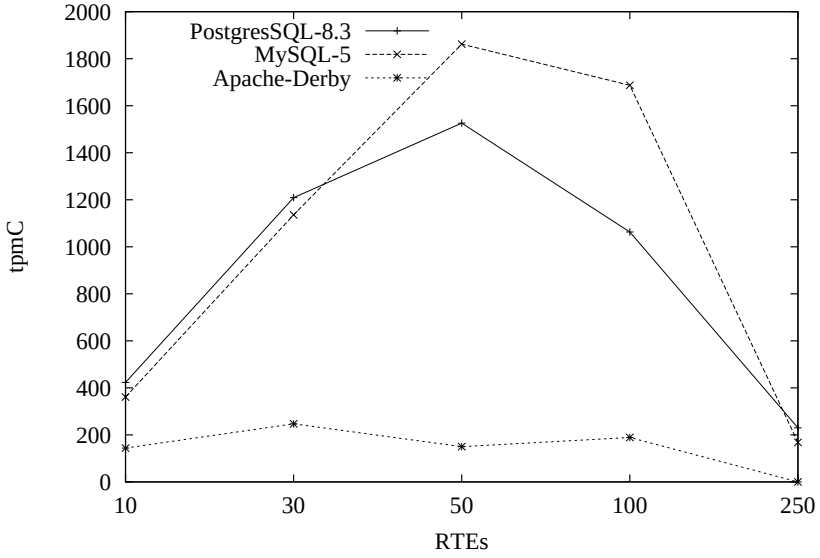

Figura 6.12: Resultado do teste de concorrência - modo *multi-client*.

(modo *multi-client*) para os três SGBDs analisados.

Desconsiderando o desempenho obtido pelo Apache Derby, os outros dois SGDBs, PostgreSQL e MySQL, apresentaram um resultado sifnificativamente melhor que o resultado da abordagem *multi-thread*, utilizada no teste modo *server*.

No teste anterior é possível notar uma saturação de desempenho quando se passa de 30 para 50 RTEs. Na abordagem distribuída, os SGBDs apresentaram crescimento de desempenho. Com 50 RTEs o desempenho da abordagem distribuída ultrapassa o desempenho da abordagem centralizada. Com 100 RTEs, o desempenho da abordagem distribuída é praticamente o dobro.

Novamente não foi possível executar o benchmark com 250 RTEs no SGBD Apache Derby porque ele não conseguiu completar o carregamento incial dos dados dentro do limite de dez horas.

Neste teste ficou evidente que a simulação de diversos clientes em uma mesma máquina influencia no resultado do benchmark, independentemente da máquina ter mais de um processador.

Teste de concorrência - modo *mono-client*. Para o teste em modo *mono-client* utilizamos a abordagem *multi-thread*, ou seja, apenas uma máquina cliente foi re-

sponsável por executar todos os RTEs. Para este teste a máquina escolhida foi a `talisker`.

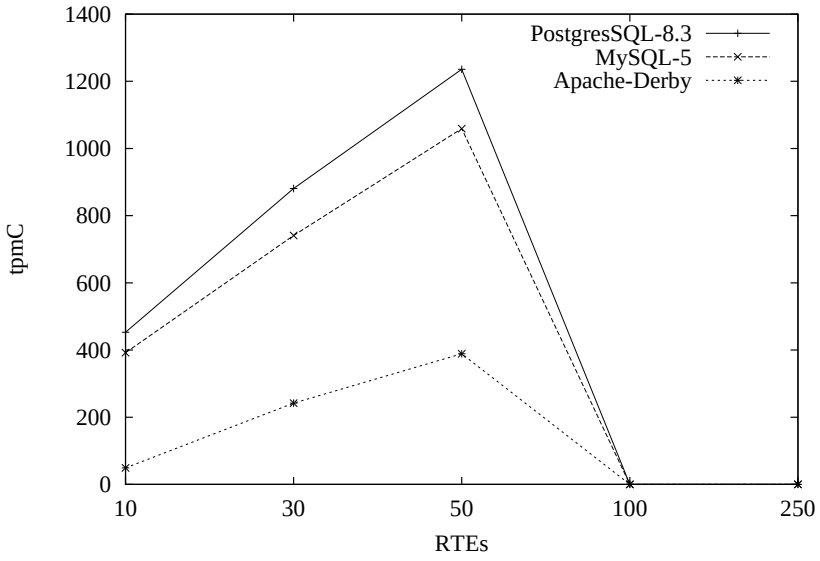

Figura 6.13: Resultado do teste de concorrência - modo *mono-client*.

A figura 6.13 apresenta o gráfico de desempenho do teste de concorrência versus tpmC (modo *mono-client*) para os três SGBDs analisados.

De maneira geral, o resultado de teste foi insatisfatório. Pode-se observar no gráfico que o desempenho obtido quando foram executados 100 RTEs foi 0. Nessa execução o benchmark não concluiu com sucesso devido a um erro de falta de memória (`java.lang.OutOfMemoryError: unable to create new native thread`) disponível para a alocação de *threads* Java.

Além disso, também não foi possível a execução do benchmark com 250 RTEs porque novamente os dados iniciais não foram carregados dentro do limite de dez horas.

6.4 Observações finais sobres os resultados

Neste capítulo descrevemos as metodologias utilizadas para realizar os testes com o TPCC-C3SL, bem como os resultados obtidos.

De modo geral podemos dizer que os SGBDs PostgreSQL e MySQL apresentaram resultados bastante similares. Já o SGBD Apache Derby alcançou resultados bastante inferiores, indicando que ainda precisa melhorar muito para chegar próximo dos outros dois SGBDs.

Um dos principais objetivos com a execução destes testes, foi apresentar o TPCC-C3SL como uma ferramenta capaz de executar benchmarks em diferentes cenários. Foram avaliadas arquiteturas distribuídas e centralizadas (execução dos RTEs no mesmo servidor dos SGBDs e em apenas uma máquina clientes) e o comportamento dos SGBDs conforme o benchmark tem suas características modificadas (executando mais uma determinada transação).

Outro grande objetivo foi avaliar quanto a arquitetura utilizada na execução do benchmark influencia no resultado final. Foi possível comprovar nos testes de concorrência que para uma pequena quantidade de RTEs o desempenho dos SGBDs em uma da arquitetura centralizada é muito superior ao desempenho encontrado na arquitetura distribuída. Porém, também comprovamos que, conforme o número de RTEs aumenta, a arquitetura centralizada vai perdendo desempenho, chegado a alcançar resultados inferiores aos da arquitetura distruibuída.

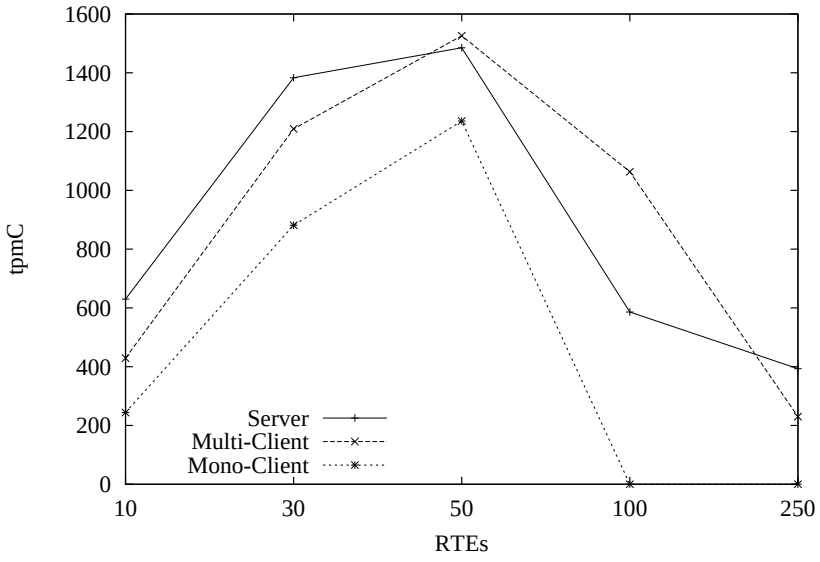

Figura 6.14: Resultado do teste de concorrência - PostgreSQL.

Podemos observar no gráfico ilustrado na figura 6.14 as curvas de desempenhos obtidas nos testes de concorrência para o SGBD PostgreSQL. No gráfico vemos que as execuções com poucos RTEs (10 e 30), o desempenho do SGBD é melhor na arquitetura centralizada (*modo server*). Porém, quando a execução envolve 50 RTEs, o desempenho na

arquitetura distribuída apresenta um ligeira vantagem. Quando 100 RTEs são executados, a arquitetura distribuída apresenta um desempenho aproximandamente 50% maior em relação à centralizada. Com 250 RTEs a arquitetura centralizada volta a apresentar melhor desempenho. Isto se deve muito provavelmente ao fato de que para alcançar esse número de RTEs, foi utilizada um abordagem *multi-thread* nas máquinas clientes fazendo com que cada uma executá-se 50 RTEs. Provavelmente se esse teste fosse executádo em número maior de máquinas clientes, o desempenho seria maior na arquitetura distribuída.

Ainda, foi possível verificar que dependendo das configurações da máquina utilizada na execução centralizada, o benchmark pode não ser executado com sucesso.

O apêndice H apresenta a tabulação dos resultados obtidos nos testes de concorrência versus desempenho. As tabela H.1, H.2 e H.3 apresentam os dados da métrica tpmC e as tabelas H.4, H.5 e H.6 apresentam os dados dos RTs da transação *New-Order*.

CAPÍTULO 7

CONCLUSÃO E TRABALHOS FUTUROS

Neste trabalho apresentamos o TPCC-C3SL, uma ferramenta para execução de benchmarks baseada na especificação do TPC-C. Com essa ferramenta, nós eliminamos algumas deficiências das ferramentas de benchmarks existentes.

Entre as melhorias propostas, podemos citar uma nova maneira de se avaliar os SGBDs em ambientes multi-usuário. Pelo resultado do testes, ficou demonstrado que a maneira como as transações são submetidas contra o SGBD, influencia no desempenho. As ferramentas anteriores utilizam uma abordagem *multi-thread*, na qual os usuários simulados submetiam as consultas a partir de uma mesma máquina. O TPCC-C3SL permite que os usuários façam múltiplas cópias suas através de diversas máquinas e execute cada uma dessas cópias em paralelo.

Em nossos teste comparamos cenários nos quais quais os RTEs submetiam as transações a partir de arquiteturas centralizadas (todos os RTEs em execução na mesma máquina) e distribuídas (os RTEs eram distribuídos por diversas máquinas clientes). Ficou comprovado que conforme o número de RTEs aumenta, o desempenho da arquitetura distribuída aumenta, e, em algum casos, chega a ser 100% maior que o desempenho da arquitetura centralizada.

Também apresentamos uma característica que o TPCC-C3SL proporciona os usuários de ajustar o benchmark as suas necessidades. Nossa ferramenta permite que se ajuste o peso de cada uma das cinco transações do TPC-C, dessa forma o usuário pode adaptar o benchmark para simular diferentes cargas de trabalho. Nos testes executados, pudemos comprovar que os SGBDs sofrem mudança de comportamento no seu desempenho conforme a aplicação muda suas características.

Por fim ainda podemos citar características que tornam o TPCC-C3SL mais próximo de ferramentas encontradas em ambientes reais. O TPCC-C3SL foi desenvolvido com mas

mesma técnicas utilizadas na construção de grandes sistemas corporativos. O TPCC-C3SL foi desenvolvido em três camadas: apresentação, lógica do negócio e persistência. Com essa divisão e com a utilização de *frameworks* tornamos o TPCC-C3SL independente de SGBD, possibilitando assim a avaliação de uma ampla variedade de SGBDs.

Trabalhos Futuros Por ser a primeira versão do TPCC-C3SL, acreditamos que existe espaço para aprimoramentos na ferramenta. Comparativos com outras ferramentas de benchmark são trabalhos futuros que ajudarão a confirmar a corretitude e os benefícios de nossa ferramenta.

Por exemplo, o relatório com as métricas primárias exigidas pelo TPC-C ainda não está completo. Os gráficos referentes à Distribuição de freqüência dos Tempos de Resposta, Tempo de Resposta vs. a produtividade da transação *new-order*, Distribuição de freqüência dos *Think Times* e Produtividade da transação *New-Order* versus o tempo decorrido ainda precisam ser desenvolvidos.

Também se faz necessária uma utilização mais ampla da ferramenta. Testes incluindo outros SGBDs considerados refêrencia no mercado podem produzir resultados interessantes.

Arquiteturas maiores também podem ser avaliadas. Por exemplo, a execução de 1000 RTEs em um *grid* com máquinas suficientes para executar apenas um RTE por máquina.

Pode-se sugerir também a adaptação do TPCC-C3SL para as outras especificações do TPC, por exemplo, o TPC-H. Uma adaptação para a avaliação de servidores de aplicação (muito comuns em ambientes web) é uma sugestão de trabalho futuro que pode apresentar resultados interessantes. Para essa adaptação pode-se utilizar a especificação do TPC para esse tipo de arquitetura (*TPC-App - Application Server*).

APÊNDICE A

TELAS DE E/S DAS TRANSAÇÕES DO TPC-C

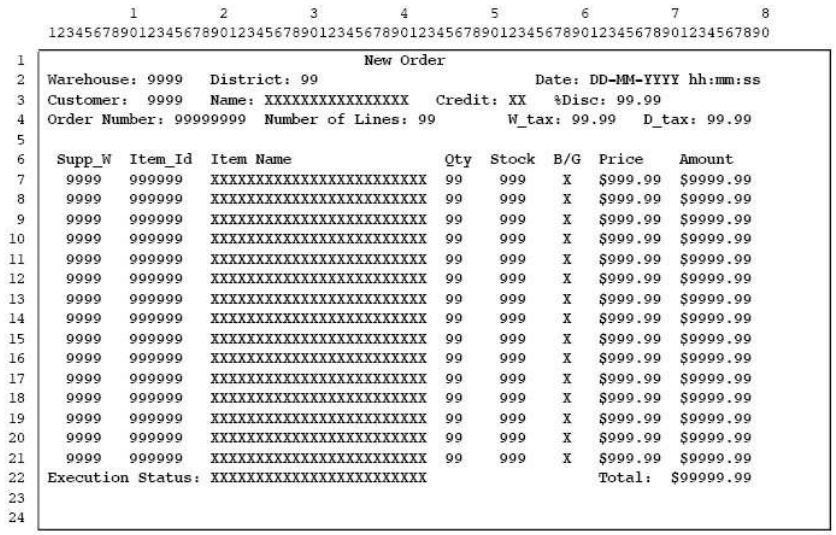

Figura A.1: Tela da transação New-Order.

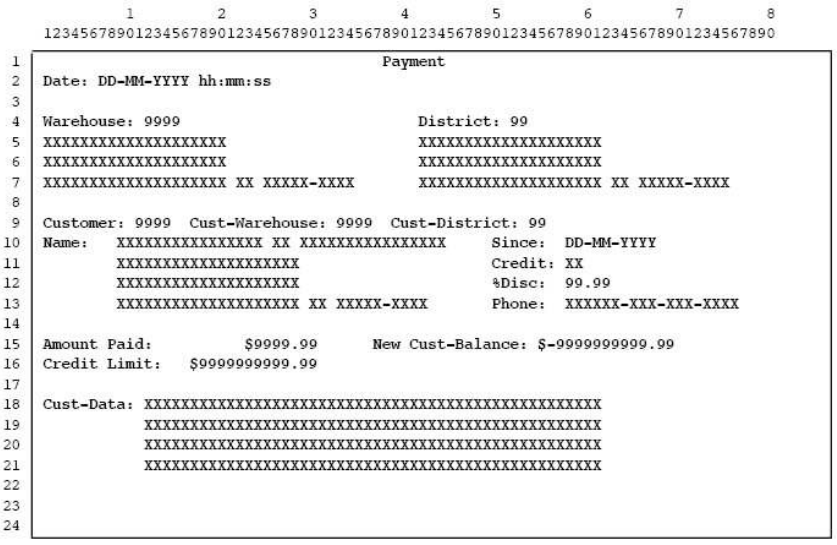

Figura A.2: Tela da transação Payment.

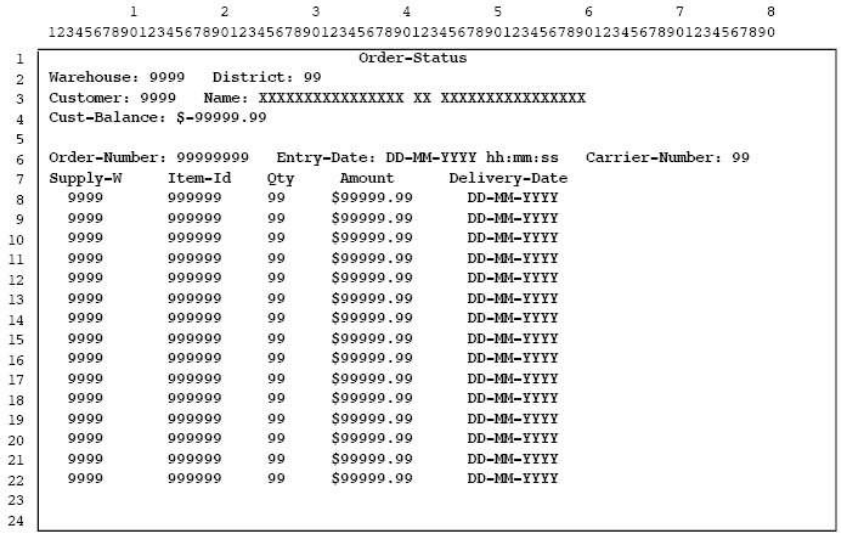

Figura A.3: Tela da transação Order-Status.

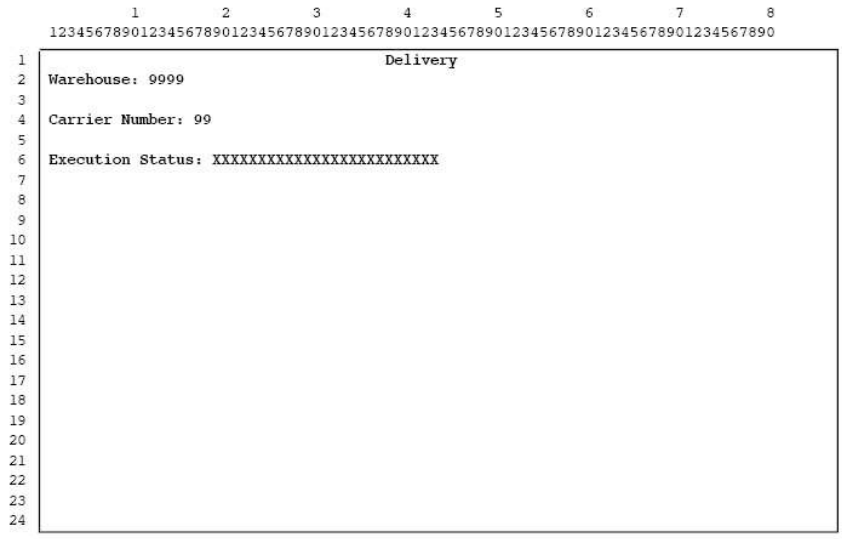

Figura A.4: Tela da transação Delivery.

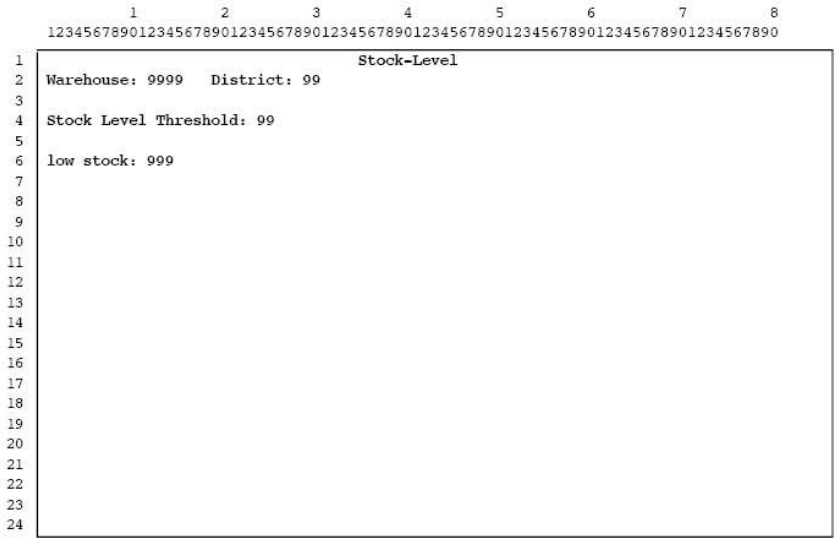

Figura A.5: Tela da transação Stock-Level.

APÊNDICE B

CÓDIGO DA TRANSAÇÃO *NEW-ORDER*

```
package br.ufpr.tpccufpr.transaction.impl;

//imports...

public class NewOrderTransaction implements TpccUfprTransaction {

   public final String name = "New-Order";
   public final int keyingTime = 18000;
   public final int thinkTime = 12;

   public TpccUfprTransactionResponse create(long homeWarehouse,
      long districtId) throws ItemNotFoundException, Exception {
      EntityManager em = JPAHelper.createEntityManager(false);
      NewOrderTransactionVO response = null;
      Warehouse warehouse = null;
      District district = null;
      Customer customer = null;
      Order order = null;
      long randomDistrict = 0;

      try {
         // A database transaction is started
         em.getTransaction().begin();

         // The row in the WAREHOUSE table with matching W_ID is selected and
         // W_TAX, the warehouse tax rate, is retrieved.
         warehouse = em.find(Warehouse.class, homeWarehouse);

          // The row in the DISTRICT table with matching D_W_ID and D_ ID is
          // selected, D_TAX, the district tax rate, is retrieved, and
          // D_NEXT_O_ID, the next available order number for the district, is
          // retrieved and incremented by one.
         randomDistrict = ValuesGenerator.nextInt(1, 10);
         district = em.find(District.class, new DistrictPK(randomDistrict,
            warehouse.getId()));
         long nextOrderId = district.getNextOId();
         district.setNextOId(nextOrderId + 1);

         // The row in the CUSTOMER table with matching C_W_ID, C_D_ID, and C_ID
         // is selected and C_DISCOUNT, the customer's discount rate, C_LAST, the
         // customer's last name, and C_CREDIT, the customer's credit status, are
         // retrieved.
         long customerId = ValuesGenerator.NURand(1023, 1, 3000);
         customer = em.find(Customer.class, new CustomerPK(customerId,
               district.getDistrictId(), homeWarehouse));

         // A new row is inserted into both the NEW-ORDER table and the ORDER
         // table to reflect the creation of the new order.  O_CARRIER_ID is set
```

```
// to a null value.  If the order includes only home order-lines, then
// O_ALL_LOCAL is set to 1, otherwise O_ALL_LOCAL is set to 0.
order = OrderDAO.create(nextOrderId, customer);
order.setCarrier(null);
NewOrder newOrder = NewOrderDAO.create(order);

int rbk = ValuesGenerator.nextInt(1, 100);

//order lines
OrderLine orderLine;
Collection<OrderLine> orderlines = new ArrayList<OrderLine>();
Collection<NewOrder> newOrderCollection = new ArrayList<NewOrder>();
Item item = null;
Stock stock = null;
float orderTotalAmount = 0;
for(int i = 1; i ≤order.getOrderLineCount(); i++) {
   orderLine = OrderLineDAO.create(order, Long.valueOf(i));

   // Verify if item is local
   if(orderLine.getWarehouseId().longValue() != orderLine.
      getSupplyWarehouseId().longValue())
      order.setAllLocal(0);

   // A non-uniform random item number (OL_I_ID) is selected using
   // the NURand(8191,1,100000) function.  If this is the last item
   // on the order and rbk = 1 (see Clause 2.4.1.4), then the item
   // number is set to an unused value.
   if(i == order.getOrderLineCount().intValue() && rbk == 1)
      throw new ItemNotFoundException("Not-Found␣condition.");

   //A quantity (OL_QUANTITY) is randomly selected within [1 ..  10]
   orderLine.setQuantity(ValuesGenerator.nextInt(1, 10));

   // The row in the ITEM table with matching I_ID (equals OL_I_ID)
   // is selected and I_PRICE, the price of the item, I_NAME, the
   // name of the item, and I_DATA are retrieved.  If I_ID has an
   // unused value (see Clause 2.4.1.5), a "not-found"condition is
   // signaled, resulting in a rollback of the database transaction
   // (see Clause 2.4.2.3).
   item = em.find(Item.class, orderLine.getItemId());


   // The row in the STOCK table with matching S_I_ID (equals
   // OL_I_ID) and S_W_ID (equals OL_SUPPLY_W_ID) is selected.
   // S_QUANTITY, the quantity in stock, S_DIST_xx, where xx
   // represents the district number, and S_DATA are retrieved.  If
   // the retrieved value for S_QUANTITY exceeds OL_QUANTITY by 10
   // or more, then S_QUANTITY is decreased by OL_QUANTITY;
   // otherwise S_QUANTITY is updated to (S_QUANTITY -
   // OL_QUANTITY)+91.  S_YTD is increased by OL_QUANTITY and
   // S_ORDER_CNT is incremented by 1.  If the order-line is remote,
   // then S_REMOTE_CNT is incremented by 1.
   stock = em.find(Stock.class, new StockPK(item.getId(), orderLine
      .getSupplyWarehouseId()));
   if(stock.getQuantity().intValue() - orderLine.getQuantity()
      .intValue() > 10)
      stock.setQuantity(stock.getQuantity().intValue() - orderLine
         .getQuantity().intValue());
```

```
      else
         stock.setQuantity(stock.getQuantity().intValue() - orderLine
            .getQuantity().intValue() + 91);

      stock.setYtd(orderLine.getQuantity());
      stock.setOrderCnt(stock.getOrderCnt() + 1);
      if(orderLine.getWarehouseId().longValue() != orderLine
         .getSupplyWarehouseId().longValue())
         stock.setRemoteCnt(stock.getRemoteCnt() + 1);

      orderLine.setStock(stock);

      // The amount for the item in the order (OL_AMOUNT) is computed
      // as:  OL_QUANTITY * I_PRICE
      orderLine.setAmount(orderLine.getQuantity().floatValue() * item
         .getPrice().floatValue());

      // The strings in I_DATA and S_DATA are examined.  If they both
      // include the string "ORIGINAL", the brandgeneric field for
      // that item is set to "B", otherwise, the brand-generic field
      // is set to "G".
      if(StringUtils.indexOf(item.getData(), "ORIGINAL") > -1 &&
         StringUtils.indexOf(stock.getData(), "ORIGINAL") > -1)
         orderLine.setBrandGeneric("B");
      else
         orderLine.setBrandGeneric("G");

      // A new row is inserted into the ORDER-LINE table to reflect
      // the item on the order.  OL_DELIVERY_D is set to a null value,
      // OL_NUMBER is set to a unique value within all the ORDER-LINE
      // rows that have the same OL_O_ID value, and OL_DIST_INFO is
      // set to the content of S_DIST_xx, where xx represents the
      // district number (OL_D_ID)
      orderLine.setDelivery(null);

      String orderLineDistrictId = StringUtils.leftPad(String
         .valueOf(orderLine.getDistrictId().longValue()), 2, "0");
      String methodNam = "getDist" + orderLineDistrictId;
      Class<? extends Stock> c = Class.forName("br.ufpr.tpccufpr.vo.Stock")
         .asSubclass(Stock.class);
      Method m = c.getMethod(methodNam, (Class[]) null);
      orderLine.setDistInfo((String)m.invoke(stock, (Object[]) null));

      // The total-amount for the complete order is computed as:
      // sum(OL_AMOUNT) * (1 - C_DISCOUNT) * (1 + W_TAX + D_TAX)
      orderTotalAmount += orderLine.getAmount().floatValue();

      // Add to Collection
      orderlines.add(orderLine);
   }

   // The total-amount for the complete order is computed as:
   // sum(OL_AMOUNT) * (1 - C_DISCOUNT) * (1 + W_TAX + D_TAX)
   order.setTotalAmount(orderTotalAmount * (1 - customer.getDiscount())
      * (1 + warehouse.getTax() + district.getTax()));

   order.setOrderLines(orderlines);
```

```
            newOrderCollection.add(newOrder);
            order.setNewOrder(newOrderCollection);

            em.persist(order);

            em.getTransaction().commit();

            response = new NewOrderTransactionVO();

         } catch (ItemNotFoundException ine) {
            em.getTransaction().rollback();
            response = new NewOrderTransactionVO();
            response.setRolledBack(true);
            response.setItemNotFoundException(true);
         } catch (Exception e) {
            em.getTransaction().rollback();
            response = new NewOrderTransactionVO();
            response.setRolledBack(true);
            response.setItemNotFoundException(false);
         } finally {
            response.setWarehouse(warehouse);
            response.setDistrict(district);
            response.setCustomer(customer);
            response.setOrder(order);
            em.close();
         }

         return response;

      }

      // other useful methods...
}
```

APÊNDICE C

CÓDIGO DA TRANSAÇÃO *PAYMENT*

```
package br.ufpr.tpccufpr.transaction.impl;

//imports ...

public class PaymentTransaction implements TpccUfprTransaction {

   private final String name = "Payment";
   private final int keyingTime = 3000;
   private final int thinkTime = 12;
   Logger logger = Logger.getLogger(PaymentTransaction.class);

   public TpccUfprTransactionResponse create(long warehouseNumber,
      long districtId) throws Exception {

      EntityManager em = JPAHelper.createEntityManager(false);
      PaymentTransactionVO response = null;

      try {
         boolean paymentByCustomerNumber = false;
      // The payment amount (H_AMOUNT) is randomly selected within [1.00 ..
      // 5,000.00].
      float amount = (float)ValuesGenerator.nextUniform(1.00, 5000.00, 2);

      // A database transaction is started
      em.getTransaction().begin();

      // The row in the WAREHOUSE table with matching W_ID is selected.
      // W_NAME, W_STREET_1, W_STREET_2, W_CITY, W_STATE, and W_ZIP are
      // retrieved and W_YTD, the warehouse's year-to-date balance, is
      // increased by H_ AMOUNT.
      //
      Warehouse warehouse = em.find(Warehouse.class, warehouseNumber);
      warehouse.setYtd(warehouse.getYtd().floatValue() + amount);

      // The row in the DISTRICT table with matching D_W_ID and D_ID is
      // selected.  D_NAME, D_STREET_1, D_STREET_2, D_CITY, D_STATE, and D_ZIP
      // are retrieved and D_YTD, the district's year-to-date balance, is
      // increased by H_AMOUNT.
      districtId = ValuesGenerator.nextInt(1, 10);
      District district = em.find(District.class, new DistrictPK(districtId, warehouse.getId()));
      district.setYtd((float)ValuesGenerator.round(district.getYtd() + amount, 2));

      // The customer is randomly selected 60// (C_W_ID , C_D_ID, C_LAST) and 40// C_D_ID ,
C_ID).
      int x = ValuesGenerator.nextInt(1, 100);//home or remote warehouse
      int y = ValuesGenerator.nextInt(1, 100);//mode of selection

      long customerDistrictId;
```

```
long customerWarehouseId;
Customer customer;
if(x ≤ 85) {
   customerDistrictId = district.getDistrictId();
   customerWarehouseId = warehouse.getId();
} else {
   customerDistrictId = ValuesGenerator.nextInt(1, 10);
   customerWarehouseId = ValuesGenerator.nextInt(1, (int)TpccUfprConstants.WAREHOUSES);
   if(TpccUfprConstants.WAREHOUSES > 1) {
      while(customerWarehouseId == warehouse.getId().longValue())
      customerWarehouseId = ValuesGenerator.nextInt(1, (int)TpccUfprConstants.WAREHOUSES);
   }
}

if (y ≤ 60) {
   String lastName = ValuesGenerator.randomLastName(ValuesGenerator
         .NURand(255, 0, 999));
   Query query = em
         .createQuery("from␣Customer␣c␣where␣c.districtId=:districtId␣and␣" +
               "c.warehouseId=:warehouseId␣and␣c.last=:last␣order␣by␣c.first");
   query.setParameter("districtId", customerDistrictId);
   query.setParameter("warehouseId", customerWarehouseId);
   query.setParameter("last", lastName);
   List<? extends Customer> customers = (List<? extends Customer>) query.getResultList();
   int listIndex = Math.round(customers.size() / 2);
   customer = customers.get(listIndex);

} else {
   long customerId = ValuesGenerator.NURand(1023, 1, 3000);
   customer = em.find(Customer.class, new CustomerPK(customerId,
         customerDistrictId, customerWarehouseId));
   paymentByCustomerNumber = true;
}

// C_BALANCE is decreased by H_AMOUNT. C_YTD_PAYMENT is increased by
// H_AMOUNT. C_PAYMENT_CNT is incremented by 1.
customer.setBalance(customer.getBalance().floatValue() - amount);
customer.setYtdPayment(customer.getYtdPayment().floatValue() + amount);
customer.setPaymentCnt(customer.getPaymentCnt().intValue() + 1);

// If the value of C_CREDIT is equal to "BC", then C_DATA is also
// retrieved from the selected customer and the following history
// information:  C_ID, C_D_ID, C_W_ID, D_ID, W_ID, and H_AMOUNT, are
// inserted at the left of the C_DATA field by shifting the existing
// content of C_DATA to the right by an equal number of bytes and by
// discarding the bytes that are shifted out of the right side of the
// C_DATA field.  The content of the C_DATA field never exceeds 500
// characters.  The selected customer is updated with the new C_DATA
// field.  If C_DATA is implemented as two fields (see Clause 1.4.9),
// they must be treated and operated on as one single field.
//String newCustomerData;
StringBuilder stringBuilder = new StringBuilder();
if(customer.getCredit().equals("BC")) {
   stringBuilder.append(customer.getCustomerId());
   stringBuilder.append(",");
   stringBuilder.append(customer.getDistrictId());
   stringBuilder.append(",");
   stringBuilder.append(customer.getWarehouseId());
```

```
            stringBuilder.append(",");
            stringBuilder.append(district.getDistrictId());
            stringBuilder.append(",");
            stringBuilder.append(warehouse.getId());
            stringBuilder.append(",");
            stringBuilder.append(amount);
            stringBuilder.append(",");
            stringBuilder.append(customer.getData());
            //customer.setData(StringUtils.substring(stringBuilder.toString(), 0, 500));
         }

         // H_DATA is built by concatenating W_NAME and D_NAME separated by 4
         // spaces.
         String data = warehouse.getName() + "␣␣␣␣" + district.getName();

         // A new row is inserted into the HISTORY table with H_C_ID = C_ID,
         // H_C_D_ID = C_D_ID, H_C_W_ID = C_W_ID, H_D_ID = D_ID, and H_W_ID =
         // W_ID.
         History history = HistoryDAO.create(customer);
         history.setAmount(amount);
         history.setData(data);

         if(customer.getHistories() == null)
            customer.setHistories(new ArrayList<History>());
         customer.getHistories().add(history);

         em.getTransaction().commit();

         response = new PaymentTransactionVO();
         response.setPaymentDate(new Date(System.currentTimeMillis()));
         response.setWarehouse(warehouse);
         response.setDistrict(district);
         response.setCustomer(customer);
         response.setPaymentByCustomerNumber(paymentByCustomerNumber);
         response.setAmount(amount);
         response.setCustomerData(stringBuilder.toString());

         }catch (Exception e) {
            em.getTransaction().rollback();
            logger.error(e.getMessage());
         } finally {
            em.close();
         }

         return response;
      }

      //other usefull methods ...
   }
```

APÊNDICE D

CÓDIGO DA TRANSAÇÃO *ORDER-STATUS*

```
package br.ufpr.tpccufpr.transaction.impl;

//imports...

public class OrderStatusTransaction implements TpccUfprTransaction {

   private final String name = "Order-Status";
   private final int keyingTime = 2000;
   private final int thinkTime = 10;
   Logger logger = Logger.getLogger(OrderStatusTransaction.class);


   public TpccUfprTransactionResponse create(long homeWarehouse,
      long districtId) throws Exception {
      EntityManager em = JPAHelper.createEntityManager(false);
      OrderStatusTransactionVO response = null;
      try {
         em.getTransaction().begin();

         // The district number (D_ID) is randomly selected within [1 ..  10]
         // from the home warehouse.
         long customerDistrictId = ValuesGenerator.nextInt(1, 10);
         District district = em.find(District.class,
            new DistrictPK(customerDistrictId, homeWarehouse));

            // The customer is randomly selected 60% of the time by last name
         // (C_W_ID, C_D_ID, C_LAST) and 40% of the time by number (C_W_ID,
         // C_D_ID, C_ID) from the selected district (C_D_ID = D_ID) and the
         // home warehouse number (C_W_ID = W_ID). This can be implemented by
         // generating a random number y within [1 ..  100];
         Customer customer;
         int y = ValuesGenerator.nextInt(1, 100);// mode of selection
         Query query;
         if (y ≤ 60) {
            String lastName = ValuesGenerator
                  .randomLastName(ValuesGenerator.NURand(255, 0, 999));
            query = em
                  .createQuery("from␣Customer␣c␣where␣c.districtId=:districtId␣and␣"
                        + "c.warehouseId=:warehouseId␣and␣c.last=:last␣order␣by␣c.first");
            query.setParameter("districtId", customerDistrictId);
            query.setParameter("warehouseId", homeWarehouse);
            query.setParameter("last", lastName);
            List<Customer> customers = (List<Customer>) query.getResultList();
            int listIndex = Math.round(customers.size() / 2);
            customer = customers.get(listIndex);

         } else {
            long customerId = ValuesGenerator.NURand(1023, 1, 3000);
```

```
            customer = em.find(Customer.class, new CustomerPK(customerId,
                  customerDistrictId, homeWarehouse));
         }

         // The row in the ORDER table with matching O_W_ID (equals C_W_ID),
         // O_D_ID (equals C_D_ID), O_C_ID (equals C_ID), and with the
         // largest existing O_ID, is selected.  This is the most recent order
         // placed by that customer.  O_ID, O_ENTRY_D, and O_CARRIER_ID are
         // retrieved.
         Order order;
         query = em.createQuery("from Order o where o.warehouseId=:warehouseId and " +
               "o.districtId=:districtId and o.customerId=:customerId order by o.orderId desc");
         query.setParameter("warehouseId", homeWarehouse);
         query.setParameter("districtId", customerDistrictId);
         query.setParameter("customerId", customer.getCustomerId());
         order = (Order)query.getResultList().get(0);
         Hibernate.initialize(order.getOrderLines());

         // All rows in the ORDER-LINE table with matching OL_W_ID (equals
         // O_W_ID), OL_D_ID (equals O_D_ID), and OL_O_ID (equals O_ID) are
         // selected and the corresponding sets of OL_I_ID, OL_SUPPLY_W_ID,
         // OL_QUANTITY, OL_AMOUNT, and OL_DELIVERY_D are retrieved.
         response = new OrderStatusTransactionVO();
         response.setWarehouseId(homeWarehouse);
         response.setDistrictId(district.getDistrictId());
         response.setCustomer(customer);
         response.setOrder(order);
         em.getTransaction().commit();

      } catch (Exception e) {
         em.getTransaction().rollback();
         logger.error(e.getMessage());
      } finally {
         em.close();
      }

      return response;

   }

   //other usefull methods ...
}
```

APÊNDICE E

CÓDIGO DA TRANSAÇÃO *DELIVERY* - PARTE *ON LINE*

```
package br.ufpr.tpccufpr.transaction.impl;

//imports...

public class DeliveryTransaction implements TpccUfprTransaction{

   private final String name = "Delivery␣(interactive␣portion)";
   private final int keyingTime = 2000;
   private final int thinkTime = 5;

   Logger logger = Logger.getLogger(DeliveryTransaction.class);

   public DeliveryTransactionVO create(long homeWarehouse, long districtId) {
      // The carrier number (O_CARRIER_ID) is randomly selected
      // within [1 ..  10].
      DeliveryTransactionVO delivery = new DeliveryTransactionVO(
            homeWarehouse, ValuesGenerator.nextInt(1, 10), new Date(System
                  .currentTimeMillis()));

      Util.deliveryQueue.add(delivery);

      // Run DeliveryBatch
      try {
         new DeliveryBatch().start();
      } catch (Exception e) {
         logger.error(e.getLocalizedMessage());
      }
      return delivery;
   }
   //other usefull methods ...
}
```

APÊNDICE F

CÓDIGO DA TRANSAÇÃO *DELIVERY* PARTE *BATCH*

```
package br.ufpr.tpccufpr.transaction.impl;

//imports...

public class DeliveryBatch extends Thread {
   private EntityManager em;
   private File deliveryFile;
   private FileChannel wChannel;
   private final String name = "Delivery␣(deferred␣portion)";
   Logger logger = Logger.getLogger(SummaryReportCommand.class);

   public DeliveryBatch() throws Exception {
      em = JPAHelper.createEntityManager(false);
      deliveryFile = new File(TpccUfprConstants.DELIVERY\_RESULT\_FILE\_NAME);

      // Set to true if the bytes should be appended to the file;
      // set to false if the bytes should replace current bytes
      // (if the file exists)
      boolean append = true;

      // Create a writable file channel
      wChannel = new FileOutputStream(deliveryFile, append).getChannel();

      // this.setPriority(Thread.MIN_PRIORITY);
   }

   public void run() {
      DeliveryTransactionVO delTransVO = null;
      StringBuilder strBuilder;
      long start = System.currentTimeMillis();
      long end = 0l;
      boolean completed = true;
      delTransVO = Util.deliveryQueue.peek();
      if (delTransVO != null) {
         Query queryNewOrder = null;
         Query queryOrder = null;
         try {
            strBuilder = new StringBuilder();
            strBuilder.append(Util.fomartDate(delTransVO.getQueuedDate(),
                  "dd/MM/yyyy␣HH:mm:ss"));
            strBuilder.append("|");
            strBuilder.append(delTransVO.getWarehouseId());
            strBuilder.append("|");
            strBuilder.append(delTransVO.getCarrierId());
            strBuilder.append("|");

            Warehouse warehouse = em.find(Warehouse.class, delTransVO
                  .getWarehouseId());
```

```
NewOrder newOrder;
Order order;
queryNewOrder = em
      .createQuery("from␣NewOrder␣where␣warehouseId=:warehouseId␣and␣"
            + "districtId=:districtId␣order␣by␣newOrderId");
queryNewOrder.setMaxResults(1);

queryOrder = em
      .createQuery("from␣Order␣where␣warehouseId=:warehouseId␣and␣"
            + "districtId=:districtId␣and␣orderId=:orderId");

int skipped = 0;
Date deliveryDate = new Date(System.currentTimeMillis());
em.getTransaction().begin();

for (District district : warehouse.getDistricts()) {
   // The row in the NEW-ORDER table with matching NO_W_ID
   // (equals W_ID) and NO_D_ID (equals D_ID) and with the
   // lowest NO_O_ID value is selected.  This is the oldest
   // undelivered order of that district.
   queryNewOrder.setParameter("warehouseId", delTransVO
         .getWarehouseId());
   queryNewOrder.setParameter("districtId", district
         .getDistrictId());

   newOrder = (NewOrder) queryNewOrder.getSingleResult();
   if (newOrder == null) {
      skipped++;
      // If no matching row is found, then the delivery of an
      // order for this district is skipped.

      // If this condition occurs in more than 1%, or in more
      // than one, whichever is greater, of the business
      // transactions, it must be reported.
   } else {
      // The selected row in the NEW-ORDER table is
      // deleted.
      em.remove(newOrder);

      // The row in the ORDER table with matching O_W_ID
      // (equals W_ ID), O_D_ID (equals D_ID), and O_ID
      // (equals NO_O_ID) is selected, O_C_ID, the customer
      // number, is retrieved, and O_CARRIER_ID is updated.
      queryOrder.setParameter("warehouseId", delTransVO
            .getWarehouseId());
      queryOrder.setParameter("districtId", district
            .getDistrictId());
      queryOrder.setParameter("orderId", newOrder
            .getNewOrderId());
      order = (Order) queryOrder.getSingleResult();
      order.setCarrier((int) delTransVO.getCarrierId());

      strBuilder.append("(");
      strBuilder.append(district.getDistrictId());
      strBuilder.append("-");
      strBuilder.append(order.getOrderId());
      strBuilder.append(")");
```

```
      // All rows in the ORDER-LINE table with matching
      // OL_W_ID (equals O_W_ID), OL_D_ID (equals O_D_ID), and
      // OL_O_ID (equals O_ID) are selected.  All
      // OL_DELIVERY_D, the delivery dates, are updated to the
      // current system time as returned by the operating
      // system and the sum of all OL_AMOUNT is retrieved.
      float amount = 0;
      for (OrderLine orderLine : order.getOrderLines()) {
         orderLine.setDelivery(deliveryDate);
         amount += orderLine.getAmount().floatValue();
      }

      // The row in the CUSTOMER table with matching C_W_ID
      // (equals W_ID), C_D_ID (equals D_ID), and C_ID (equals
      // O_C_ID) is selected and C_BALANCE is increased by the
      // sum of all order-line amounts (OL_AMOUNT) previously
      // retrieved.  C_DELIVERY_CNT is incremented by 1.
      Customer customer = em.find(Customer.class,
            new CustomerPK(order.getDistrictId(), district
                  .getDistrictId(), delTransVO
                  .getWarehouseId()));
      customer.setBalance(customer.getBalance().floatValue()
            + amount);
      customer.setDeliveryCnt(customer.getDeliveryCnt()
            .intValue() + 1);

   }// else
}// for
em.getTransaction().commit();

end = System.currentTimeMillis();

// Upon completion of the business transaction, the following
// information must have been recorded into a result file:  * The
// time at which the business transaction was queued.  * The
// warehouse number (W_ID) and the carried number (O_CARRIER_ID)
// associated with the business transaction.  * The district
// number (D_ID) and the order number (O_ID) of each order
// delivered by the business transaction.  * The time at which
// the business transaction completed.
strBuilder.append("|");
strBuilder.append(Util.fomartDate(new Date(System
      .currentTimeMillis()), "dd/MM/yyyy␣HH:mm:ss"));
strBuilder.append("\\r\\n");

try {
   // Write the ByteBuffer contents; the bytes between the
   // ByteBuffer's
   // position and the limit is written to the file
   wChannel.write(ByteBuffer.wrap(strBuilder.toString()
         .getBytes()));

   // Close the file
   wChannel.close();
} catch (IOException e) {
   logger.error(e.getLocalizedMessage());
}
```

```
            // Definitely removes the VO
            delTransVO = Util.deliveryQueue.poll();

            // Sleep for a random time no longer than 80s
            Thread.sleep(ValuesGenerator.nextInt(1, 80)// 1000);
         } catch (Exception e) {
            em.getTransaction().rollback();
            completed = false;
            logger.error(e.getLocalizedMessage());
         } finally {
         }

         if(!TpccUfprConstants.WITHIN\_RAMPUP && TpccUfprConstants.IS\_BENCH\_RUNNING) {
            em.getTransaction().begin();
            Metric m = new Metric();
            m.setTransaction(this.name);
            m.setStatus(completed == true ? "COMPLETED" : "INCOMPLETED");
            m.setResponseTime(ValuesGenerator.round((end - start) / 1000, 2));
            m.setTransactionResponseTime(ValuesGenerator.round((end - start) / 1000, 2));
            em.persist(m);
            em.getTransaction().commit();
         }
      }
      em.close();
   }
}
```

APÊNDICE G

CÓDIGO DA TRANSAÇÃO *STOCK-LEVEL*

```
package br.ufpr.tpccufpr.transaction.impl;

//imports...

public class StockLevelTransaction implements TpccUfprTransaction {

   private final String name = "Stock-Level";
   private final int keyingTime = 2000;
   private final int thinkTime = 5;
   Logger logger = Logger.getLogger(StockLevelTransaction.class);

   public TpccUfprTransactionResponse create(long homeWarehouse,
      long districtId) throws Exception {
      EntityManager em = JPAHelper.createEntityManager(false);
      StockLevelTransactionVO response = null;
      int minQtd = 0;
      List<Long> lowStockItems = null;
      try {
         em.getTransaction().begin();

         // The threshold of minimum quantity in stock (threshold) is selected at
         // random within [10 ..  20].
         minQtd = ValuesGenerator.nextInt(10, 20);

         // The row in the DISTRICT table with matching D_W_ID and D_ID is
         // selected and D_NEXT_O_ID is retrieved.
         District district = em.find(District.class, new DistrictPK(
               districtId, homeWarehouse));

         // All rows in the ORDER-LINE table with matching OL_W_ID (equals
         // W_ID), OL_D_ID (equals D_ID), and OL_O_ID (lower than D_NEXT_O_ID
         // and greater than or equal to D_NEXT_O_ID minus 20) are selected.
         // They are the items for 20 recent orders of the district.
         Query query = em.createQuery("select␣distinct␣itemId␣from␣OrderLine␣where" +
               "warehouseId=:warehouseId␣and␣districtId=:districtId␣and␣" +
               "orderLineId<:nextOIdInf␣and␣orderLineId␣≥:nextOIdTop");
         query.setParameter("warehouseId", homeWarehouse);
         query.setParameter("districtId", districtId);
         query.setParameter("nextOIdInf", district.getNextOId());
         query.setParameter("nextOIdTop", district.getNextOId() - 20);
         List<Long> itemsId = (List<Long>)query.getResultList();

         // All rows in the STOCK table with matching S_I_ID (equals OL_I_ID)
         // and S_W_ID (equals W_ID) from the list of distinct item numbers
         // and with S_QUANTITY lower than threshold are counted (giving
         // low_stock).
         query = em.createQuery("from␣Stock␣where␣itemId=:itemId␣and␣" +
            "warehouseId=:warehouseId␣and␣quantity<:threshold");
```

```
            query.setParameter("warehouseId", homeWarehouse);
            query.setParameter("threshold", minQtd);

            List<Stock> stocks;

            lowStockItems = new ArrayList<Long>();
            for (Long itemId : itemsId) {
               query.setParameter("itemId", itemId);
               stocks = (List<Stock>)query.getResultList();

               for (Stock stock : stocks) {
                  // Stocks must be counted only for distinct items.  Thus,
                  // items that have been ordered more than once in the 20
                  // selected orders must be aggregated into a single summary
                  // count for that item.
                  if(!lowStockItems.contains(stock.getItemId())) {
                     lowStockItems.add(stock.getItemId());
                     break;
                  }
               }
            }
            em.getTransaction().commit();
         } catch (Exception e) {
            em.getTransaction().rollback();
            logger.error(e.getLocalizedMessage());
         } finally {
            response = new StockLevelTransactionVO();
            response.setWarehouseId(homeWarehouse);
            response.setDistrictId(districtId);
            response.setStockLevel(minQtd);
            response.setLowStockItems(lowStockItems.size());
            em.close();
         }
         return response;
      }
      //other useful methods...
}
```

APÊNDICE H

DADOS ENCONTRADOS NOS TESTES DE CONCORRÊNCIA

	PostgresSQL-8.3		
RTE/Teste	Server	Multi-Client	Mono-Client
10	689	423	453
30	1383	1209	881
50	1485	1526	1236
100	586	1063	0*[a]
250	393	230	0*

[a]* Teste não executado por falta de memória disponível para alocar todos os RTEs.

Tabela H.1: tpmC encontrado nos testes de concorrência para o SGBD PostgreSQL.

	MySQL-5		
RTE/Teste	Server	Multi-Client	Mono-Client
10	981	361	392
30	2197	1136	741
50	1609	1862	1059
100	738	1687	0*[a]
250	74	168	0*

[a]* Teste não executado por falta de memória disponível para alocar todos os RTEs.

Tabela H.2: tpmC encontrado nos testes de concorrência para o SGBD MySQL.

	Apache-Derby		
RTE/Teste	Server	Multi-Client	Mono-Client
10	122	76	113
30	389	247	242
50	479	150	389
100	225	189	0*[a]
250	0**[b]	0**	0**

[a]* Teste não executado por falta de memória disponível para alocar todos os RTEs.

[b]** Não foi possível executar este teste porque o SGBD não conseguir executar o carregamento inicial dos dados dentro do intervalo máximo de 10 horas.

Tabela H.3: tpmC encontrado nos testes de concorrência para o SGBD Apache Derby.

	PostgresSQL-8.3		
RTE/Teste	Server	Multi-Client	Mono-Client
10	0,1	0,1	0,3
30	0,1	0,2	0,6
50	0,3	0,3	0,7
100	2,4	0,8	0*[a]
250	3,8	6,1	0*

[a]* Teste não executado por falta de memória disponível para alocar todos os RTEs.

Tabela H.4: RT encontrado nos testes de concorrência para o SGBD PostgreSQL (em segundos).

	MySQL-5		
RTE/Teste	Server	Multi-Client	Mono-Client
10	0,1	0,5	0,3
30	0,1	0,3	0,6
50	0,4	0,3	0,7
100	1,9	0,3	0*[a]
250	39	8,6	0*

[a]* Teste não executado por falta de memória disponível para alocar todos os RTEs.

Tabela H.5: RT encontrado nos testes de concorrência para o SGBD MySQL(em segundos).

	Apache-Derby		
RTE/Teste	Server	Multi-Client	Mono-Client
10	0,1	1,1	0,1
30	0,7	0,7	1,4
50	1,2	0,9	1,5
100	6,7	3,4	0*[a]
250	0**[b]	0**	0**

[a]* Teste não executado por falta de memória disponível para alocar todos os RTEs.

[b]** Não foi possível executar este teste porque o SGBD não conseguir executar o carregamento inicial dos dados dentro do intervalo máximo de 10 horas.

Tabela H.6: RT encontrado nos testes de concorrência para o SGBD Apache Derby(em segundos).

BIBLIOGRAFIA

[1] A measure of transaction processing power. *Datamation*, 31(7):112–118, 1985.

[2] American National Standards Institute. *ANSI X3.135-1992: Information Systems — Database Language — SQL (includes ANSI X3.168-1989).* 1989.

[3] Open Source Database Benchmark. The open source database benchmark, http://osdb.sourceforge.net/.

[4] Dina Bitton, David J. DeWitt, e Carolyn Turbyfill. Benchmarking database systems a systematic approach. *VLDB*, páginas 8–19, 1983.

[5] Haran Boral e David J. DeWitt. Methodology for database system performance evaluation. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 14(2):176–185, 1984.

[6] Bill Burke e Richard Monson-Haefel. *Enterprise JavaBeans 3.0 (5th Edition).* O'Reilly Media, Inc., May de 2006.

[7] Eduardo Cunha de Almeida. Estudo de viabilidade de uma plataforma de baixo custo para data warehouse. Dissertação de Mestrado, Universidade Federal do Paraná, Brazil, 2004.

[8] Apache Derby. The apache db project, http://db.apache.org/derby/.

[9] David J. DeWitt. The wisconsin benchmark: Past, present, and future. Jim Gray, editor, *The Benchmark Handbook for Database and Transaction Systems (2nd Edition).* Morgan Kaufmann, 1993.

[10] 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 Record*, 37(2):45–49, 2008.

[11] Jim Gray. The transaction concept: Virtues and limitations (invited paper). *VLDB*, páginas 144–154, 1981.

[12] Jim Gray e Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[13] PostgreSQL Global Development Group. Postgresql, http://www.postgresql.org/.

[14] John L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, 2000.

[15] IBM. Ibm smalltalk tutorial, http://www.inf.ufsc.br/poo/smalltalk/ibm/tutorial/oop.html.

[16] JCP. Jsr-000220 enterprise javabeans 3.0, http://jcp.org/aboutjava/communityprocess /final/jsr220/index.html.

[17] Diego R. Llanos. Tpcc-uva: An open-source tpc-c implementation for global performance measurement of computer systems. *ACM SIGMOD Record*, December de 2006. ISSN 0163-5808.

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

[19] MySQL. Mysql.

[20] NASA. Nas parallel benchmark, http://www.nas.nasa.gov/.

[21] Patrick E. O'Neil. The set query benchmark. *The Benchmark Handbook*. 1993.

[22] M. Tamer Özsu e Patrick Valduriez. *Principles of Distributed Database Systems, Second Edition*. Prentice-Hall, 1999.

[23] Omri Serlin. The history of debitcredit and the tpc. Jim Gray, editor, *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993.

[24] Abraham Silberschatz, Henry F. Korth, e S. Sudarshan. *Database System Concepts, 4th Edition*. McGraw-Hill Book Company, 2001.

[25] SUN. The java persistence api - a simpler programming model for entity persistence, http://java.sun.com/developer/technicalarticles/j2ee/jpa.

[26] SUN. Jdbc drivers, http://developers.sun.com/product/jdbc/drivers.

[27] SUN. Processes and threads, http://java.sun.com/docs/books/tutorial/ essential/-concurrency/index.html.

[28] TPC. Tpc benchmark c, http://www.tpc.org/tpcc/.

[29] TPC. Tpc benchmark h, http://www.tpc.org/tpch/.

[30] TPC. Transactional processing process council, http://www.tpc.org/.

[31] Carolyn Turbyfill, Cyril U. Orji, e Dina Bitton. As^{3}ap - an ansi sql standard scaleable and portable benchmark for relational database systems. *The Benchmark Handbook*. 1993.

MURILO RODRIGUES DE LIMA

EXECUÇÃO DISTRIBUÍDA DE BENCHMARKS EM SISTEMAS DE BANCOS DE DADOS RELACIONAIS.

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. Marcos Sfair Sunyé

CURITIBA

2008