UNIVERSIDADE FEDERAL DO PARANÁ

RODRIGO PIASSETTA

INTEGRAÇÃO DA FERRAMENTA APACHE JMETER À METODOLOGIA DE TESTE DE ESTRESSE DE BANCO DE DADOS BASEADO EM MODELO

CURITIBA/PR

2016

RODRIGO PIASSETTA

INTEGRAÇÃO DA FERRAMENTA APACHE JMETER À METODOLOGIA DE TESTE DE ESTRESSE DE BANCO DE DADOS BASEADO EM MODELO

Trabalho Final de Graduação apresentado à disciplina Trabalho de Graduação em Engenharia de *Software* II (CI-071), Curso de Bacharelado em Ciência da Computação, Departamento de Informática, Setor de Ciências Exatas da Universidade Federal do Paraná, como requisito para a conclusão do Curso.

Orientador: Prof.º Dr. Marcos Didonet Del Fabro

CURITIBA/PR

2016

À memória do Prof.º Dr. Alexandre Ibrahim Direne, grande e dedicado mestre, pesquisador e enxadrista.

AGRADECIMENTOS

Agradeço em primeiro lugar aos meus pais.

Aos amigos e colegas de produções Samuel Ferrari Lago, Rodrigo Barros Del Rei e Luiz Antônio Ferreira pela confiança dada a um recém-formado, pela parceria nos projetos e empreitadas, e pelas chances proporcionadas para que eu enfrentasse mais uma graduação; aqui estou recém-formado novamente.

Agradecimento mais do que especial ao Edson de Vulcanis, o Aranha, que com suas inúmeras patas abriu as portas para eu poder trabalhar com esses caras sensacionais acima citados.

A Universidade Federal do Paraná, ao Departamento de Informática e todo seu corpo docente, pelas oportunidades oferecidas e conhecimento compartilhado.

Aos professores Letícia Mara Peres e Marcos Didonet Del Fabro pela paciência e orientação para conclusão deste trabalho.

Ao saudoso mestre Alexandre Dirente, que me fez apreciar a área da Inteligência Artificial e abominar pesudocódigo.

Às amigas Tamy Ribeiro e Larissa Maciel, pela grande ajuda oferecida para que eu conseguisse finalizar este projeto. Vocês são incríveis.

À Aline Castanhari e ao Gil - amigo punk - pelo zelo e insistente preocupação com minha condição física, sempre encorajando a rápida reconquista do tão simples e tão vital ato de andar.

Aos amigos Allan Sklarow, Marco Aurélio Villa, Adriel Kotwiski, Ricardo Alvares, Adriana Kowalski, Dalila Lapuse e a todos os outros não citados - mas certos de que não são menos estimados - pelo incentivo e pelos momentos de lazer e distração, descanso mais do que necessário durante todo período de elaboração deste trabalho.

Por fim, um agradecimento mais do que especial ao Floyd, cuja companhia ainda traz muitas saudades. Com sua paciência felina me acompanhou por quase toda essa trajetória, sempre aguardando pelo meu retorno nas noites em que eu precisava ficar em aula até mais tarde.

RESUMO

Desde o surgimento dos banco de dados, os quais revolucionaram a forma de armazenar e gerir dados e informações, que esta tecnologia veio a ser cada vez mais adotada e difundida. Ao longo das últimas décadas, diferentes modelos de sistemas para gerenciamento de banco de dados foram desenvolvidos e propostos, tornando-se cada vez mais capazes de administrar e gerenciar um crescente volume de dados e atendendo as mais diversas demandas, as quais geram novas necessidades [29].

Esse cenário exige que os sistemas se reformulem constantemente, expandindo suas capacidades para atender a novas demandas. Atualmente, dentre as novas tecnologias apresentadas, está a classe de sistemas de banco de dados NewSQL, uma abordagem que propõe-se a dar suporte eficiente e eficaz ao gigantesco volume de dados com os quais convivemos nos dias atuais, como os que trafegam pela Internet e outras redes de computadores, sem que para isso exija uma configuração complexa como dos sistemas antecessores [7].

No entanto, com o surgimento de novas tecnologias é preciso elaborar novas abordagens para testar e avaliar essas novas tecnologias. Uma delas é a metodologia de teste de estresse baseada em modelo, ou MoDaST, desenvolvida para mensurar o desempenho dos novos sistemas de banco de dados NewSQL, em contraponto às metodologias existentes para testes de estresse em sistemas de banco de dados convencionais [9].

Neste trabalho de graduação propomos desenvolver uma aplicação que permita integrar a metodologia MoDaST ao Apache Jmeter [10], uma ferramenta bastante difundida e voltada para testes de carga e testes de estresse dos mais diversos serviços e protocolos, entre eles os sistemas de gerenciamento de banco de dados. Com esta aplicação será possível utilizar o JMeter para realizar testes em sistemas de banco de dados NewSQL e avaliar seu desempenho como pretende a metodologia MoDaST.

Além do mais, por utilizar uma ferramenta de código aberto, este trabalho pode vir a ser de contribuição para futuros projetos que pretendam utilizar essa metodologia para testes em outros sistemas ou mesmo aperfeiçoar a implementação oferecida por este projeto.

SUMÁRIO

LISTA DE FIGURAS

1. INTRODUÇÃO

Atualmente presenciamos um cenário de grande crescimento no volume de tráfego de dados e demanda de requisições nos sistemas informatizados existentes pelo mundo.

Com a chegada da Internet das Coisas e a popularização de aplicações móveis, conceitos como computação em nuvem e big data, que se refere ao gerenciamento e processamento de informação que excede a capacidade das tecnologias de informação tradicional [1], tornam-se cada vez mais presentes, tendo impacto direto tanto no desenvolvimento dos novos sistemas, que são desenhados para se adequarem a essa nova realidade, quanto na infraestrutura que dá suporte a antigos e novos sistemas, a qual tem que lidar com um número cada vez maior de usuários e de volume de dados processados por estes sistemas. O número de dispositivos comunicando-se entre eles próprios e com os seres humanos tende a aumentar com a Internet das Coisas, criando uma rede de comunicação altamente distribuída e aumentando significativamente a onipresença da Internet [2].

O conceito de computação em nuvem pode ser definido como um paradigma computacional e de infraestrutura, em que aplicações são disponibilizadas como serviços e suas funcionalidades acessadas por uma rede [3]. A nuvem pode ser entendida como uma metáfora para a Internet. Os usuários têm acesso apenas aos serviços que são oferecidos, tendo toda complexidade da infraestrutura encoberto por essa abstração a que chamamos de nuvem [4].

Com esse aumento no volume de dados e, consequentemente, no volume de transações nos sistemas e aplicações, uma categoria de sistema que é diretamente influenciado por essa mudança de cenário são os sistemas de gerenciamento de banco de dados, ou simplesmente, SGBD. Estes sistemas, além de conterem um conjunto de dados, comumente conhecido como banco de dados, apresentam meios adequados para armazenar e recuperar estes dados [5].

Os SGBD tradicionais foram bem sucedidos nos últimos 40 anos com relação ao processamento de transações, podendo-se citar como exemplo

o Oracle, PostgreSQL e IBM DB2. Porém, com o aumento do volume de transações, o qual foi comentado acima, estes sistemas têm recebido uma carga de trabalho além de suas capacidades, mostrando-se necessário revisar suas arquiteturas [6].

Entre novas tecnologias de SGBD desenvolvidas para lidar com esse crescimento da carga de trabalho, em alternativa aos SGBD tradicionais, se destaca o NewSQL, um SGBD relacional de alto desempenho e escalabilidade, mas que conserva o SQL e as propriedades ACID [7] (acrônimo derivado de atomicidade, consistência, isolamento e durabilidade).

Diferentemente dos SGBD tradicionais, que apresentam uma grande quantidade de parâmetros de configuração para otimização, os sistemas NewSQL apresentam uma abordagem denominada *no knobs*, que nada mais é do que uma maneira automatizada de configuração do sistema, sem a necessidade de intervenção externa [6].

A fim de medir o desempenho dos SGBD NewSQL, testes não-funcionais como testes de carga e estresse estão dentre as opções mais adequadas, principalmente levando-se em conta o caráter distribuído de alguns sistemas NewSQL disponíveis no mercado. Testes não-funcionais são uma técnica apropriada para avaliar o desempenho de uma aplicação. Entretanto, deve-se levar em conta de que na nuvem computacional o escopo de escalabilidade é muito mais abrangente do que em técnicas de teste de desempenho convencionais [8].

Então como medir os requisitos não-funcionais de sistemas NewSQL, como o desempenho, levando-se em conta que essa abordagem de configuração *no knobs* tem influência direta na arquitetura interna destes sistemas, não sendo suficiente adotar metodologias de teste utilizadas em SGBD tradicionais [9]?

Em sua tese de doutorado, *Model-based stress testing for database systems* [9], Meira contribui expondo uma metodologia específica para teste de sistemas de banco de dados NewSQL, denominada Teste de Estresse de Banco de Dados Baseado em Modelo, também referida pelo acrônimo MoDaST. Essa abordagem é baseada em uma máquina de estados denominada *Database State Machine (DSM)* a qual deve refletir os estados internos do

sistema de banco de dados, que são alterados de acordo com o desempenho do SGBD sob diferentes níveis de carga de trabalho [9].

Esta metodologia se difere de outras metodologias de teste de estresse em banco de dados por focar exclusivamente na indução e observação dos estados do SGBD, enquanto abordagens para SGBD tradicionais focam na relação entre a carga de trabalho submetida e o ajuste de configuração do sistema. Dentre essas abordagens, Meira apresenta a STEM, um acrônimo para *Stress Testing Methodology*, ou metodologia de teste de estresse, sobre a qual falaremos no próximo capítulo.

Neste trabalho desenvolvemos uma aplicação que estende a ferramenta de testes de carga Apache JMeter, permitindo a aplicação da metodologia MoDaST para a execução de testes tanto em sistemas gerenciadores de banco de dados tradicionais quanto sistemas NewSQL.

O JMeter é uma software de código aberto amplamente utilizado e patrocinado por empresas como Google, Microsoft, Facebook e AOL. É uma aplicação desenvolvida em Java e projetada para análise e medição de desempenho de diversos serviços, tais como os que utilizam requisições FTP, HTTP e JDBC, por exemplo [10].

Por não termos encontrado uma alternativa de implementação do JMeter aplicada para testes de estresse utilizando a metodologia MoDaST, este projeto colabora para difundir essa metodologia entre aqueles que utilizam o Apache JMeter.

Para execução dos testes foram escolhidos o SGBD tradicional PostgreSQL [11] e o SGBD NewSQL VoltDB [12]. Ambos foram escolhidos por possuírem licença de software livre, tendo assim liberdade de utilização para fins acadêmicos.

O esquema do banco de dados e a especificação da transação usados para os testes se baseiam na definição do *benchmark* TPC-B, como recomendado por Meira em sua tese. O *benchmark* TPC-B é destinado para mensurar o total de transações simultâneas que um sistema de gerenciamento de banco de dados pode tratar [13].

A linguagem adotada para o desenvolvimento da aplicação de controle e integração com o Jmeter foi a Java, uma vez que a API do JMeter se encontra na mesma linguagem.

No capítulo 2, nós abordamos os conceitos de cada elemento envolvido neste trabalho, como o sistema de banco de dados em teste, as metodologias de teste de estresse STEM e MoDaST e a ferramenta de testes adotados.

O capítulo 3 apresenta os detalhes da implementação da aplicação de integração do JMeter com a metodologia MoDaST, bem como as dificuldades, limitações e alternativas encontradas durante o desenvolvimento. Também apresentamos os resultados obtidos com os testes realizados nos SGBD PostgreSQL e VoltDB, com a análise dos nossos testes em comparação ao resultados dos testes realizados por Meira e descritos em sua tese sobre o MoDaST [9]. Por fim, no capítulo 4, apresentaremos a conclusão e propostas para trabalhos futuros.

2. CONTEXTUALIZAÇÃO E PROBLEMA DA PESQUISA

2.1. Introdução

Neste capítulo apresentamos uma visão geral de todos os elementos envolvidos neste trabalho, sendo eles: os sistemas de banco de dados relacionais tradicionais e NewSQL, o teste de estresse, a metodologia adotada Teste de Estresse de Banco de Dados Baseado em Modelo, o *benchmark* TPC-B e a ferramenta de testes Apache JMeter.

2.2 Sistemas de Gerenciamento de Banco de Dados (SGBD)

Um banco de dados pode ser definido como uma coleção organizada de informação que é utilizada por um computador [14]. Basicamente, trata-se de um modelo em que os dados são armazenados de forma estruturada. Esta estrutura é importante para que os dados possam ser recuperados posteriormente.

Dentre os diversos modelos de banco de dados existentes, damos destaque ao modelo de banco de dados relacional, o qual será alvo dos testes e da análise deste trabalho.

O modelo relacional foi proposto pela primeira vez por Edgar F. Codd, em seu artigo “Um modelo de dados relacional para grandes bancos de dados compartilhados” [15]. Com este novo modelo os usuários não precisavam mais ser especialistas em bancos de dados para conseguir recuperar uma informação. O modelo relacional abstraía como e onde essa informação era armazenada e recuperada, facilitando o uso e exigindo menos tempo e esforço para buscar uma informação. Na época o modelo de dados relacional foi visto como uma ideia revolucionária [16].

Os dados em um banco de dados relacional são estruturados sob os conceitos de tabelas (ou relações), linhas (ou registros), colunas (ou atributos) e campos (figura 1) [16].

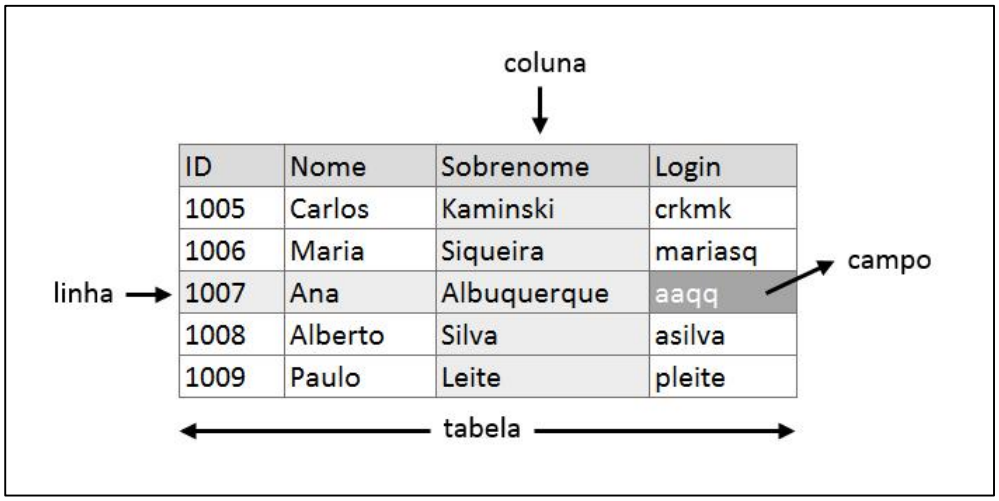


Figura 1 – Estrutura de um banco de dados relacional

Os sistemas que implementam o modelo de dados relacionais são conhecidos como Sistemas de Gerenciamento de Banco de Dados Relacional (SGBDR), podendo ser também chamados de SBGD de forma genérica. Estes sistemas usam uma linguagem conhecida como Linguagem de Consulta Estruturada, ou SQL para executar operações de manipulação dos dados armazenados, como inserção ou remoção de registros, atualização de campos e seleção de dados. A versão original da linguagem SQL foi desenvolvida pela IBM no início dos anos 70, estabelecendo-se desde então como o padrão para os bancos de dados relacionais [5].

As operações executadas em um SGBD são denominadas de transações. Uma transação é uma unidade lógica a qual contém uma ou mais declarações SQL. Ao ser processada, uma transação pode ser aplicada ao banco de dados ou revertida, mas sempre de forma integral, por ser uma unidade atômica. Nenhuma transação é processada parcialmente [17].

Diz-se que uma transação manteve a integridade se ela conservou as seguintes propriedades conhecidas como ACID: Atomicidade, Consistência, Isolamento e Durabilidade.

Um sistema de banco de dados está sujeito a receber um grande número de transações em um mesmo momento. Para que as propriedades ACID sejam preservadas, mais exatamente o isolamento, é necessário que o SGDB administre o controle de concorrência entre as diversas transações a serem tratadas. Esse controle é necessário para que a propriedade de

isolamento seja preservada. Uma variedade de mecanismos chamados de esquemas de controle de concorrência controlam a interação entre as transações, garantindo a propriedade de isolamento [5].

Embora o controle de concorrência garanta o isolamento e a integridade das transações, ele tem impacto no desempenho do sistema, em especial na vazão de transações. Esse é um ponto importante a ser destacado neste trabalho, uma vez que a vazão é uma das principais métricas para avaliar o desempenho de um SGBD e para a implementação da metodologia de testes que será apresentada nos capítulos a seguir.

A vazão representa o número de transações respondidas em um determinado intervalo de tempo, em outras palavras, o número de transações que são aplicadas ao banco de dados em um determinado intervalo de tempo. Outra métrica importante para se avaliar o desempenho de um sistema de banco de dados é o tempo de resposta, que nada mais que a quantidade de tempo que o sistema leva para completar uma única tarefa.

Há duas formas de melhorar o desempenho de um sistema, de acordo com o tipo de carga de trabalho submetida a ele. Se o sistema processa um grande volume de pequenas transações, ele pode ter sua vazão melhorada com o processamento em paralelo dessas transações. No entanto, se o sistema processa grandes transações, pode-se melhorar o tempo de resposta, realizando subtarefas de cada transação em paralelo [5]. Assim, diferentes SGBD apresentarão diferentes taxas de vazão e diferentes variações de desempenho, quando lidando com um alto número de transações a serem processadas.

Como descrito anteriormente, foram escolhidos os SGBD PostgreSQL e VoltDB como itens-alvo para a aplicação da metodologia que será tratada neste trabalho. O PostgreSQL é um sistema de gerenciamento de banco de dados relacional, similar ao modelo relacional descrito neste capítulo, e que também dá suporte ao modelo objeto relacional. Por sua vez, o VoltDB faz parte de uma classe de SGBD conhecida por NewSQL.

SGBD tradicionais como o PostgreSQL requerem intervenção manual para ajuste de desempenho, sendo necessários configurar inúmeros e complexos parâmetros de configuração. Costumam existir centenas de

ajustes nos principais sistemas tradicionais, o que acarreta em um grande esforço e com possibilidade de erros durante a tarefa de otimizar a configuração destes SGBD.

A nova arquitetura dos SGBD NewSQL apresentam a ideia de ajuste *no knobs*, diminuindo o trabalho manual de ajuste de desempenho e delegando essa tarefa ao próprio sistema. Pode-se definir a abordagem *no knobs* como soluções automatizadas para configuração dos parâmetros de um SGBD, onde *knobs* são basicamente os botões de ajustes os quais necessitam de intervenção humana para sua regulagem. O propósito do *no knobs* é diminuir ao máximo o número destes botões, tornando o processo de ajuste cada vez mais automatizado [6].

Como exemplo, podemos comparar o ajuste do número máximo de conexões entre o PostgreSQL e o VoltDB. Enquanto no PostgreSQL, por exemplo, o número máximo de conexões é definido pelo parâmetro MAX_CONNECTIONS, o VoltDB, através da abordagem *no knobs*, automaticamente ajusta esse parâmetro de acordo com os recursos disponíveis no sistema operacional [9].

Além dessa abordagem de ajuste de desempenho automatizado, a arquitetura dos SGBD NewSQL apresenta outras mudanças em relação aos SGBD tradicionais, sendo algumas delas:

- A possibilidade de operar de forma distribuída através da escalabilidade horizontal, diminuindo a concorrência e compartilhamento de recursos computacionais pela adição de mais servidores ao conjunto de máquinas que executam o SGBD. Apesar de existirem no mercado SGBD tradicionais que operam distribuidamente, uma característica geral e inerente aos SGBD tradicionais costuma ser a escalabilidade vertical, que consiste em adicionar recursos que aumentem o desempenho de um mesmo servidor SGBD, como memória e processamento [31];
- Carregamento da base de dados em memória (ao invés do carregamento em disco, como nos SGBD tradicionais), permitindo uma maior velocidade de leitura e escrita de dados;

- Processamento de transações em múltiplas *threads* únicas, o que permite evitar o uso de protocolos de controle de concorrência entre as transações.

Como foi anteriormente afirmado, estas mudanças arquiteturais requerem uma abordagem apropriada para a execução de testes de estresse.

Nas próximas seções teremos uma visão geral de testes de estresse em banco de dados e da metodologia adotada neste trabalho, o Teste de Estresse de Banco de Dados Baseado em Modelo (MoDaST).

2.3 Testes de Estresse em SGBD

A evolução e popularização das aplicações móveis, a migração de serviços essenciais para o ambiente virtual, a chegada da Internet das Coisas, entre outros fatores, têm contribuído para uma explosão no volume de dados em tráfego e a serem armazenados pelos sistemas de informação atuais.

Ao longo dos anos a armazenagem de dados tem tido seu custo aumentado cada vez mais em relação aos outros componentes que compõem o orçamento geral das empresas de Tecnologia de Informação, compreendendo uma parcela significativa de todos os gastos dessas organizações [18].

As empresas cada vez mais coletam e armazenam dados sobre o mercado e seus clientes, os tipos de dados deixaram de ser majoritariamente textuais ou numéricos para serem de natureza multimídia, como imagens, áudios e vídeos, políticas internas das empresas e legislações de cada nação determinam que muitos desses dados devem ser retidos e preservados por um grande período de tempo, além da Internet que tem contribuído significativamente para esse frenético aumento no volume de dados [18]. Estima-se que a quantidade de dados que as organizações de TI precisam gerir é aumentada ao dobro a cada ano. Como resultado os custos para manter a qualidade da infraestrutura de armazenamento de dados tornam-se cada vez mais altos, tornando-se um fator crítico para o sucesso de muitas dessas empresas [18].

Essa explosão de dados tem impacto diretamente nos sistemas de banco de dados, um dos principais atores na gestão e qualidade dos dados de

uma organização. Um SGBD mal dimensionado pode entrar em estado de degradação de desempenho, tendo impacto direto na disponibilidade e confiabilidade do sistema, do acesso aos dados e dos serviços de uma organização, podendo representar altos custos financeiros e perda em qualidade.

Dessa maneira, testes dos requisitos não-funcionais se fazem necessários para que se possa dimensionar corretamente a infraestrutura de um sistema à carga de utilização esperada e prever quando um sistema pode ser submetido a um estado de sobrecarga e, consequentemente, falhar. Entre os tipos de testes não-funcionais podemos citar os testes de estresse, no qual o sistema é submetido a cargas simuladas maiores às que usualmente ele deve gerir. O objetivo do teste de estresse é conduzir o sistema até os limites dos requisitos de desempenho especificados, de forma a estabelecer a carga sob a qual ele irá falhar e de que maneira ele irá falhar. Esse último ponto é bastante importante, pois espera-se que o sistema apresente uma desaceleração gradativa do seu desempenho. No entanto, ele pode falhar repentinamente e é muito importante saber sob quais condições podem ocorrer estas falhas catastróficas [19].

Ao definir um plano de teste de estresse, é preciso estabelecer métricas que sirvam como parâmetros para a escolha da carga de trabalho a qual o sistema será submetido. No caso de um SGBD, um parâmetro crítico de configuração é o número de conexões simultâneas permitidas pelo sistema. Um teste não-funcional simples que pode ser realizado é submeter o sistema a requisições de conexões simultâneas acima do limite configurado. Se o sistema for incapaz de garantir esse número máximo de conexões preestabelecido, ele apresenta um defeito que tem impacto direto na qualidade do serviço disponibilizado e consequente perda de desempenho [9].

No entanto, definir as métricas para avaliar o desempenho dos sistemas de banco de dados não é uma tarefa fácil, podendo resultar em resultados insatisfatórios e pouco confiáveis. Os SGBD tornaram-se maiores e mais complexos, visando atender uma gama maior de usuários e tipos de informação. Sob esse cenário, tem-se optado por técnicas de *benchmarking* para medir e avaliar o desempenho determinados aspectos dos SGBD [20].

Para este trabalho utilizamos um *benchmark* fornecido pela TPC, que é uma corporação sem fins lucrativos cuja missão é definir *benchmarks* de bancos de dados e processamento de transações, a fim de divulgar dados objetivos e verificáveis de desempenho para a indústria [13].

Em seu artigo, Meira utiliza o *benchmark* TPC-B para definir o esquema da base de dados e a especificação da transação a ser aplicada nos testes [9]. Este *benchmark* propõe uma abordagem de estresse para SGBD tradicionais, visando validar a integridade das transações submetidas ao sistema sob um alto processamento em disco (por exemplo, operações de E/S).

Basicamente, a transação simula uma movimentação bancária trivial, na qual um cliente faz um depósito ou um saque de um determinado valor em uma agência bancária. Tanto o saldo da conta do cliente quanto o balanço do caixa que o atendeu e o balanço geral da agência devem ser atualizados com o mesmo valor, mantendo a consistência da base de dados [30].

2.4 Metodologia de Teste de Estresse em Banco de Dados (STEM)

STEM, um acrônimo para *Stress Testing Methodology*, consiste em uma abordagem de teste incremental para avaliar o desempenho de SGBD tradicionais e expor possíveis defeitos. Esta metodologia estabelece uma série de cinco passos a serem seguidos, cada um com objetivos bem definidos, visando identificar eventuais defeitos ou perda de desempenho durante a execução do SGBD.

Como comentado anteriormente, SGBD tradicionais possuem uma vários parâmetros de configuração para otimização do sistema. A metodologia STEM tem foco na relação entre a carga de trabalho transacional e o ajuste dos parâmetros de configuração do SGBD. Basicamente, a carga de trabalho é o número de transações submetidas ao sistema. A aplicação de um crescente nível de carga de trabalho relacionado a uma variedade de ajustes de configuração dos parâmetros tornam possível determinar as condições sob as quais o sistema apresenta defeitos, principalmente aqueles relacionados à degradação de desempenho. O STEM também leva em consideração algumas variáveis que podem influenciar o resultado final dos testes, como a forma de

aplicação da metodologia e as funcionalidades do sistema operacional utilizado [9].

Embora os SGBD tradicionais possuam diversos parâmetros de configuração, podemos nos ater àqueles que são mais afetados por grandes cargas de transações. O STEM utiliza de variadas combinações desses parâmetros críticos para determinar a sequência de aplicação do seu método de teste.

Em um SGBD, o controle de conexões concorrentes e o gerenciamento do buffer de entrada, o qual é responsável por armazenar os dados e operações antes que sejam gravadas em disco, são parâmetros bastante sensíveis às variações na carga de transações que é recebida pelo sistema. No PostgreSQL, estes parâmetros são denominados como work_mem, que define a quantidade de memória disponível ao buffer, e max_connections, que é o limite de conexões simultâneas que o SGBD pode conceder [21].

Como tanto os parâmetros quanto a carga de trabalho podem assumir uma extensa gama de valores, o STEM propõe estabelecer 3 níveis para os parâmetros *work_mem* e *max_connections* e para a carga de trabalho: mínimo, médio e máximo.

Assim, considerando que cada uma das 3 variáveis envolvidas podem receber valores entre mínimo, médio e máximo, teríamos um conjunto de 27 combinações entre as variáveis work_mem, max_connections e a carga de trabalho. Porém, utilizando a abordagem de *pairwise testing*, um método combinatório no qual todos possíveis pares de variáveis são cobertos por ao menos um teste [9][22], e considerando que o nível da variável *work_mem* deve ser igual ou maior do que da *max_connections*, pois é necessário ter memória suficiente para administrar uma grande quantidade de conexões, conseguimos reduzir os possíveis o número de combinações para cinco. É essa particularidade que dá origem às cinco etapas da sequência de execução do STEM (figura 2) [9]. A figura 3 apresenta os níveis de ajustes das variáveis para cada etapa da metodologia.

Setup	Work_mem	Max_conn	Workload	Test Objective
1	Min	Min	Min	Installation
2	Max	Min	Max	Degradation baseline
3	Max	Max	Min	Tuning
4	Max	Max	Med	Robustness
5	Max	Max	Max	Stress

Figura 2 – Sequência de execução da metodologia STEM [9]

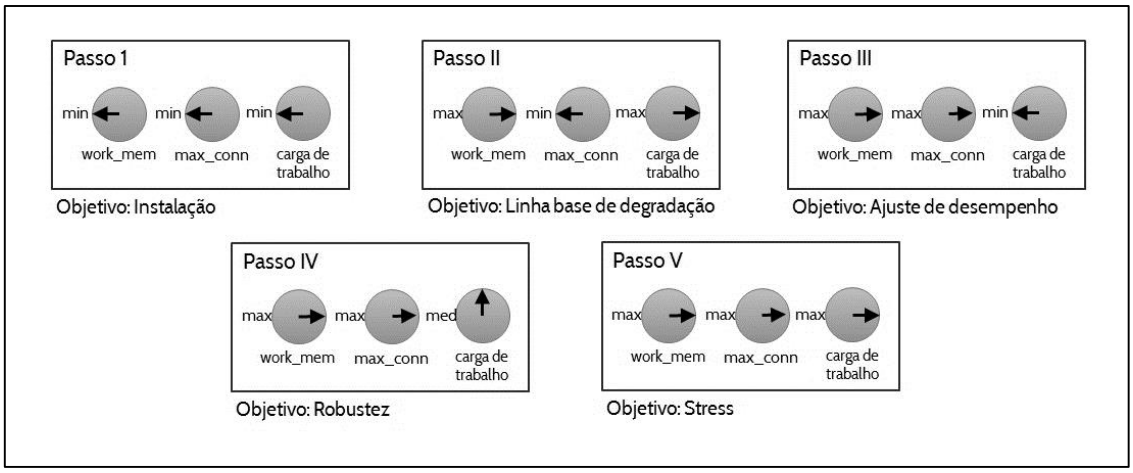


Figura 3 – Ajuste das variáveis para cada etapa da metodologia STEM

Essas cinco etapas são:

1) *Configuração inicial do SGBD sobre uma carga de trabalho inicial:* essa etapa consiste simplesmente em verificar a configuração inicial do sistema. A carga de trabalho aplicada é mínima, apenas visando garantir que o SGBD atende a todas as transações e que suas funcionalidades estão correspondendo às expectativas, não havendo qualquer problema em sua instalação.
2) *Tamanho do buffer (work_mem) ajustado para nível máximo sobre uma carga de trabalho de estresse:* nesta fase do teste, aplica-se uma carga de trabalho que demande um grande esforço da memória do sistema. O SGBD deve responder às requisições até onde suas definições de configuração permitirem, rejeitando todas aquelas que excedam sua capacidade. Com isto, podemos definir de maneira clara uma

linha base de degradação de desempenho do sistema e observar o tempo de reação e estabilização do SGBD em uma condição de estresse.

3) *SGBD ajustado para nível máximo sobre uma carga de trabalho mínima:* a terceira etapa utiliza os resultados obtidos no passo anterior para conferir se os parâmetros de configuração do SGBD estão ajustados em níveis apropriados. Para isso, as variáveis *work_mem* e *max_connections* são ajustadas para sua capacidade máxima.
4) *SGBD ajustado para nível máximo sobre uma carga de trabalho de estresse até o limite de desempenho do sistema:* o objetivo dessa etapa é testar a robustez do sistema. Esta etapa se relaciona diretamente com a segunda no que diz respeito ao objetivo, uma vez que, assim como na segunda fase do STEM, a expectativa é que o sistema responda às requisições até sua capacidade ser excedida.
5) *SGBD ajustado para nível máximo sobre uma carga de trabalho de estresse acima do limite de desempenho do sistema:* o objetivo dessa etapa é reproduzir uma situação de estresse máximo, observando as reações do sistema sob condições de degradação de desempenho. Assim como nas fases 2 e 4, o sistema rejeitará toda a carga de requisição de transações que exceda seus limites. A expectativa é que a taxa de rejeição de transações seja muito maior do que a quantidade de transações processadas com sucesso [9].

2.5 Metodologia de Teste de Estresse em Banco de Dados Baseado em Modelo (MoDaST)

A abordagem de teste de estresse de em banco de dados baseado em modelo, ou MoDaST (*Model-based Database Stress Testing*) é uma metodologia desenvolvida com objetivo de avaliar o desempenho de um sistema de gerenciamento de banco de dados, em particular os SGBD NewSQL, e encontrar defeitos não-funcionais através da submissão de um crescente nível

de carga de trabalho e da indução e observação de diferentes estados deste sistema.

Seu funcionamento se baseia na combinação de uma máquina de estados do banco de dados (*Database State Machine*) e de um módulo controlador de teste (*Test Driver*). Este controlador é responsável por comandar os testes, administrando a carga de trabalho a ser submetida ao sistema, enquanto a máquina de estados é encarregada por inferir os estados internos do SGBD testado, os quais podem ser: *Warm-up* ou inicialização, *Steady* ou estabilizado, *Under-Pressure* ou sob pressão, *Stress* ou estresse e *Thrashing* ou em degradação. A partir dessa dinâmica, o controlador analisa os resultados de cada teste e, de acordo com a resposta do sistema, define qual será o próximo estado e como se dará a transição, provendo ao sistema as informações e configurações necessárias para que a transição ocorra (figura 4) [9].

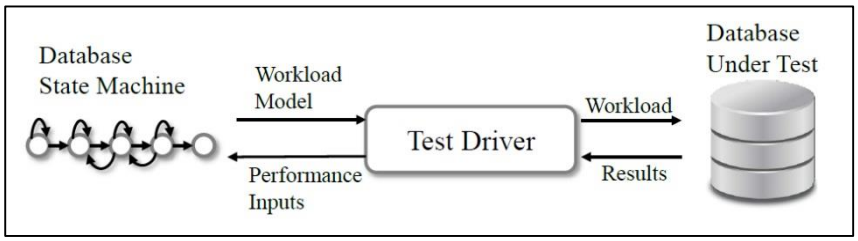


Figura 4 – Visão geral do funcionamento do MoDaST [9]

Através dessa abordagem que se percebe a diferença do MoDaST com relação ao STEM. Enquanto a primeira metodologia determina seus passos de forma dinâmica, de acordo com os resultados obtidos ao longo do teste, o STEM segue um protocolo com 5 etapas rigorosamente definidas, concentrando-se na configuração de parâmetros do sistema testado. Esta, inclusive, é uma das razões pelas quais o MoDaST é uma metodologia apropriada para testes em SGBD NewSQL. Uma vez que estes sistemas apresentam uma abordagem *no knobs*, a qual se propõe a minimizar a tarefa de configuração de parâmetros, é pertinente que a metodologia a ser aplicada desvie o enfoque da configuração do sistema.

Em seu artigo, Meira apresenta a seguinte definição formal da máquina de estados. A máquina de estados T é representada por uma 5-upla $(S, s_1, Ƒ, \beta, \tau)$, onde:

- $S = \{s_1, s_2, s_3, s_4, s_5\}$ é o conjunto de estados;
- $s_1 \in S$ é o estado inicial;
- $Ƒ \subset S$ é o conjunto de estados finais, onde $Ƒ = \{s_5\}$;
- β é o conjunto de variáveis de entrada, onde $\beta = < \Delta, \delta, \varphi >$;
- τ é uma função de transição definida como $\tau: S \times \beta \rightarrow S$.

A máquina de estados ainda possui 3 variáveis de entrada utilizadas para inferir o estado interno do SGBD, denotados por uma tupla $\beta = < \Delta, \delta, \varphi >$, onde Δ é a variação de desempenho, δ é a vazão de transações e φ é a tendência de desempenho [9].

As variáveis de entrada são necessárias para que a máquina de estados determine as transições de estado, sendo então relevante descrever o que cada uma dessas variáveis representa, como definido no artigo por Meira.

A variação de desempenho, ou Δ, representa a estabilidade do SGBD e calcula a dispersão da quantidade de transações tratadas por segundo pelo sistema de banco de dados. Se a variação de desempenho tende a zero após um determinado número de observações, podemos dizer que o SGBD está processando as transações que chegam a ele de forma estável.

A vazão de transações, ou δ, representa a taxa de transações tratadas por segundo em relação ao total de transações requisitadas por segundo. Esta métrica define o limite superior para que possa se dizer que o SGBD apresenta um comportamento estável. Se a vazão tende a 1, o sistema está tratando a maioria das transações que chegam a ele. À medida que este valor diminui, o SGBD se aproxima de um estado de perda de desempenho.

Por último, temos a tendência de desempenho, ou φ, que representa a esperada curva de desempenho do SGBD dentro de um determinado período de tempo. Se a tendência de desempenho tende a zero, podemos dizer que o SGBD está em um estado de degradação de desempenho, ou *thrashing* [9].

O conjunto de estados S que integra a máquina de estados é composto por: s_1 (*warm-up*), s_2(*steady*), s_3(*under-pressure*), s_4(*stress*) e s_5(*thrashing*). Estes estados podem ser assim definidos:

- **Warm-up (s_1):** é o estado inicial, que representa o SGBD a ser testado em processo de inicialização. Como o sistema não se encontra com todos seus serviços ativos e estabilizados, parte das transações requisitadas podem não ser atendidas. Tão logo o sistema se estabilize, ou seja, a variação de desempenho aproxime-se de zero, temos uma transição para o estado estável ou *steady*. Chamamos de t_w o limiar de transição para o próximo estado, logo, temos uma transição quando $\Delta < t_w$.
- **Steady (s_2):** diz-se que o SGBD se encontra em um estado estável se a sua variação de desempenho também se encontra estabilizada, tendendo a zero. A variável de entrada relevante nesse estado é a vazão de transações. Espera-se que em um estado estável, o SGBD processe de forma bem-sucedida a maioria das transações a ele submetidas, ou seja, $\delta \rightarrow 1$. Se a vazão se mantiver acima do limiar de estabilidade t_s, o SGBD se mantém nesse estado. No momento em que a vazão ficar abaixo desse limiar ($\delta < t_s$), entende-se que o sistema está operando sob pressão e temos uma transição para o próximo estado.
- **Under-Pressure (s_3)**: uma vez neste estado, o sistema se encontra em uma situação de alerta, operando próximo do seu limite. O SGBD ainda é capaz de tratar uma quantidade significativa das requisições a ele submetidas, mas se nenhuma atitude for tomada para contornar essa situação, a tendência é de que a vazão continue a cair, rejeitando cada vez mais requisições de transações. Isto reflete diretamente na variação de desempenho. Uma vez que ela aumente a ponto de ultrapassar o limiar t_{st} ($\Delta > t_{st}$), temos uma transição para o estado de estresse.

- **Stress (s_4):** este estado indica que o sistema está sobrecarregado, com um número de requisições que ultrapassa sua capacidade de processamento de transações. Se o estado anterior era considerado uma situação de alerta, este estado pode ser considerado como uma situação crítica. Mantendo-se nesse estado, o SGBD está sujeito a falhas, além da variação de desempenho, que tende a aumentar significativamente. A partir desse estado, devemos começar a monitorar a tendência de desempenho do sistema (φ). Com base nessa variável podemos predizer se o sistema está sendo conduzido para um estado de degradação. Se a condição de estresse se agravar e a tendência de desempenho ficar abaixo do limiar t_{th} ($\varphi < t_{th}$), assume-se que houve uma transição para o último estado.
- **Thrashing (s_5):** aqui o sistema apresenta um estado máximo de degradação de seu desempenho. A maior parte das requisições a ele submetidas serão rejeitadas, e o processamento da pequena parcela restante exigirá o consumo de uma grande quantidade de recursos pelo SGBD. Este é o estado final da metodologia MoDaST e, uma vez nesta condição, não há como retornar para os estados anteriores [9].

A metodologia MoDaST é vantajosa pois permite prever quando um sistema está sendo conduzido para o estado de degradação (ou *thrashing*). Através da tendência de desempenho é possível determinar se o sistema está desperdiçando recursos e agir antes que o SGBD entre em colapso.

O processo de teste de estresse, pela abordagem MoDaST, é dividido em iterações de teste, passos de teste ou *steps*. O controlador de teste é responsável por gerenciar todo processo, administrando a carga de trabalho adequada a cada iteração. Ao fim de cada iteração os valores das variáveis de entrada são recalculados e a máquina de estados e comparados aos limiares pré estabelecidos, determinando se deve ocorrer uma transição de estado.

O controlador de testes deve gerar dois tipos diferentes de carga de trabalho, um para testes de estresse no número de conexões concorrentes do SGBD e outro para testes de estresse no número de transações concorrentes requisitados ao sistema. É importante que ambos os casos sejam testados, pois exercitam módulos diferentes e críticos do SGBD [9].

A carga de trabalho no número de conexões impõe uma alta demanda no módulo de conexão do SGBD. Cada solicitação representa um usuário requisitando conexão ao banco de dados. O número de conexões solicitadas deve aumentar gradualmente a cada iteração de teste, mas apenas uma transação por conexão é requisitada, como definido abaixo (figura 5).

Iteração de teste	Conexões	Transações	Transações/iteração
1	10	1	10
2	20	1	20
3	30	1	30
...	...	...	...
n	n*10	1	n*10

Figura 5 – Carga de trabalho para testes de estresse no módulo de conexão do SGBD

O outro tipo de carga de trabalho submete uma grande quantidade de requisições no módulo de processamento de transações do SGBD a ser testado. Neste caso, temos um aumento gradual do número de transações requisitadas a cada iteração, enquanto o número de conexões solicitadas permanece constante, como definido na tabela abaixo (figura 6).

Na implementação de nossa aplicação será previsto ambos os casos de teste (teste do módulo de conexão e teste do módulo de processamento de transação), permitindo que ambos sejam realizados seguindo a metodologia MoDaST.

Iteração de teste	Conexões	Transações	Transações/iteração
1	10	10	100
2	10	20	200
3	10	30	300
...	...	...	...
n	10	n*10	n*10*10

Figura 6 - Carga de trabalho para testes de estresse no módulo de processamento de transações do SGBD

2.6 Apache JMeter

Este trabalho apresenta uma aplicação que integra a ferramenta Apache JMeter à metodologia MoDaST, permitindo assim executar testes de estresse em sistemas de gerenciamento de banco de dados da classe NewSQL com a utilização das funcionalidades do JMeter.

Apache JMeter é uma ferramenta de código aberto utilizada e patrocinada por inúmeras organizações, dentre elas a Microsoft, Google e AOL. Ele foi desenvolvido para permitir executar testes de carga, testar o comportamento funcional e medir o desempenho de servidores, serviços e sistemas.

Como exemplo de projetos, a AOL utiliza o JMeter para seus testes de desempenho e garantia da qualidade (QA) de seus produtos. A Lufthansa, empresa de aviação, utiliza essa ferramenta para testes de desempenho de sua aplicação do programa de fidelidade oferecido aos seus cliente. Podemos também citar como exemplo a Data Resolve, empresa que oferece soluções de segurança de dados, estabelecida na Índia que utiliza o JMeter para testar o desempenho de seus servidores [10]. No Brasil temos como exemplo a SERPRO, uma empresa pública cujo objetivo é a prestação de serviços em tecnologia da informação para o setor público [23].

A escolha por criar uma aplicação para estender esta ferramenta, de forma a implementar a metodologia MoDasT, se deve a inúmeras razões:

- Apache JMeter é uma conhecida e difundida ferramenta para testes de carga e medição de desempenho de diferentes tipos de serviços e sistemas, como servidores Web/HTTP, servidores FTP e, no nosso caso, sistemas de banco de dados. Isso torna relevante a produção de um trabalho visando integrar a

metodologia MoDaST com a ferramenta em questão, contribuindo para que futuros projetos acadêmicos possam usufruir deste trabalho, possibilitando ampliar a gama de testes aos quais se tem como objetivo;

- Apache JMeter é um software de código aberto, licenciado pela Licença Apache, sendo livre a utilização e desenvolvimento de aplicações integradas para fins acadêmicos;
- Apache JMeter permite a execução de testes distribuídos, requisito indispensável para implementação da metodologia escolhida;
- Apache JMeter possui uma extensa e completa API, permitindo o desenvolvimento de aplicações integradas à ferramenta;
- Apache JMeter é 100% desenvolvido em Java, favorecendo a portabilidade da aplicação de testes para diferentes máquinas de teste de diferentes configurações e sistemas operacionais, sem a necessidade de configurações morosas ou complexas de ambiente [24].

Desta maneira, o objetivo deste trabalho em estender o JMeter, permitindo que esta ferramenta execute testes de SGBD NewSQL com base na metodologia MoDaST, pode ser encarado como uma proveitosa contribuição inicial para aqueles que pretendem utilizar esta ferramenta para testes dessa categoria de sistemas de banco de dados.

3. PROJETO E IMPLEMENTAÇÃO DA APLICAÇÃO DE INTEGRAÇÃO DA FERRAMENTA JMETER À METODOLOGIA MODAST

3.1 Introdução

O objetivo desse trabalho é desenvolver uma aplicação que permita a integração da metodologia MoDaST com a ferramenta de testes Apache JMeter, a fim de realizar testes de estresse de acordo com essa metodologia, tanto em SGBDs tradicionais quanto aqueles da categoria dos NewSQL. Devido ao JMeter oferecer uma API implementada na linguagem Java [10], optou-se por desenvolver esse projeto de integração na mesma linguagem, assim como também pela portabilidade das aplicações nela desenvolvidas, uma vez que os testes serão executados de maneira distribuída. Desta forma é possível executar a aplicação de testes em diversas máquinas sem precisar se preocupar com detalhes de ambiente como arquitetura de hardware ou sistema operacional.

Dentre as soluções oferecidas pelo JMeter, temos uma interface de comunicação para a execução de testes distribuídos, no modelo cliente-servidor. O pacote JMeter oferece dois tipos de motores de testes, denominados JMeter-Server, o servidor, e JMeter-Client, os clientes de teste. Os clientes são controlados totalmente pela instância do motor de teste servidor, o qual é responsável por todas as tomadas de decisão durante a execução dos testes, limitando-se a executar as ações solicitadas pelo servidor e entregar os resultados obtidos. JMeter também provê interfaces para conexão com sistema de gerenciamento de banco de dados e requisição de transações, gerenciando o tempo de resposta e resultado de cada transação.

Como a intenção desse projeto é desenvolver uma solução para integrar uma metodologia de testes ao JMeter, não nos preocuparemos em implementar funcionalidades de comunicação entre os clientes de teste ou com o SGBD, delegando estas responsabilidades ao JMeter. A aplicação a ser desenvolvida será responsável por controlar o motor de teste servidor do JMeter, fornecendo os dados de entrada, determinando os passos de teste a serem executados e realizando as tomadas de decisão sobre os resultados

obtidos, com base nas definições da metodologia aplicada e no benchmark escolhido para os testes.

A aplicação de integração deve:

- Criar e configurar o plano de teste a ser executado, seguindo as definições da metodologia MoDaST e as especificações do *benchmarking* TPC-B;
- Inicializar a instância do servidor de teste JMeter;
- Inicializar as instâncias dos clientes de teste JMeter e conectá-las à instância do servidor de teste JMeter;
- Submeter os dados de entrada ao servidor de teste JMeter para a iteração de teste "i", iniciando como i = 1;
- Requisitar ao servidor de teste o início da iteração de teste "i";
- Coletar os resultados da iteração de teste "i" e, com base neles, tomar decisões para a iteração "i+1" de acordo com as definições da metodologia MoDaST e as especificações do *benchmarking* TPC-B;
- Preparar e iniciar a iteração de teste "i+1".

Na aplicação de integração também foi implementada uma máquina de estados, a qual deve refletir os estados internos do sistema de gerenciamento de banco de dados sob o teste de estresse. A vantagem desta abordagem que utiliza uma máquina de estados para os testes com relação a outras ferramentas de teste é permitir induzir e explorar os diferentes estados do SGBD sob variadas cargas de trabalho, podendo assim detectar condições de perda de desempenho ou estados de degradação do sistema [9]. Os detalhes da implementação da máquina de estados e de outros componentes da aplicação de integração serão expostos nas próximas seções deste capítulo.

3.2 Arquitetura da Aplicação de Integração

Como visto anteriormente, a ferramenta Apache JMeter provê em seu pacote soluções para a comunicação entre o motor de teste servidor e as diversas instâncias dos clientes de teste e para o controle destes clientes, implementações para a utilização dos mais diversos serviços e protocolos,

como por exemplo o HTTP, FTP e o JDBC, o qual será utilizado neste projeto por se tratar especificamente para comunicação com SGBDs, além de possuir facilidades e componentes específicos que permitem capturar e entregar os resultados dos testes em formatos que podem ser definidos pela nossa aplicação de integração, através da API fornecida pelo JMeter.

A aplicação de integração deve ser responsável pelas seguintes tarefas:

- Criar e configurar de forma dinâmica os planos de teste seguindo as especificações do *benchmarking* TPC-B, que estabelece o modelo das transações a serem processadas pelo SGBD e as definições de carga de trabalho do MoDaST, que sugere 2 tipos de carga de trabalho, uma para teste de estresse no módulo de conexão do SGBD e outra para teste de estresse no módulo de processamento de transações [9];
- Instanciar, configurar e gerenciar o servidor de teste JMeter;
- Determinar o número de clientes de teste envolvidos, instanciar e submeter cada um deles ao servidor de teste Jmeter, após verificar que se encontram comunicáveis e aptos para iniciarem o processo de teste;
- Controlar o processo de teste, dividindo-o em iterações ou conjunto de passos de teste, sendo que cada passo compreende a definição da carga de trabalho a ser submetida ao SGBD, a configuração do plano de teste para o passo de teste atual com a carga de trabalho definida, a preparação de cada cliente de teste com o plano de teste configurado, requisitar a execução do teste aos clientes e aguardar pela coleta dos resultados, a validação, processamento e análise dos resultados obtidos, a tomada de decisão com base nesses resultados, com uma possível transição da máquina de estados para um estado antecedente ou subsequente e o encaminhamento para uma próxima iteração;
- Persistir os resultados obtidos em cada iteração para análise posterior dos testes realizados.

Do ponto de vista estrutural, a aplicação de integração situa-se em uma camada de controle sobre as camadas já estabelecidas do JMeter. Todo envolvimento da aplicação com as camadas inferiores compreende apenas em se comunicar com a API fornecida pelo JMeter, delegando todos outros serviços e tarefas à implementação desta ferramenta (figura 7).

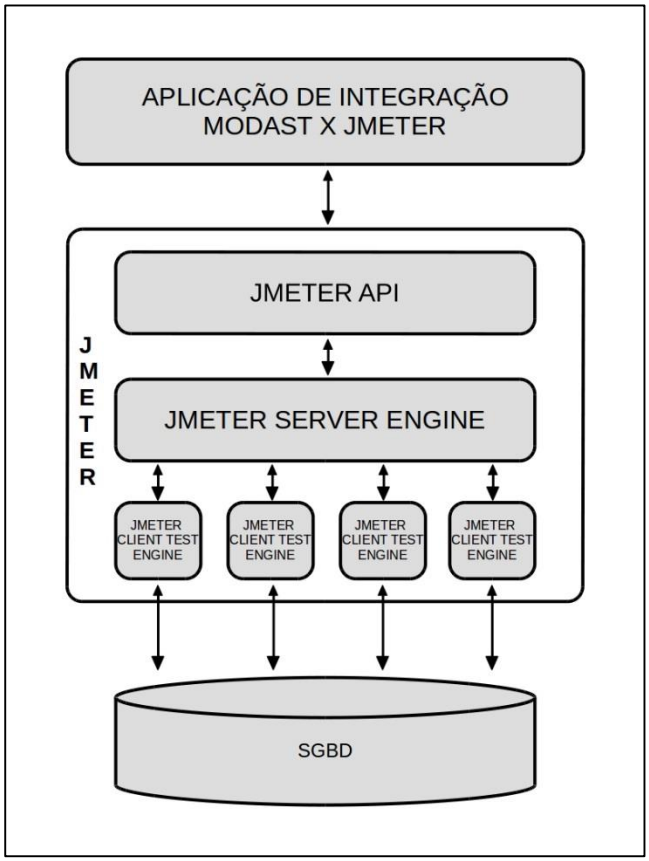


Figura 7 – Visão geral das camadas da integração com o JMeter para testes de estresse em SGBD

Podemos elencar como os principais desafios deste projeto:

- A implementação de uma máquina de estados (*Database State Machine*): o objetivo da metodologia MoDaST é inferir os estados internos do SGBD através da submissão de diferentes níveis de carga de trabalho, observando a resposta de desempenho do sistema e possíveis condições de perda deste desempenho. Sendo assim, é necessário que a aplicação possua uma máquina de estados, a qual reflita os estados

internos do sistema de banco de dados em teste e seja capaz de transitar entre esses estados de acordo com as decisões tomadas sobre os resultados obtidos a cada iteração de teste e carga de trabalho submetidos ao SGBD;

- A máquina de estados é o componente chave para a implementação da metodologia MoDaST. Todo seu funcionamento e considerações se baseiam na capacidade de identificar, observar e analisar o estado interno de um sistema de banco de dados de acordo com as condições e carga de trabalho a ele submetida. Pode-se então dizer que a máquina de estados é módulo intrínseco para uma bem sucedida implementação e integração da metodologia MoDaST ao Jmeter;
- A implementação de uma controladora de plano de teste: a metodologia MoDaST sugere duas maneiras de testar o desempenho dos SGBD, teste de estresse no módulo de conexão e teste de estresse no módulo de processamento de transações. A controladora do plano de teste é necessária para gerenciar e configurar o plano de teste de acordo com a categoria de teste escolhido, bem como a carga de trabalho, requisitada pelo controlado de teste, a ser submetida ao SGBD a cada passo do teste;
- A implementação de um módulo para monitoração, coleta e avaliação dos resultados de cada iteração de teste: como visto anteriormente, a metodologia MoDaST se baseia em 3 variáveis de desempenho para inferir o estado interno atual do SGBD em teste. Este módulo é responsável por coletar os resultados de cada iteração do teste de estresse, armazenar os resultados de todas as iterações em uma estrutura que permita que sejam recuperados e calcular o valor atual das variáveis de desempenho, permitindo que a máquina de estados tome decisões de acordo com esses novos valores;

- A implementação de um módulo controlador de teste: um módulo que tenha total controle sobre o processo de teste e sobre todos os outros módulos, definindo o plano de teste escolhido (teste de estresse de conexões ou de transações), controle do início e finalização de cada iteração de teste, gerenciamento das cargas de trabalho e armazenamento persistente dos resultados dos testes para análise futura;
- A implementação de um módulo principal: um módulo de controle da aplicação, responsável pelo tratamento dos parâmetros de entrada inseridos pelo usuário, pelas configurações iniciais do servidor de teste JMeter e dos clientes de teste, inicialização do módulo controlador de teste e inicialização geral do processo do teste a ser executado.

3.3 Desenvolvimento e Implementação

Nesta seção expomos os detalhes gerais do desenvolvimento e implementação da aplicação proposta nesse projeto, assim como a descrição do funcionamento dos principais módulos que compõem esta aplicação de integração da metodologia MoDaST ao Jmeter.

3.3.1 Módulo Principal

É o módulo de controle geral da aplicação, realizando tarefas e rotinas de inicialização, como a identificação dos parâmetros passados pelo usuário através da linha de comando, configuração da aplicação, carregamento do arquivo de configuração global do JMeter, carregamento do componente de controle de teste e dos clientes de teste JMeter, bem como requisitar o início da rotina de testes ao controlador de teste e imprimir os dados dos resultados do teste em disco ao final da rotina de testes.

A lista de parâmetros que devem ser informados ao se executar a aplicação são:

- Caso de teste a ser executado: Teste de estresse no módulo de conexão do banco de dados ou Teste de estresse no módulo de processamento de transações;

- Tipo de teste: Teste de estresse em SGBD NewSQL (VoltDB) ou Teste de estresse em SGBD tradicional (PostgreSQL);
- Endereço IP do SGBD a ser testado;
- Porta do serviço do SGBD a ser testado;
- Nome da base de dados a ser testada;
- Nome de usuário com privilégios de acesso à base de dados a ser testada;
- Senha do usuário;
- Quantidade de iterações de teste a serem executadas (o valor "0" indica que o teste será interrompido apenas quando o SGBD transitar para o estado de degradação).

Após o processamento dos parâmetros de entrada, é efetuada a configuração inicial do Jmeter e o carregamento do arquivo de configuração global do JMeter (anexo I), o qual apresenta dentre seus diversos parâmetros, os endereços IP dos clientes de teste remotos e quais dados dos resultados dos testes devem ser gravados ao final da execução dos testes.

O módulo principal também é encarregado de criar e configurar o controlador de testes, que irá gerenciar todo processo do teste a ser executado. O controlador de teste é definido de acordo com o caso de teste e o tipo de teste selecionados pelo usuário. O caso de teste se refere ao módulo do SGBD a ser testado, podendo ser um teste de estresse de conexões ou um teste de estresse de transações, enquanto o tipo de teste se refere à classe de SGBD que será alvo do teste, tendo como opções um SGBD NewSQL (VoltDB) e um SGBD tradicional (PostgreSQL).

Por fim, com o controlador de teste criado e pré-configurado. O módulo principal deve instanciar os clientes de teste JMeter para cada endereço IP especificado no arquivo de configuração global citado acima (clientes remotos), além de um cliente de teste para a máquina hospedeira da nossa aplicação (cliente local).

Para os clientes de teste utilizamos a própria implementação disponibilizada na API do JMeter. O cliente de teste remoto é definido pela classe ClientJMeterEngine, enquanto o cliente de teste local é definido pela

classe StandardJMeterEngine, ambas as quais implementam a interface JmeterEngine, a qual determina os métodos necessários para a execução dos testes em um cliente JMeter [25].

Durante a criação dos clientes remotos, é verificado se há comunicação com estes clientes. Se por qualquer motivo um ou mais clientes estiverem inalcançáveis, serão descartados do processo de teste. Os clientes de teste que responderem de forma bem-sucedida dentro do limite de tempo esperado, são adicionados ao controlador de teste.

Uma vez que a aplicação e o JMeter estão adequadamente configurados, o controlador de teste ajustado para o caso e tipo de teste selecionado pelo usuário e os clientes de teste devidamente inicializados comunicando-se com a aplicação principal, é requisitado ao controlador o início da rotina de testes do SGBD escolhido.

A rotina de testes é encerrada após esgotarem o número de iterações de teste determinado pelo usuário ou o SGBD em teste entrar em estado de degradação (ou *thrashing*). Os resultados obtidos nos testes são então gravados em um arquivo e a aplicação é encerrada.

3.3.2 Módulo Controlador de Teste (*Test Driver*)

O módulo controlador de teste centraliza integralmente todo processo de teste, o qual é definido pela classe base AbstractTestController e implementado pelas classes derivadas PostgreSQLTestController, destinada aos testes de SGBD PostgreSQL, e VoltDBTestController, para a realização de testes de estresse de SGBD VoltDB (figura 8).

Quase a totalidade da dinâmica da metodologia MoDaST se concentra neste módulo, portanto é importante detalhar sua implementação, desde a rotina executada pelo seu construtor, os principais atributos que o compõem, seus métodos internos e o algoritmo executado ao longo de uma iteração de teste.

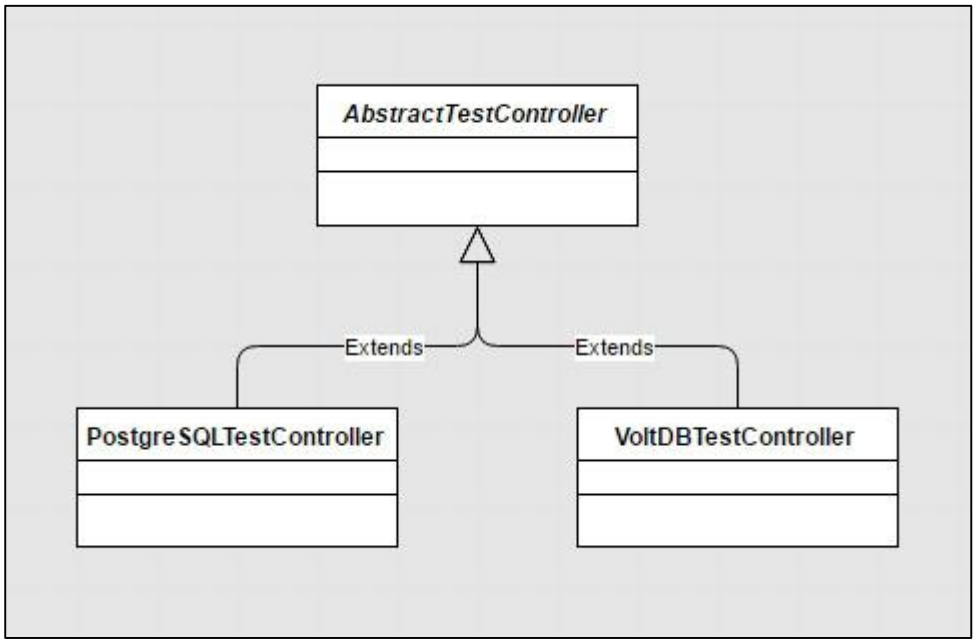


Figura 8 – Diagrama de classe do polimorfismo do módulo controlador de teste

3.3.2.1 Principais Atributos

A implementação do módulo controlador de testes possui dois grupos de atributos bastante importantes para execução dos testes de estresse. Denominamos esses grupos de atributos de configuração e atributos de controle.

Os atributos de configuração armazenam valores que serão utilizados para a comunicação e conexão com o sistema de banco de dados a ser testado, sendo eles:

- databaseAddress: o endereço IP do SGBD a ser testado;
- databasePort: a porta configurada para conexão com o SGBD;
- databaseName: o nome da base de dados a ser testada;
- username: o nome de usuário com privilégios de conexão à base de dados;
- password: a senha do usuário.

Todos estes valores são informados pelo usuário no momento da execução da aplicação e os atributos são inicializados no momento em que o módulo principal configura o controlador de teste, logo após instanciá-lo.

Os atributos de controle são responsáveis por gerenciar todas as etapas do processo de teste de estresse de um SGBD. Alguns deles representam componentes que serão detalhados nos próximos itens. São eles:

- testCase: o caso de teste escolhido pelo usuário para ser executado. O caso de teste é recebido por parâmetro e definido no construtor do módulo. Os possíveis casos de teste estão definidos no enumerador TestCaseEnum, sendo eles CONNECTIONS_STRESS_TESTCASE, teste de estresse sobre o número de conexões concorrentes suportadas pelo banco de dados, e TRANSACTIONS_STRESS_TESTCASE, teste de estresse sobre o número de transações concorrentes suportadas pelo banco de dados. Na hipótese de nenhum caso de teste ser escolhido ao chamar o construtor do módulo controlador de teste, o módulo será inicializado com o caso de teste padrão, que no caso é o teste de estresse sobre o número de conexões;
- testPlan: o plano de teste a ser executado durante o processo de teste. O plano de teste descreve ao JMeter os passos a serem executados quando o teste é iniciado, como por exemplo a carga de trabalho a ser submetida ao sistema de banco de dados. Este componente será melhor detalhado nos próximos itens;
- mainTestClient: a instância do cliente de teste JMeter local;
- mapRemoteTestClients: coleção dos clientes de teste JMeter remotos, indexados por seu endereço IP;
- testStep: contador das iterações ou passos de teste. Determina a iteração de teste em execução;
- modastStateMachine: a máquina de estados, um dos principais requisitos da metodologia MoDaST, que deve refletir os estados internos do sistema de banco de dados em teste. A máquina de estados será detalhada nos próximos itens;
- monitorLogResults: monitora e grava em arquivos os resultados de cada iteração de teste durante sua execução.

Pertence ao módulo avaliador de resultados, que será detalhado nos próximos itens;

- testResultParser: processa e analisa as amostras dos resultados de teste gravadas em arquivo pelo monitorLogResults, convertendo os dados para valores que serão armazenados pelo testResultAggregator. Pertence ao módulo avaliador de resultados, que será detalhado nos próximos itens;
- testResultAggregator: responsável por agrupar os resultados de todas as iterações de teste. Assim como os dois atributos anteriores, também pertence ao módulo avaliador de resultados.

3.3.2.2 Construtor do Módulo Controlador de Teste

A classe base do controlador de teste, AbstractTestController, estabelece uma rotina obrigatória de inicialização para todas as subclasses que dela derivam. Essa rotina está declarada no construtor da superclasse AbstractTestController e apresenta a seguinte sequência de instruções:

- createStateMachine(): cria e inicializa a máquina de estados modastStateMachine;
- initTestStep(): inicializa o contador de iterações de teste testStep e o configura para o estado inicial;
- createMonitorLogResults(): cria o monitor dos resultados de teste monitorLogResults;
- createTestResultAggregator(): cria o componente responsável por agrupar os resultados das iterações de teste testResultAggregator.

A execução dessa rotina deve garantir que o controlador de teste atenda os requisitos mínimos para poder receber os clientes de teste e realizar os testes de estresse de acordo com a metodologia MoDaST.

3.3.2.3 Algoritmo de Execução de Teste

O método executeTest() da classe AbstractTestController define o algoritmo obrigatório para a execução de uma iteração de teste. Este método é composto por uma série de chamadas de métodos abstratos que podem ser implementados livremente de acordo com os propósitos da subclasse a ser desenvolvida, desde que a ordem de chamada dos métodos permaneça intacta. Este método não deve ser sobrescrito.

O algoritmo para a execução de uma iteração de teste, representado pelo fluxograma do anexo VI, é assim definido:

- configureTestPlan(): configura o plano de teste a ser executado pelos clientes de teste JMeter na iteração atual. Como iremos ver no item dedicado ao plano de testes e comentado anteriormente, a carga de trabalho a ser submetida ao SGBD é aumentada a cada iteração. Por esse motivo, o plano de teste deve ser atualizado a cada início de uma iteração de teste;
- configureMonitorLogResults(): configura o monitor de resultados de forma que os dados por ele obtidos sejam identificados como originados da iteração de teste atual, distinguindo-os dos resultados das iterações anteriores;
- testStartTime: armazena o momento exato do início do teste;
- runTestClients(): requisita a todos os clientes de teste JMeter (local e remotos) que dêem início à iteração de teste utilizando o plano de teste acima configurado. Neste ponto foram encontradas algumas dificuldades para identificar o momento em que todos os clientes concluíram suas tarefas de teste. O método isTestRunning() foi criado para auxiliar a identificar que o teste foi concluído. Esta e outras dificuldades encontradas durante o desenvolvimento da aplicação de integração serão detalhadas na seção sobre os problemas encontrados e soluções adotadas;
- testEndTime: armazena o momento exato da conclusão do teste;

- processTestResults(): os resultados monitorados e gravados em arquivo pelo monitor de resultados são processados e convertidos em valores a serem avaliados e armazenados;
- evaluateTestResults(testStartTime, testEndTime): os resultados são avaliados de acordo com as definições da metodologia MoDaST como as variáveis de desempenho, nas quais se baseiam as tomadas de decisão quanto ao estado interno do sistema de banco de dados. Após a avaliação os resultados são armazenados;
- checkStateTransition(): verifica se houve transição de estado interno do SGBD de acordo com os resultados obtidos;
- nextTestStep(): configura o controlador de teste para executar a próxima iteração do processo de teste de estresse.

3.3.3 Máquina de Estados (*Database State Machine*)

Assim como o módulo controlador de teste, que implementa a dinâmica da metodologia MoDaST, a máquina de estados é um componente essencial para a implementação dessa metodologia. Como visto anteriormente, toda metodologia se baseia em inferir os estados internos do SGBD submetendo a ele diferentes níveis de carga de trabalho.

A variação da carga de trabalho, que é aumentada gradualmente a cada iteração de teste, é gerenciada pelo módulo de controle de teste. Já a representação dos estados internos do SGBD fica então a cargo da máquina de estados. A implementação dessa máquina de estados é bastante simples e é definida pela classe ModastStateMachine. Ela é basicamente composta por duas pilhas, stackStateTransitions e stackTimestampTransitions, as quais armazenam as transições de estado e o momento em que ocorreu a transição, respectivamente, além de um atributo state que indica o estado atual em que se encontra a máquina.

Os possíveis estados são definidos pelo enumerador ModastStateEnum e representam os 5 estados internos especificados pela metodologia que um SGBD pode apresentar (WARMUP_STATE,

STEADY_STATE, UNDER_PRESSURE_STATE, STRESS_STATE, THRASHING_STATE) [9].

Por fim, a implementação da máquina de estados disponibiliza métodos para alterar seu estado ou obter o estado atual. Dentro do módulo da máquina de estados, além da implementação da máquina em si, temos também a implementação de um avaliador de transição de estados a partir das variáveis de entrada, definido pela classe ModastStateEvaluator.

Este componente é responsável por verificar as variáveis de entrada (variação de desempenho, vazão de transações e tendência de desempenho) ao fim de cada iteração de teste e avaliar se houve uma transição de estado no SGBD, com base no seu estado atual, comparando o valor das variáveis com os limiares definidos na metodologia e declarados como constantes no avaliador de estado: WARMUP_THRESHOLD, STEADY_THRESHOLD, STRESS_THRESHOLD e THRASHING_THRESHOLD.

3.3.4 Plano de Teste

O plano de teste é o componente básico do JMeter, o qual descreve os passos que a ferramenta deve executar ao iniciar o teste [10][26]. Trata-se de uma estrutura em árvore na qual seus elementos são os componentes de teste disponibilizados pelo JMeter, como grupos de usuários, requisições de serviços (HTTP, FTP, etc.), relatórios, entre outros.

Os planos de teste podem ser criados através da *factory* ModastTestPlanFactory. O método createPostgreSQLModastTestPlan() cria um plano de teste para o SGBD PostgreSQL, enquanto o método createVoltDBModastTestPlan() entrega um plano de teste para o SGBD VoltDB.

Para atender as necessidades de nossa aplicação, criamos uma estrutura própria definida pela classe ModastTestPlan, a qual herda da classe HashTree, que define a estrutura de dados utilizada pelo JMeter para a criação de seus planos de teste. Como o plano de teste deve ser reconfigurado a cada nova iteração de teste, precisamos de uma maior independência e acessibilidade aos componentes que fazem parte do plano de teste.

O JMeter oferece uma infinidade de componentes que podem ser usados para as mais diversas finalidades e objetivos de testes. No entanto, vamos descrever apenas os componentes utilizados neste projeto e que fazem parte do plano de teste ModastTestPlan.

O componente de configuração de conexão JDBC (DataSourceElement) estabelece conexão com o SGBD a ser testado. Entre seus valores de conexão estão o endereço IP e URL do SGBD, nome de usuário e senha para acesso ao banco de dados, nome totalmente qualificado da classe do driver JDBC a ser utilizado, além de outras propriedades de gerenciamento de conexão. Para os testes de nossa aplicação dispomos de dois drivers JDBC: org.postgresql.Driver, para testes no SGBD PostgreSQL, e org.voltdb.jdbc.Driver, para testes no SGBD VoltDB.

O grupo de threads (ThreadGroup) é um componente que representa um grupo de usuários virtuais. Basicamente cada usuário corresponde a uma thread que irá executar uma determinada solicitação. Como vimos no capítulo que descreve a metodologia MoDaST, a carga de trabalho varia a cada iteração de teste.

No caso do teste de estresse no módulo de conexões de um SGBD, por exemplo, iniciamos com uma carga de trabalho de 10 requisições de conexão ao SGBD, sendo que cada conexão irá solicitar o processamento de uma transação. Na iteração de número 2, são submetidas 20 conexões, 30 na iteração de número 3 e 10*n solicitações de conexão na iteração “n”. Assim, o número de threads configurado neste componente representa o número de usuários solicitando conexão ao SGBD.

No entanto, é importante salientar que o número de usuários configurado para este elemento define o número de threads que cada um dos clientes de teste Jmeter irá executar, ou seja, se configurarmos o grupo de threads para 10 usuários em um teste distribuído com 3 clientes, serão executadas 30 solicitações de conexão. Não há como configurar o número de threads a serem executadas por cada cliente de forma independente. Essa é uma limitação da ferramenta que certamente causa alguma dificuldade para aplicar a metodologia de maneira estrita, ampliando a carga de trabalho em múltiplos de 10 solicitações de conexão a cada iteração de teste. Na seção

sobre os problemas encontrados e soluções adotadas comentaremos como decidimos contornar essa limitação.

O próximo componente do JMeter utilizado em nosso projeto é o controlador de iterações (LoopController). Trata-se de um componente lógico do JMeter que define o número de vezes que uma mesma thread (ou usuário) deve executar uma determinada solicitação. Este componente é associado ao grupo de threads, trabalhando em conjunto ao número de threads definido.

No caso do teste de estresse no módulo de processamento de transações, a carga de trabalho inicial definida pelo MoDaST é de 10 usuários solicitando o processamento de 10 transações cada. Na iteração de número 2, 10 usuário solicitam 20 transações cada, 30 transações na iteração de número 3 e n*10 transações na iteração “n”.

Como o controlador de iterações está relacionado ao número de usuários configurado no grupo de threads, facilmente se percebe que este componente possui a mesma limitação para corresponder precisamente ao tamanho da carga de trabalho a cada iteração, dependendo do número de clientes de teste em execução.

O *sampler* JDBC (JDBCSampler) é uma categoria de componente que realiza efetivamente o trabalho de teste do JMeter. Os *samplers* representam uma solicitação, que neste caso será uma solicitação JDBC. Esta solicitação será executada por cada thread configurada no grupo de threads, pelo número de vezes definido no controlador de iterações. Logo, em um teste que possui 5 clientes, com o grupo de threads configurado para 10 usuários e o controlador de iterações ajustado para 10 repetições, teremos um total de 500 solicitações realizadas pelo *sampler* JDBC. É neste componente que são configuradas as instruções SQL a serem executadas.

Por último, utilizamos o coletor de resultados (TestResultCollector): este componente é uma implementação própria deste projeto, a qual herda do componente nativo do JMeter ResultCollector. Sua função, basicamente, é a de ouvir os eventos disparados pelo *sampler* JDBC sempre que há uma nova amostra com os resultados da execução de uma solicitação. O coletor de resultados então contabiliza o número de solicitações bem-sucedidas e o número das que resultaram em erro.

3.3.5 Módulo Avaliador de Resultados

Este módulo é responsável por todo processamento, avaliação e armazenamento dos resultados dos testes. Ele é composto por 3 principais componentes que: monitora os resultados durante a execução de uma iteração de teste (MonitorLogResults), processa e analisa as amostras dos resultados (TestResultParser), agrupa os resultados de todo processo de teste (TestResultAggregator).

No entanto, a principal função deste módulo é o cálculo das variáveis de entrada do MoDaST (variação de desempenho, vazão de transações e tendência de desempenho), a fim de determinar o estado interno do SGBD em teste. Para auxiliar nesta tarefa temos a classe utilitária PerformanceInputsCalculator, que disponibiliza métodos com a implementação dos cálculos formalizados no MoDaST para obtenção das variáveis de entrada.

3.3.5.1 Monitor de Resultados (MonitorLogResults)

Este componente foi desenvolvido devido à dificuldade encontrada em identificar o momento em que todos os clientes de teste encerram a execução da iteração de teste.

O monitor faz uma varredura pelos arquivos de log gerados pelo JMeter durante a execução dos testes, identificando o momento em que todos os clientes concluíram a iteração de teste. Para fins de otimização, o monitor de resultados mantém uma lista em memória com todas as amostras lidas dos arquivos de log.

3.3.5.2 Processador e Analisador de Resultados (TestResultParser)

As amostras dos resultados da iteração de teste previamente coletadas dos arquivos de log pelo monitor de resultados estão no formato de string. Este componente tem como função processar cada string, que representa uma transação submetida ao SGBD, separando cada valor e analisando de forma a identificar se é relevante para o contexto de nossa aplicação.

Ao final do processamento deveremos ter o número total de transações requisitadas, o número de transações bem-sucedidas, o número de transações que resultaram em erro, o horário de início da primeira transação requisitada e o horário de conclusão da última transação processada.

3.3.5.3 Agregador de Resultados (TestResultAggregator)

Este componente nada mais é do que uma coleção de listas que armazenam todos os resultados das iterações executadas durante o processo de teste, indexados pela iteração (base 0) em que foram coletados.

Os seguintes resultados são armazenados por esse componente:

- listTestStartTime: reúne o *timestamp* de início das iterações de teste;
- listTestEndTime: reúne o *timestamp* de encerramento das iterações de teste;
- listTestState: lista o estado interno do SGBD em cada iteração do teste;
- listTransactionsRequestedTotal: o total de transações requisitadas em cada iteração;
- listTransactionsTreatedTotal: o total de transações tratadas em cada iteração;
- listTransactionsRequestedRate: taxa de transações requisitadas por segundo em cada iteração;
- listTransactionsTreatedRate: taxa de transações tratadas por segundo em cada iteração;
- listPerformanceVariation: armazena a variação de desempenho em cada iteração do teste;
- listTransactionsThroughput: armazena a vazão de transações em cada iteração;
- listPerformanceTrend: armazena a tendência de desempenho em cada iteração.

3.4 Problemas Encontrados e Soluções Adotadas

Durante o desenvolvimento e implementação da aplicação de integração do MoDaST ao JMeter, confrontamos com algumas limitações da ferramenta e com alguns problemas, os quais serão apresentados nesta seção, bem como os recursos utilizados para contorná-los.

3.4.1 Encerramento dos Testes Pelos Clientes Remotos

Apesar dos clientes de teste JMeter possuírem uma implementação que permite o envio de comandos aos clientes remotos de uma maneira bastante simples, houve uma certa dificuldade em detectar o momento em que os clientes remotos já realizaram o número de requisições que lhes foram delegadas, indicando que sua etapa de teste encerrou. No entanto, conseguimos receber do lado da aplicação de integração todas as amostras de resultados enviadas pelos clientes de teste, as quais são armazenadas em disco, em um arquivo texto.

Com base nisso, foi desenvolvido o monitor de resultados (MonitorLogResults) (seção 3.3.5.1). O propósito deste componente é monitorar e coletar as amostras de resultados enviadas pelos clientes de teste, tendo ao início da iteração de teste a informação do número de amostras requisitadas definido no plano de teste. Assim que o monitor de resultados recebe a resposta para cada amostra requisitada, seja de que ela bem-sucedida ou tenha falhado, o controlador de testes é informado de que os clientes remotos finalizaram suas tarefas e dá sequência à rotina de teste.

Cada amostra de resultado representa a conclusão de uma solicitação de transação feita a um cliente de teste. O número de solicitações é definido no plano de teste a cada nova iteração, de acordo com o tipo de teste de estresse escolhido, sendo então requisitados pelo controlador de teste.

Como visto no capítulo 2, que trata da metodologia MoDaST, se estivermos executando a primeira iteração de um teste de estresse no módulo de conexão do SGBD, utilizando 5 clientes de teste, cada cliente de teste será encarregado de solicitar 2 conexões ao SGBD, requisitando uma transação por conexão, totalizando uma carga de trabalho de 10 conexões concorrentes e, consequentemente, 10 transações também concorrentes (figura 5). O monitor de resultados irá percorrer o arquivo texto com os resultados gerados

pelos clientes de teste até que sejam coletadas as amostras para as 10 transações, independentemente de terem sido processadas ou resultado em erro. Cada amostra é identificada por um rótulo, o qual indica a qual iteração de teste ela pertence, evitando que uma ou mais amostras de uma iteração anterior sejam contabilizadas como pertencentes àquela em execução.

Os resultados de cada iteração são armazenados de acordo com a seguinte estrutura:

<tsi> | <tt> | <rot> | <res> | <lat> | <idl>

Onde:

- tsi: timestamp inicial da solicitação de transação;
- tt: tempo total de execução da transação, desde sua solicitação (em ms);
- rot: rótulo da amostra indicando a iteração a que ela pertence;
- res: resultado da transação (true para bem-sucedida ou false para erro);
- lat: latência da solicitação, ou seja, o tempo entre a execução da solicitação e o momento em que ela efetivamente solicita uma transação ao SGBD (em ms);
- idl: tempo ocioso da solicitação de transação (em ms).

Esses e outros valores são coletados pelos clientes de teste, de acordo com a implementação do próprio JMeter. Em nossa implementação, definimos o formato e o rótulo de cada amostra, além de configurar o arquivo de configuração global do JMeter (anexo I) para determinar quais resultados queremos que sejam armazenados e quais devem ser descartados em cada amostra.

Para fins de otimização, os resultados são armazenados em memória, para que possam ser processados pelos passos posteriores da rotina de teste, sem a necessidade de acessar novamente os arquivos em disco para efetuar os cálculos.

No final, os arquivos com os resultados são armazenados para permitir consulta e análise de todas as amostras requisitadas durante os testes.

3.4.2 Divisão da Carga de Trabalho Total Pelos Clientes de Teste

Na seção 2.5 que descreve a metodologia MoDaST, vimos que a carga de trabalho de ambos os testes de estresse (de conexões e de transações) é calculada a partir de uma progressão aritmética, a qual possui razão igual a 10, tendo seu primeiro termo também igual a 10, para o teste de estresse no módulo de conexão (figura 5), e razão igual a 100 e primeiro termo igual a 100 no caso do teste de estresse no módulo de processamento de transações (figura 6).

Ao configurar o plano de teste com a carga de trabalho a ser executada, utilizamos os componentes nativos do JMeter chamados Grupo de threads e Controlador de iterações. Em nosso contexto, o primeiro componente representa o número de usuários ou conexões que serão solicitadas ao SGBD. O segundo componente representa o número de transações que serão submetidas ao banco de dados por cada usuário (ou conexão).

Nesta etapa temos uma limitação referente à carga de trabalho a ser definida no plano de teste. Um mesmo plano de teste é aplicado a todos os clientes de teste. Em outras palavras, todos os clientes de teste irão realizar o mesmo número de conexões e transações. Não é possível definir valores diferentes para clientes diferentes em uma mesma iteração de teste. Como consequência, não é possível atender com precisão a carga de trabalho definida pela metodologia se essa carga não representar um múltiplo do número de clientes que irão realizar a rotina de testes.

Para contornar esse problema, ao definir a carga de trabalho que será aplicada a cada cliente de teste, utilizamos um arredondamento da carga de trabalho total definida no MoDaST. O menor valor a ser aplicado deve ser 1 (para caso de valores menores que 1). Os possíveis erros de arredondamento não têm impacto direto no resultado dos testes, uma vez que a diferença entre a carga de trabalho esperada e a carga de trabalho a ser utilizada só é

significativa nas iterações iniciais do teste, as quais não tendem a levar o SGBD a um estado de estresse.

Vejamos esse exemplo de carga de trabalho para um teste de estresse no módulo de processamento de transações utilizando 13 clientes de teste (figura 9):

Iteração de teste	Transações/Iteração (esperado)	Transações/Iteração (utilizado)	Diferença
1	100	104	4,00%
2	200	195	-2,50%
3	300	299	-0,33%
4	400	403	0,75%
5	500	494	-1,20%
11	1100	1105	0,45%
12	1200	1196	-0,33%
13	1300	1300	0,00%
14	1400	1404	0,29%
15	1500	1495	-0,33%

Figura 9 – Carga de trabalho de um teste de estresse no módulo de processamento de transações com 13 clientes de teste

Percebe-se que só há uma diferença relevante entre a carga de trabalho estimada pela metodologia MoDaST e a carga de trabalho a ser utilizada no teste nas primeiras iterações. Após poucas iterações do teste, a tendência é que essa diferença se mantenha abaixo de 1%, aproximando-se de zero.

3.4.3 Cálculo da Tendência de Desempenho

A tendência de desempenho, ou φ, uma das variáveis de entrada da metodologia MoDaST, pode ser obtida através de uma função quadrática, com a qual pode-se prever a curva de desempenho do SGBD testado [9]. No entanto, essa função requer três coeficientes que devem ser obtidos através do método dos quadrados mínimos, em um dado período de tempo.

Tivemos alguns problemas para implementar uma forma de obter esses coeficientes sem onerar o desempenho da aplicação com os cálculos que eram necessários e afetar o andamento das iterações de teste. Já havia uma

preocupação com a adoção da solução citada anteriormente, a qual executa acesso ao disco para determinar se os clientes de teste encerraram seus testes, e que poderia gerar um gargalo de desempenho.

Além disso, de acordo com os testes preliminares efetuados, o banco de dados sequer se aproxima do estado de estresse. A tendência de desempenho só se faz útil na metodologia para determinar a transição do estado de estresse para o estado final, ou de thrashing. Ela não é utilizada em nenhuma das outras transições definidas pelo MoDaST. Sendo assim, optamos por considerar o cálculo da tendência de desempenho como um trabalho futuro.

3.5 Requerimentos e Procedimentos para Execução dos Testes

Para execução dos testes de estresse em um SGBD utilizando a metodologia MoDaST, é necessário que o ambiente de testes atenda a alguns requisitos mínimos. Apesar de nossa aplicação ser independente de arquitetura ou sistema operacional, se restringindo apenas às limitações das tecnologias a serem utilizadas, como os requisitos mínimos da Máquina Virtual Java ou do servidor PostgreSQL, por exemplo, a configuração de máquinas descrita por Meira em seu artigo para os testes por ele realizados é bastante robusta e muito distante de nossa possibilidade de infraestrutura. A sua configuração é composta por 13 máquinas com processador Dual Xeon X5675 de 3,07Ghz e 48GB de memória RAM, as quais executam o sistema operacional Debian GNU/Linux e se comunicam por uma rede InfiniBand do tipo QDR, de baixa latência e alta vazão, chegando a velocidades de transmissão de cerca de 40Gb/s [9][27][28].

Para nossos testes utilizamos uma configuração bastante simples de 5 máquinas virtuais, das quais uma é dedicada ao servidores dos SGBDs PostgreSQL e VoltDB, mantendo apenas aquele que será alvo do teste ativado, 3 máquinas como clientes de teste remotos e a última como cliente de teste principal e dedicada a execução da aplicação de integração entre o JMeter e o MoDaST. As configurações dessas máquinas variam de acordo com as suas necessidades, exigindo mais recursos ao servidor SGBD e muito menos àquelas que simplesmente irão rodar o cliente de teste remoto.

De todo modo, a enorme diferença entre a configuração sugerida por Meira e a configuração a qual tínhamos disponibilidade irá representar um impacto crucial nos resultados de nossos testes, sobre o qual iremos comentar na seção dedicada a análise dos resultados.

3.5.1 Configuração Mínima Exigida e Preparação do Ambiente

A configuração mínima exigida varia de acordo com a função de cada máquina envolvida no processo de teste: servidor SGBD PostgreSQL, servidor SGBD VoltDB, cliente de teste remoto, cliente de teste principal e hospedeiro da aplicação de integração JMeter x MoDaST.

Servidor SGBD PostgreSQL:

- PostgreSQL 9.5 [11];
- Sistema operacional compatível com o PostgreSQL;
- Certificar-se de que o servidor pode receber solicitações das outras máquinas envolvidas no teste;
- Criação da base de dados para teste de acordo com a especificação do *benchmarking* TPC-B (anexo II).

Servidor SGBD VoltDB:

- VoltDB 6.3 [12];
- Sistema operacional compatível com o VoltDB;
- Certificar-se de que o servidor pode receber solicitações das outras máquinas envolvidas no teste;
- Criação da base de dados para teste de acordo com a especificação do *benchmarking* TPC-B (anexo II).

Cliente de Teste Remoto:

- Sistema operacional com suporte à máquina virtual Java;
- Máquina virtual Java (Java SE Runtime) 1.7 ou mais recente;
- Descompactar o arquivo AppModastClient em qualquer pasta.

Cliente de Teste Principal e Hospedeiro da Aplicação JMeter x MoDaST:

- Sistema operacional com suporte à máquina virtual Java;
- Máquina virtual Java (Java SE Runtime) 1.7 ou mais recente;
- Descompactar o arquivo AppModastServer em qualquer pasta;
- Certificar-se de que a máquina pode ser comunicar com os clientes de teste remotos (utilizar o comando ping);
- Adicionar os endereços de IP dos clientes remotos ao arquivo de configuração global do JMeter (anexo I).

3.5.2 Execução da Rotina de Testes de Estresse de um SGBD

Esta seção apresenta os passos para iniciar uma rotina de teste de estresse em um SGBD utilizando a aplicação de integração do JMeter à metodologia MoDaST.

Servidor do SGBD a ser testado (PostgreSQL ou VoltDB):

- Certifique-se de que o serviço do SGBD está ativo;
- Certifique-se de que existe um usuário com privilégios de acesso ao banco de dados a ser testado.

Cliente de Teste Remoto:

- Na pasta onde foi descompactado o arquivo AppModastClient, acesse “jmeter/bin”;
- No terminal, execute “jmeter-server” (Linux) ou no prompt de comando, execute “jmeter-server.bat” (Windows).

Cliente de Teste Principal e Hospedeiro da Aplicação JMeter x MoDaST:

- Certifique-se que todos os clientes remotos executaram a aplicação jmeter-server;
- Acesse a pasta onde foi descompactado o arquivo AppModastServer;
- No terminal (Linux) ou no prompt de comando (Windows) execute a linha de comando de acordo como descrito no anexo IV.

Aguarde a finalização da rotina de teste.

Os arquivos com os resultados do teste podem ser encontrados na pasta "logs" (na pasta onde foi descompactado o arquivo AppModastServer), com o nome "test_results_<timestamp do início do teste>.log".

Dentro da pasta "logs" existe ainda a pasta "logs<timestamp do encerramento do teste>", a qual contém todas as amostras de resultados de todas iterações executadas durante o teste, sob o nome "logs<iteração><timestamp de início da iteração>.jtl".

3.6 Resultados Obtidos

Nesta seção apresentamos os resultados obtidos durante alguns testes realizados com a aplicação desenvolvida. Como comentado anteriormente, escolhemos o SGBD convencional PostgreSQL e o SGBD NewSQL VoltDB para os testes, por possuírem licença de software livre e serem dois sistemas bastante conhecidos. Além disso, Meira utiliza estes mesmos SGBD para a realização dos testes descritos em seu artigo [9], o que possibilita estabelecer um comparativo com os resultados obtidos em nossos testes.

Por dispormos de infraestrutura bastante limitada, utilizamos para os testes uma configuração de 5 máquinas virtuais, das quais:

- 1 máquina dedicada aos servidores SGBD PostgreSQL e VoltDB, mantendo em atividade apenas o servidor que irá ser

testado, com o sistema operacional Ubuntu Linux Server 14.04;

- 3 clientes de teste remotos, executando o sistema operacional Linux Mint;
- 1 cliente de teste principal, dedicado à execução da aplicação de integração entre o JMeter e o MoDaST, executando o sistema operacional Ubuntu Linux Desktop 14.04.

Dentre os testes realizados, escolhemos 4 conjuntos de resultados para serem analisados:

- Teste de estresse no módulo de conexões do SGBD PostgreSQL, com 4 clientes de teste e 500 iterações executadas;
- Teste de estresse no módulo de processamento de transações do SGBD PostgreSQL, com 4 clientes de teste e 500 iterações executadas;
- Teste de estresse no módulo de conexões do SGBD VoltDB, com 4 clientes de teste e 500 iterações executadas;
- Teste de estresse no módulo de processamento de transações do SGBD VoltDB, com 4 clientes de teste e 500 iterações executadas.

A base de dados foi restaurada ao seu estado inicial a cada novo teste, apenas com a carga de dados necessária para a execução correta da rotina de testes (anexo II).

Nenhuma outra aplicação foi mantida em execução durante os testes, exceto aquelas necessárias para o funcionamento do sistema.

O arquivo com o resultados final dos testes pode ser encontrados na pasta “logs” (na pasta onde foi descompactado o arquivo AppModastServer), com o nome “test_results_<timestamp do início do teste>.log”.

Os resultados são processados pelo analisador de resultados (TestResultParser) e armazenados pelo agregador de resultados

(TestResultAggregator), sendo então impressos no seguinte formato definido por nossa implementação:

> testStep | startTime | endTime | state | transRqts | transTreated | transRqts/s | transTreated/s | perfVar | transThroughput | perfTrend

Onde:

- testStep: número da iteração de teste executada;
- startTime: timestamp do início da iteração de teste;
- endTime: timestamp da conclusão da iteração de teste;
- state: estado do SGBD ao iniciar a iteração;
- transRqts: número de transações requisitadas;
- transTreated: número de transações processadas;
- transRqts/s: taxa de transações requisitadas por segundo;
- transTreated/s: taxa de transações processadas por segundo;
- perfVar: variação de desempenho;
- transThroughput: vazão de transações;
- perfTrend: tendência de desempenho.

Nos itens a seguir, iremos comentar os resultados obtidos em cada teste realizado.

3.6.1 Teste de Estresse no Módulo de Conexões do SGBD PostgreSQL

- Número de clientes de teste: 4
- Número total de iterações de teste solicitadas: 2000
- Número total de iterações de teste executadas: 1318
- Número total de conexões solicitadas: 8.677.712
- Número total de transações solicitadas: 8.677.712
- Taxa de transações processadas: 99,00%
- Duração do teste: 4h59min47seg

Embora a intenção fosse executar um teste com 2000 iterações, em todas as tentativas a execução é interrompida muito antes de alcançar esta etapa, por escassez de recursos dos clientes de teste.

Neste teste pretendemos verificar o desempenho do módulo de conexões do SGBD PostgreSQL, submetendo a ele uma crescente carga de solicitação de conexões. Para cada conexão obtida, é requisitada uma transação, de maneira a garantir que a conexão foi obtida com sucesso e capaz de processar transações.

Apesar de termos executado praticamente 5 horas de teste, não conseguimos conduzir o SGBD para uma condição de estresse, sequer para um estado sob pressão durante um período de tempo significativo, com exceção de alguns momentos bastante pontuais dos quais falaremos a seguir. Como já comentado, devido à precariedade de nossa infraestrutura para teste, os clientes de teste ficaram sem recursos muito antes de atingir o objetivo do teste, que era o de simular uma condição que colocasse o PostgreSQL em estresse.

O gráfico a seguir (figura 10) mostra a taxa de transações requisitadas e processadas ao longo do teste. O limiar de transição para o estado sob pressão (*under-pressure*) foi definido com o valor de 0,9 [9]. Isto significa que a vazão (δ) deve ficar abaixo desse valor para que o SGBD possa ser considerado sob pressão. Enquanto o sistema processar 90% ou mais das transações requisitadas, admitimos que ele está em um estado estável (*steady*).

Após aproximadamente 1 hora e 40 minutos de teste, o PostgreSQL transitou pela primeira vez para o estado *under-pressure*, repetindo essa transição por outros 5 diferentes momentos, os quais são desprezíveis para a nossa análise. Em todos esses momentos, o PostgreSQL se manteve sob pressão por poucos segundos, imediatamente retornado a um estado estável (*steady*). Em um único momento, o sistema se manteve sob pressão por 7 minutos (entre 3 horas e 17 minutos e 3 horas e 24 minutos de teste). Entretanto, como indica o gráfico, ainda era capaz de processar 86,58% das transações a ele submetidas. Esse valor está muito próximo do limiar definido

(90%), o que indica que o desempenho não foi comprometido de forma relevante.

Entre 4 horas e 23 minutos e 4 horas e 25 minutos de teste, a vazão teve uma queda significativa, processando apenas 57,08% das transações requisitadas. Mesmo assim, o PostgreSQL logo retornou a um estado estável.

Logo, podemos considerar que o PostgreSQL se manteve estável durante todo processo de teste, desconsiderando os momentos de pressão apontados.

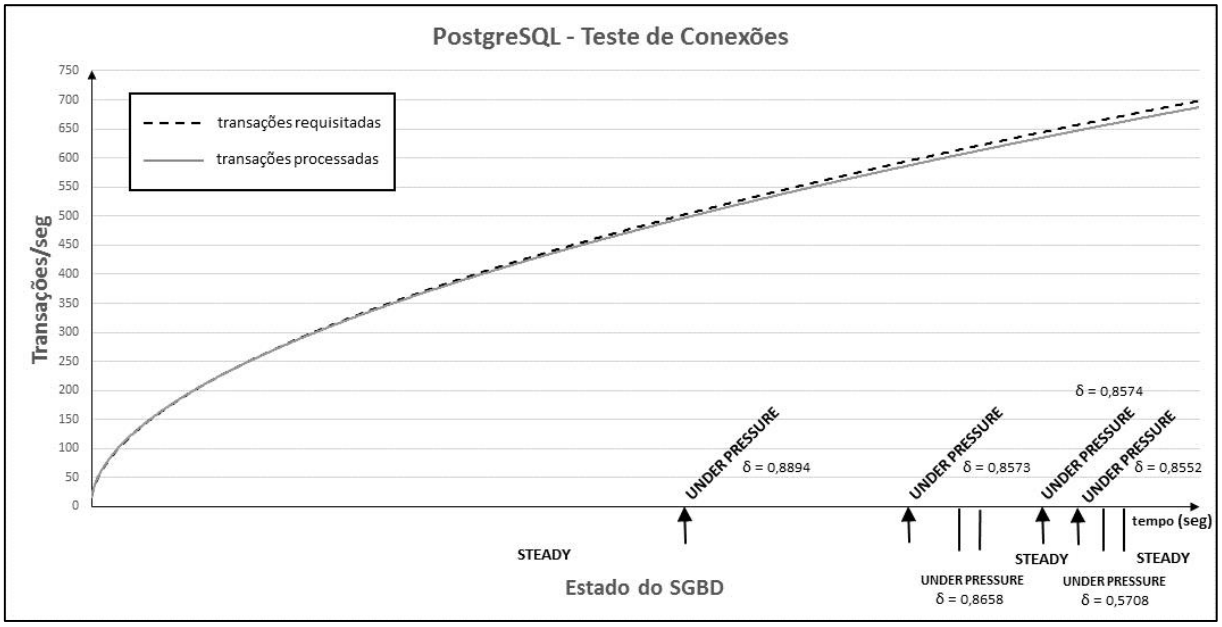


Figura 10 – Desempenho do PostgreSQL em um teste de estresse no módulo de conexões

Ao analisar as linhas de tendência da taxa de transações requisitadas e de transações processadas, pode-se dizer que, a longo prazo, o número de transações que serão processadas pelo SGBD tende a diminuir, o que possivelmente poderia levar o PostgreSQL a um estado de estresse, no entanto, com a submissão de uma carga de trabalho muito maior do que a qual nosso ambiente de teste foi capaz de administrar.

Além da vazão, é necessário fazer uma análise pela perspectiva da variação de desempenho para poder avaliar em que momento o SGBD se encontra em uma condição de estresse. O gráfico a seguir (figura 11) registra os valores para a variação de desempenho, a variável de entrada Δ do MoDaST, ao longo da execução do teste.

De acordo com a metodologia MoDaST, o SGBD é considerado em estado estável enquanto a variação de desempenho tende a zero (Δ→0). De acordo com o gráfico, a variação se mantém estável, crescendo a uma taxa cada vez menor à medida em que o teste avança, mesmo nos momentos em que ocorrem as transições para o estado *under-pressure*. O limiar estabelecido para a transição para o estado de estresse representa 10% da taxa de transações processadas do estado anterior (*under-pressure*) [9], o que evidencia que o PostgreSQL está muito distante de entrar em uma condição de estresse.

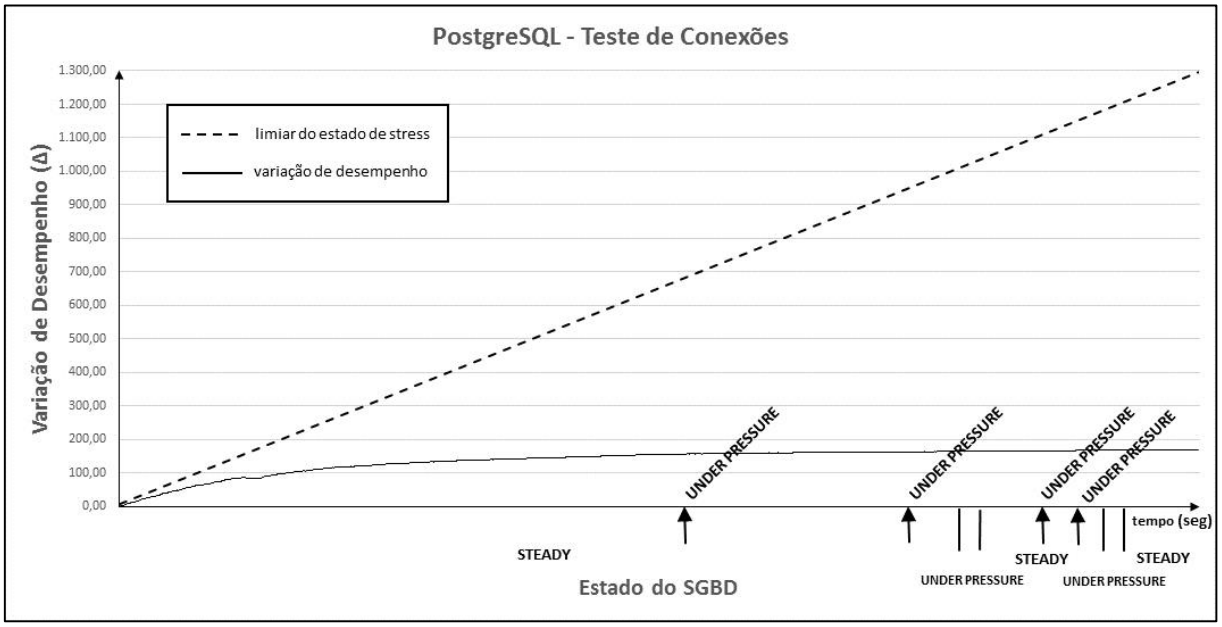


Figura 11 – Variação de Desempenho do PostgreSQL em um teste de estresse no módulo de conexões

Se estabelecermos um comparativo com os resultados obtidos por Meira nos testes apresentados em seu artigo, veremos que em seus resultados foi demonstrado que o PostgreSQL entra em um estado sob pressão à medida em que a carga de trabalho se aproxima de 1000 solicitações de conexões simultâneas. A partir desse momento, o SGBD não consegue administrar nenhuma conexão acima desse valor, consequentemente tendendo para uma condição de estresse à medida que a carga de trabalho é aumentada (figura 12) [9].

Nosso teste foi interrompido à medida em que o número de solicitações de conexões simultâneas se aproximava de uma taxa de 700

requisições por segundo. Os resultados obtidos mostram o SGBD sob pressão em alguns curtos momentos a partir de 600 solicitações de conexões por segundo (figura 10). Até este patamar de carga de conexões solicitadas, os resultados dos testes de Meira mostram que o PostgreSQL se mantém em um estado estável (*steady*) (figura 12).

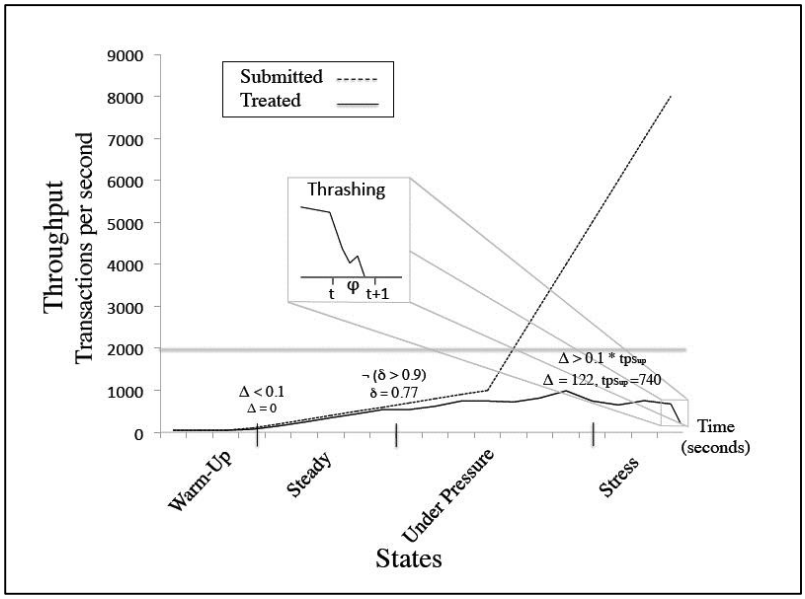


Figura 12 – Desempenho do PostgreSQL sob um teste de estresse no módulo de conexões [9]

Dessa forma, embora não possamos afirmar que nossos testes iriam conduzir o SGBD a um estado de estresse se a rotina de teste não tivesse sido interrompida, é razoável dizer que os resultados de nosso teste, até o ponto onde pode ser executado, estão em conformidade com o que era esperado com a aplicação da metodologia MoDaST.

3.6.2 Teste de Estresse no Módulo de Processamento de Transações do SGBD PostgreSQL

- Número de clientes de teste: 4
- Número total de iterações de teste solicitadas: 2000
- Número total de iterações de teste executadas: 982
- Número total de transações solicitadas: 9.651.096
- Taxa de transações processadas: 100,00%
- Duração do teste: 4h51min50seg

Assim como nos testes no módulo de conexões, não obtivemos sucesso em executar todas as iterações solicitadas. Os recursos de nosso ambiente de teste sempre se esgotam antes de concluir toda tarefa esperada.

Neste teste, o número de conexões se mantém constante, sendo que o número de requisições de transação aumenta a cada iteração. Diferentemente do teste anterior, absolutamente todas as transações foram processadas pelo SGBD. Realizamos outras tentativas esperando resultados diferentes e em todas elas o PostgreSQL foi capaz de tratar 100% das transações a ele submetidas, mantendo-se em estado estável (*steady*) durante todo processo (figura 13).

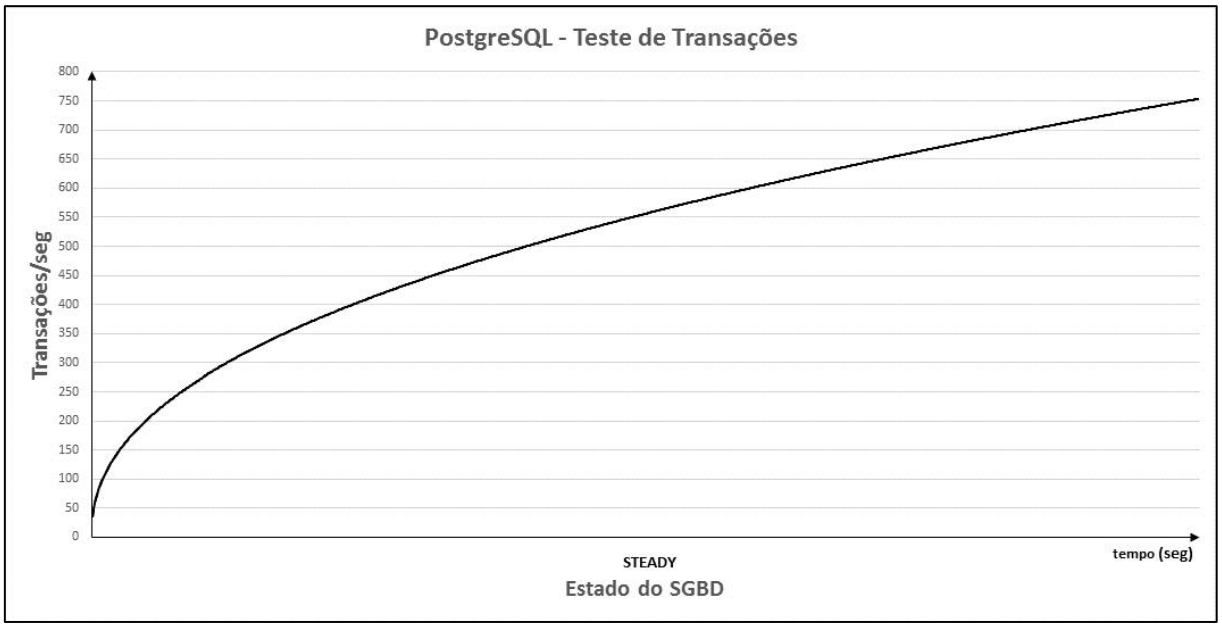


Figura 13- Desempenho do PostgreSQL em um teste de estresse no módulo de processamento de transações

Se compararmos o comportamento da variação de desempenho (Δ) com o teste anterior, teremos resultados muito semelhantes (figura 14). À medida em que o valor da variação de desempenho se aproxima de 200 (Δ→200), essa variação tende a estabilizar, se distanciando cada vez mais do limiar de transição para o estado de estresse. Isso mais uma vez leva a crer que precisamos de um ambiente de teste com recursos para executar os testes durante mais tempo e de forma adequada, para possivelmente conduzir o

PostgreSQL a um estado de estresse, a fim de confirmar a aplicação da metodologia MoDaST.

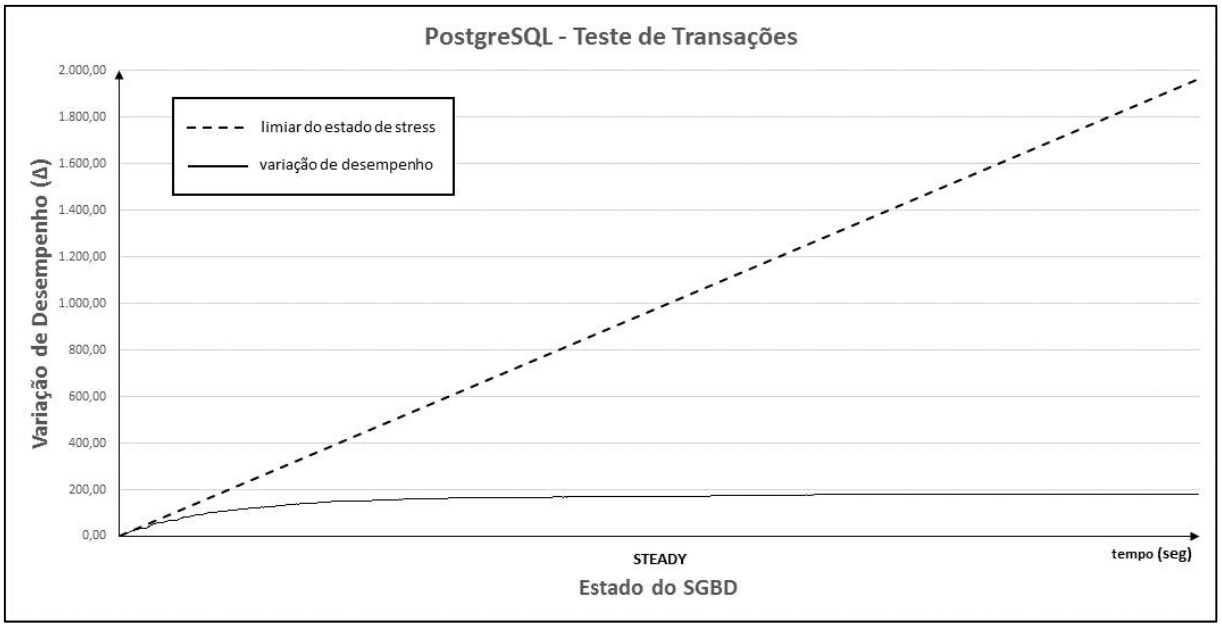


Figura 14 - Variação de Desempenho do PostgreSQL em um teste de estresse no módulo de processamento de transações

3.6.3 Teste de Estresse no Módulo de Conexões do SGBD VoltDB

- Número de clientes de teste: 4
- Número total de iterações de teste solicitadas: 2000
- Número total de iterações de teste executadas: 897
- Número total de conexões solicitadas: 4.026.632
- Número total de transações solicitadas: 4.026.632
- Taxa de transações processadas: 100,00%
- Duração do teste: 2h31min11seg

No testes com o VoltDB, a rotina foi interrompida ainda mais prematuramente do que nos testes executados no SGBD PostgreSQL, provavelmente por ter sido necessário alocar mais recursos do ambiente ao servidor VoltDB, depreciando os clientes de teste.

Os gráficos a seguir mostram o desempenho do VoltDB sob a perspectiva da taxa de transações requisitadas por segundo e de sua variação de desempenho. Nota-se bastante semelhança com os testes realizados com o PostgreSQL (figuras 15 e 16).

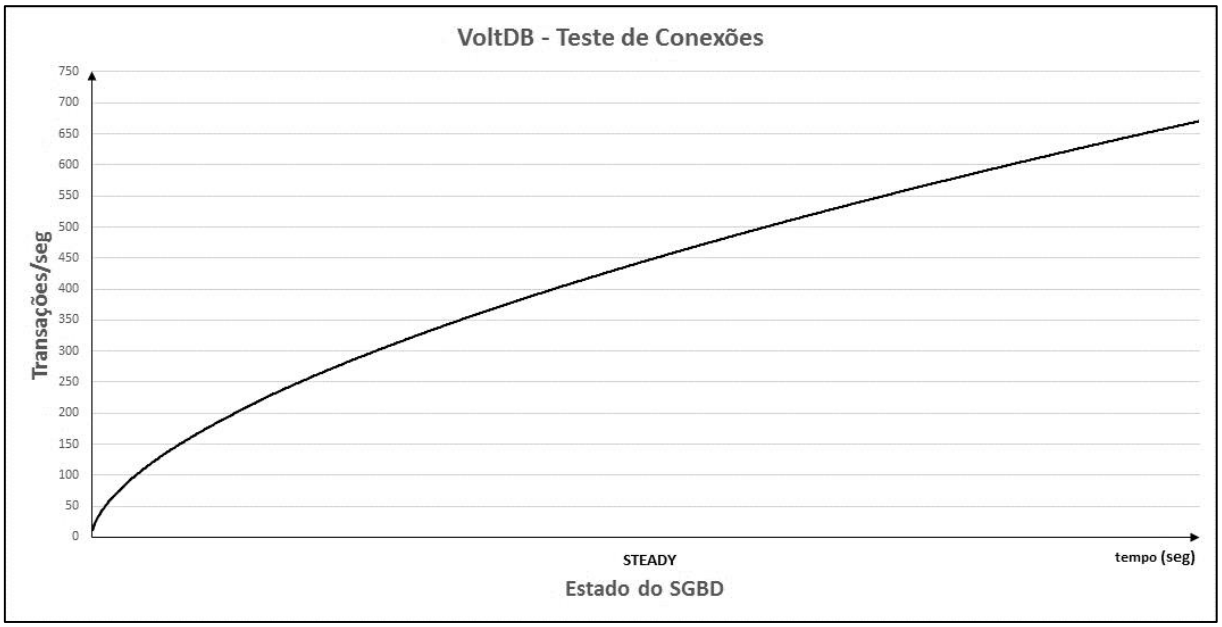


Figura 15 - Desempenho do VoltDB em um teste de estresse no módulo de conexões

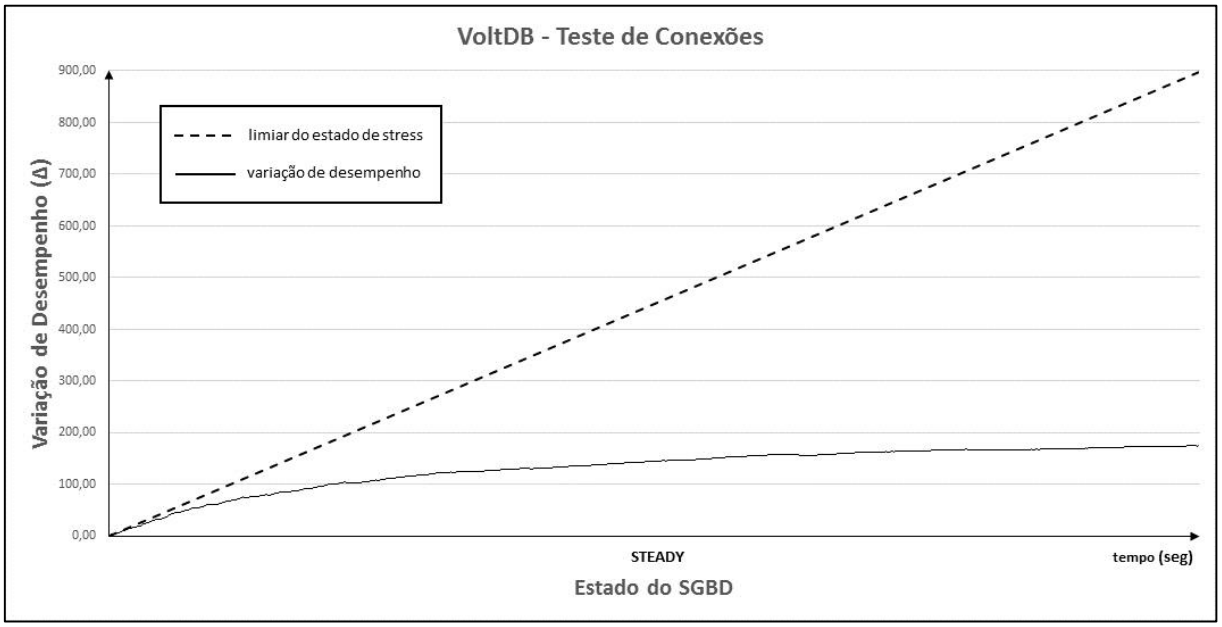


Figura 16 - Variação de Desempenho do VoltDB em um teste de estresse no módulo de conexões

Igualmente concluímos sobre os resultados obtidos com os testes de conexões do PostgreSQL, não podemos afirmar que nossos testes iriam conduzir o SGBD a um estado de estresse se a rotina de teste não tivesse sido interrompida. Nos resultados obtidos nos testes realizados no VoltDB por Meira, o SGBD se mantém basicamente em um estado estável ou, no máximo,

sob pressão, mostrando bastante estabilidade no processamento das solicitações [9].

3.6.4 Teste de Estresse no Módulo de Processamento de Transações do SGBD VoltDB

- Número de clientes de teste: 4
- Número total de iterações de teste solicitadas: 2000
- Número total de iterações de teste executadas: 1742
- Número total de transações solicitadas: 30.359.576
- Taxa de transações processadas: 100,00%
- Duração do teste: 13h50min15seg

De todos os testes realizados, esta foi a execução mais duradoura e que chegou mais próximo do número de iterações pretendido antes de ser interrompido por falta de recursos no ambiente de teste.

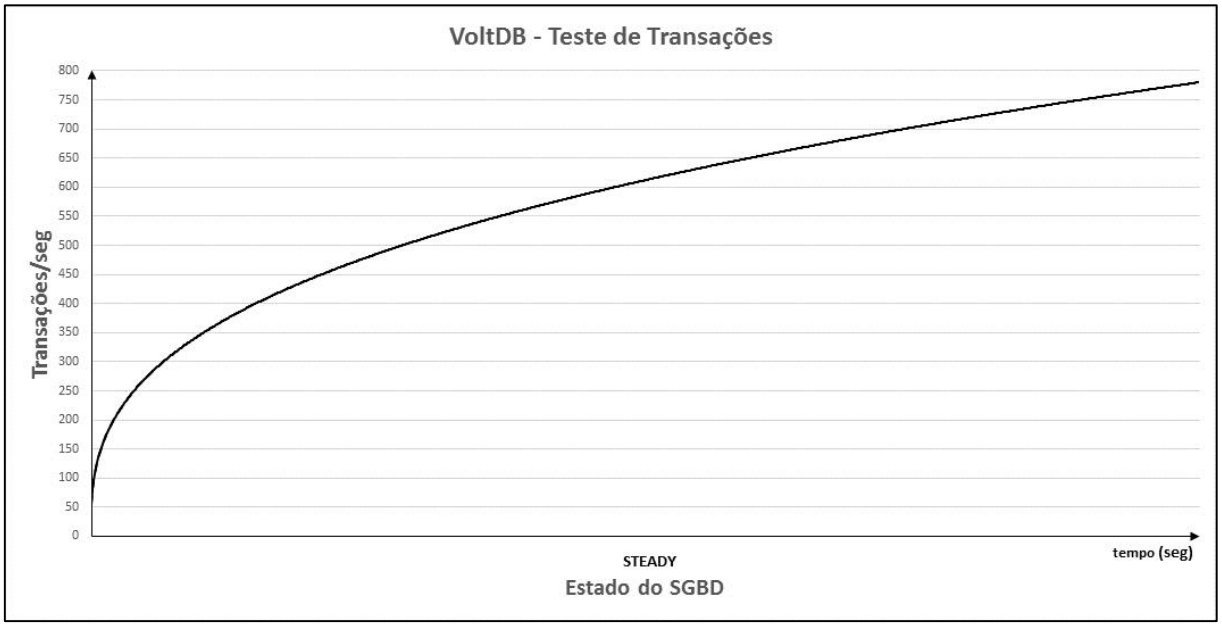


Figura 17 - Desempenho do VoltDB em um teste de estresse no módulo de processamento de transações

O SGBD se manteve estável durante toda execução do teste de estresse. A taxa de solicitações de transações chegou próximo de 800 requisições por segundo (figura 17) e o comportamento da variação de desempenho ao longo do teste também não aponta qualquer evidência de que o SGBD esteja se encaminhando para uma condição de estresse (figura 18).

Além disso, neste teste observamos que a variação de desempenho começa a diminuir a partir de um certo ponto de execução do teste, o que evidencia uma capacidade do VoltDB de manter um desempenho estável mesmo submetido a uma carga de solicitações de transações altamente crescente.

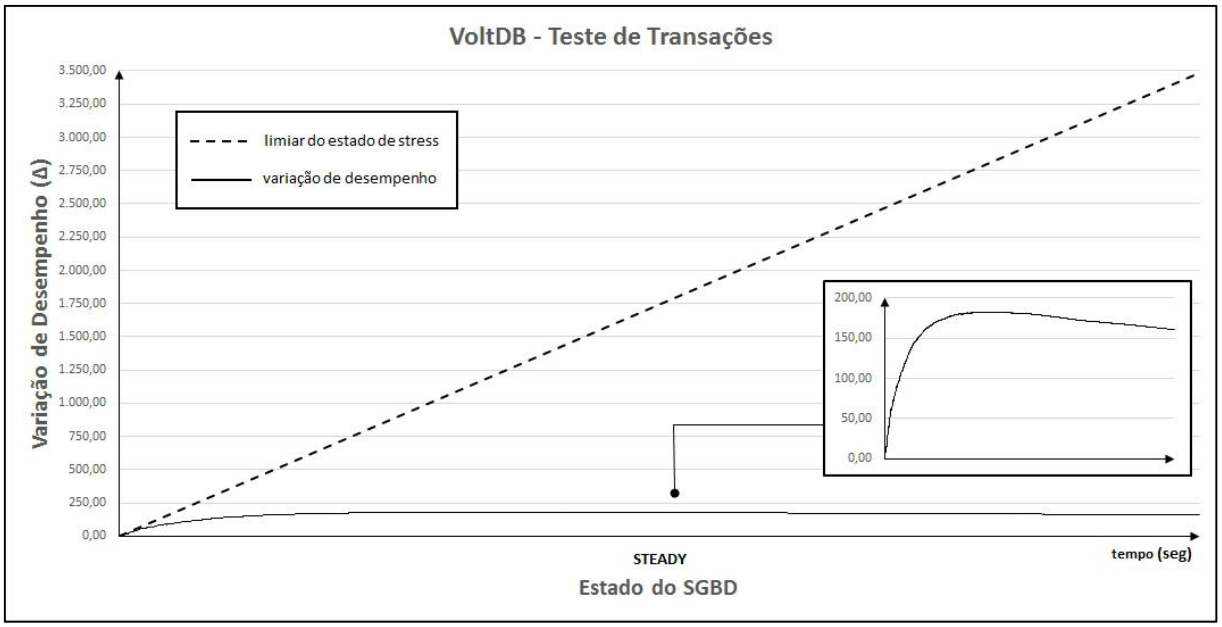


Figura 18 - Variação de Desempenho do VoltDB em um teste de estresse no módulo de conexões

4. CONCLUSÃO E SUGESTÕES PARA TRABALHOS FUTUROS

Apesar de não termos obtido sucesso em conduzir os SGBD testados a uma condição de estresse, julgamos termos atingido satisfatoriamente o objetivo principal desse trabalho, de desenvolver uma aplicação que permita a integração da metodologia MoDaST com a ferramenta de testes Apache JMeter.

As definições da metodologia no que diz respeito à carga de trabalho a ser aplicada ao SGBD a ser testado puderam ser implementadas adequadamente, exceto pelo seu dimensionamento exato, devido a uma limitação da ferramenta JMeter, como comentado anteriormente (seção 3.4.2). De todo modo, essa diferença mostrou não ter impacto relevante na execução e resultados dos testes.

As amostras de resultados de cada etapa dos testes são coletadas como esperado, contendo todos os valores necessários para o cálculo das variáveis de entrada definidas no MoDaST e a correta análise do estado atual do SGBD em teste. Os resultados puderam também ser armazenados em dados puros, que permitem diferentes perspectivas de análise do andamento de toda rotina de teste executada, de acordo com a necessidade.

O fato dos nossos testes não terem sido capazes de levar os sistemas de gerenciamento de banco de dados testados a um estado de estresse muito se deve à precariedade da infraestrutura de testes a qual tínhamos à disposição. Como descrito na seção 3.5, referente ao ambiente adequado para execução dos testes, apenas uma das máquinas utilizadas como cliente de teste e sugeridas no artigo sobre o MoDaST [9] possuía uma configuração além daquela que tínhamos disponível para execução dos testes.

No entanto, os mecanismos que conduzem a máquina de estados de nossa aplicação a refletir o estado de um SGBD em teste puderam ter seu funcionamento verificado, simulando os dados de resposta de um SGBD com os valores esperados pelo MoDaST, com o propósito de extrapolar os limiares que definem as possíveis transições de estado de um SGBD e definidos na metodologia.

Devemos considerar que os SGBD testados prometem robustez e alta disponibilidade de seus sistemas, devendo permanecerem estáveis e em

operação mesmo quando sob uma enorme demanda de requisições. Sendo assim, apenas com uma infraestrutura adequada, que possa disponibilizar um número considerável de clientes de teste que tenham capacidade para processar um grande volume de requisições de transações, simulando um cenário real de sobrecarga de demanda de transações e de tráfego de rede, é que poderíamos obter resultados equivalentes aos obtidos por Meira [9] e que teriam sucesso em conduzir os sistemas testados a um estado de estresse ou mesmo de degradação.

Por fim, este trabalho foi de grande utilidade para aprofundar no conhecimento do funcionamento dos SGBD NewSQL e nas possibilidades da ferramenta Apache JMeter para a execução dos mais variados tipos de testes. Sugerimos para um trabalho futuro a correção das deficiências existentes na aplicação devido às limitações encontradas durante o desenvolvimento e apontadas neste trabalho. Sugerimos também melhorias que permitam modificações dinâmicas durante o processo de teste, como alteração da carga de trabalho, inclusão ou remoção de clientes de teste e alteração da transação a ser submetida ao SGBD. Outra possível melhoria para um trabalho futuro é a implementação desta aplicação para testes de estresse em SGBD em nuvem, utilizando a metodologia MoDaST, como sugerido por Meira na conclusão de seu artigo, com a inclusão de estados adicionais à máquina de estado [9].

A aplicação foi desenhada com a intenção de oferecer versatilidade para modificá-la e estendê-la a toda gama de serviços e protocolos que podem ser testados pelo JMeter. Acreditamos que assim esse projeto possa contribuir para a elaboração de possíveis outros trabalhos que pretendam utilizar a metodologia MoDaST ou mesmo investigar outras eventuais metodologias para testes em sistemas de bancos de dados NewSQL, assim como também para aqueles que destinam-se a criar novas aplicações para a ferramenta JMeter, seja para testar sistemas de banco de dados ou mesmo outros tipos de serviços e tecnologias.

ANEXO I - Arquivo de Configuração Global do JMeter

O arquivo de configuração global do JMeter (jmeter.properties) apresenta diversos parâmetros de configuração para a inicialização adequada da ferramenta, bem como para a personalização de seu funcionamento de acordo com as necessidades do usuário.

Neste anexo vamos apresentar apenas as configurações que foram alteradas para atender as necessidades de nossa aplicação.

O arquivo pode ser encontrado na pasta onde foi descompactado o arquivo AppModastServer, dentro da pasta "jmeter/bin", pelo nome de "jmeter.properties".

Apesar dos clientes de teste remotos também possuírem esse arquivo dentro da pasta de mesmo nome, apenas as mudanças no arquivo da máquina de teste principal serão efetivadas.

- Remote hosts and RMI configuration:

O parâmetro "remote_hosts" deve ser configurado com o endereço de IP de todos os clientes envolvidos no teste, inclusive o cliente local (localhost ou endereço de loopback 127.0.0.1). Os endereços devem ser separados por vírgula. Não há necessidade de declarar o número da porta.

```
remote_hosts=localhost,192.168.200.130,192.168.200.131,192.168.200.132
```

- Results file configuration:

Os parâmetros desta seção definem a forma como e quais os resultados que serão gravados em disco.

```
# assertion_results_failure_message only affects CSV output
jmeter.save.saveservice.assertion_results_failure_message=false
```

As mensagens de erro devem ser impressas em todos os formatos de saída, não apenas no formato CSV.

```
jmeter.save.saveservice.data_type=false
jmeter.save.saveservice.response_code=false
jmeter.save.saveservice.response_data=false
```

```
jmeter.save.saveservice.response_data.on_error=false
jmeter.save.saveservice.response_message=false
jmeter.save.saveservice.thread_name=false
jmeter.save.saveservice.samplerData=false
jmeter.save.saveservice.responseHeaders=false
jmeter.save.saveservice.requestHeaders=false
jmeter.save.saveservice.encoding=false
jmeter.save.saveservice.bytes=false
jmeter.save.saveservice.url=false
jmeter.save.saveservice.filename=false
jmeter.save.saveservice.thread_counts=false
```

Os valores acima foram configurados para não serem impressos entre os resultados.

```
jmeter.save.saveservice.default_delimiter=|
```

Define o caractere delimitador dos valores a serem impressos.

```
jmeter.save.saveservice.print_field_names=false
```

Não queremos imprimir o cabeçalho com os nomes dos campos.

- Settings that affect SampleResults:

Configurações referentes às amostras de resultados. Configuramos para as amostras registrarem o seu timestamp utilizando a função System.currentTimeMillis() ao invés da função System.nanoTime().

```
sampleresult.useNanoTime=false
```

- Remote batching configuration:

Configura os clientes de teste para reter as amostras de resultado até a finalização da sua iteração de teste. Isto evita tráfego desnecessário na rede durante a execução dos testes.

```
mode=Hold
```

ANEXO II - Esquema da Base de Dados

O esquema da base de dados definido pelo *benchmarking* TPC-B representa uma pequena instituição bancária, com uma ou mais agências, vaŕios caixas e inúmeros clientes [13][30].

O script aqui exposto está simplificado e pode precisar de ajustes de acordo com o SGBD no qual será aplicado.

Logo em seguida será apresentado um script de carga inicial que pode ser executado via shell script, console SQL, ou qualquer outra ferramenta ou interface preferida pelo usuário.

- Script para criação de esquema da base de dados:

```
CREATE DATABASE <nome_da_base_de_dados>
      OWNER=<nome_do_usuário>;

CREATE TABLE branches
(
   "bid" numeric(9),
   "bbalance" numeric(10),
   filler char(88)
);
ALTER TABLE branches OWNER TO <nome_do_usuário>;

CREATE TABLE tellers
(
   "tid" numeric(9),
   "bid" numeric(9),
   "tbalance" numeric(10),
   filler character(84)
);
ALTER TABLE tellers OWNER TO <nome_do_usuário>;

CREATE TABLE accounts
(
   "aid" numeric(9),
   "bid" numeric(9),
   "abalance" numeric(10),
   filler character(84)
);
ALTER TABLE accounts OWNER TO <nome_do_usuário>;

CREATE TABLE history
(
   "tid" numeric(9),
   "bid" numeric(9),
   "aid" numeric(9),
   "delta" numeric(10),
   "time" timestamp,
   filler character(22)
```

```
);
ALTER TABLE history OWNER TO <nome_do_usuário>;

ALTER TABLE ONLY branches
    ADD CONSTRAINT pk_bid PRIMARY KEY ("bid");
ALTER TABLE ONLY tellers
    ADD CONSTRAINT pk_tid PRIMARY KEY ("tid");
ALTER TABLE ONLY accounts
    ADD CONSTRAINT pk_aid PRIMARY KEY ("aid");

ALTER TABLE ONLY tellers
    ADD CONSTRAINT fk_bid  FOREIGN KEY ("bid") REFERENCES branches("bid");
ALTER TABLE ONLY accounts
    ADD CONSTRAINT fk_bid  FOREIGN KEY ("bid") REFERENCES branches("bid");
ALTER TABLE ONLY history
    ADD CONSTRAINT fk_tid  FOREIGN KEY ("tid") REFERENCES tellers("tid"),
    ADD CONSTRAINT fk_bid  FOREIGN KEY ("bid") REFERENCES branches("bid"),
    ADD CONSTRAINT fk_aid  FOREIGN KEY ("aid") REFERENCES accounts("aid");
```

- Script de carga inicial da base de dados:

```
INSERT INTO branches(bid,bbalance) VALUES (1,0);
INSERT INTO tellers(tid,bid,tbalance) VALUES (generate_series(1,10),1,0);
INSERT INTO accounts(aid,bid,abalance) VALUES (generate_series(1,100000),1,0);
```

ANEXO III – Definição da Transação

A transação definida pelo *benchmarking* TPC-B simula uma transação bancária usual, na qual um cliente faz um depósito ou uma retirada de qualquer valor de sua conta. Então o balanço do caixa em que foi feita a movimentação financeira deve ser atualizado, bem como o balanço geral da agência a qual pertence a conta do cliente. Por fim, é criado um registro com o histórico da movimentação realizada [13][30].

As instruções SQL a serem executadas pelas transação são:

```
BEGIN TRANSACTION;

UPDATE accounts SET abalance = abalance + $delta WHERE aid = $aid;

SELECT abalance FROM accounts WHERE aid = $aid;

UPDATE tellers SET tbalance = tbalance + $delta WHERE tid = $tid;

UPDATE branches SET bbalance = bbalance + $delta WHERE bid = $bid;

INSERT INTO history(tid, bid, aid, delta, time) VALUES ($tid, $bid, $aid, $delta,
    CURRENT_TIMESTAMP);

COMMIT TRANSACTION;
```

Onde $bid, $tid, $aid e $delta são valores de entrada gerados pela aplicação de integração MoDaST x JMeter.

ANEXO IV – Linha de Comando para Execução da Aplicação de Integração e Cliente de Teste Principal

Para execução da aplicação de integração do JMeter ao MoDaST e do cliente de teste principal, execute a seguinte linha de comando no terminal (Linux) ou prompt de comando (Windows):

java -jar testmodast -tc [c|t] -tp [volt|post] -ip dbIpAdress -p dbPort -db databaseName -u username -pwd password -s testStepsTotal –Xms heapSize –Xmx heapSize

Onde:

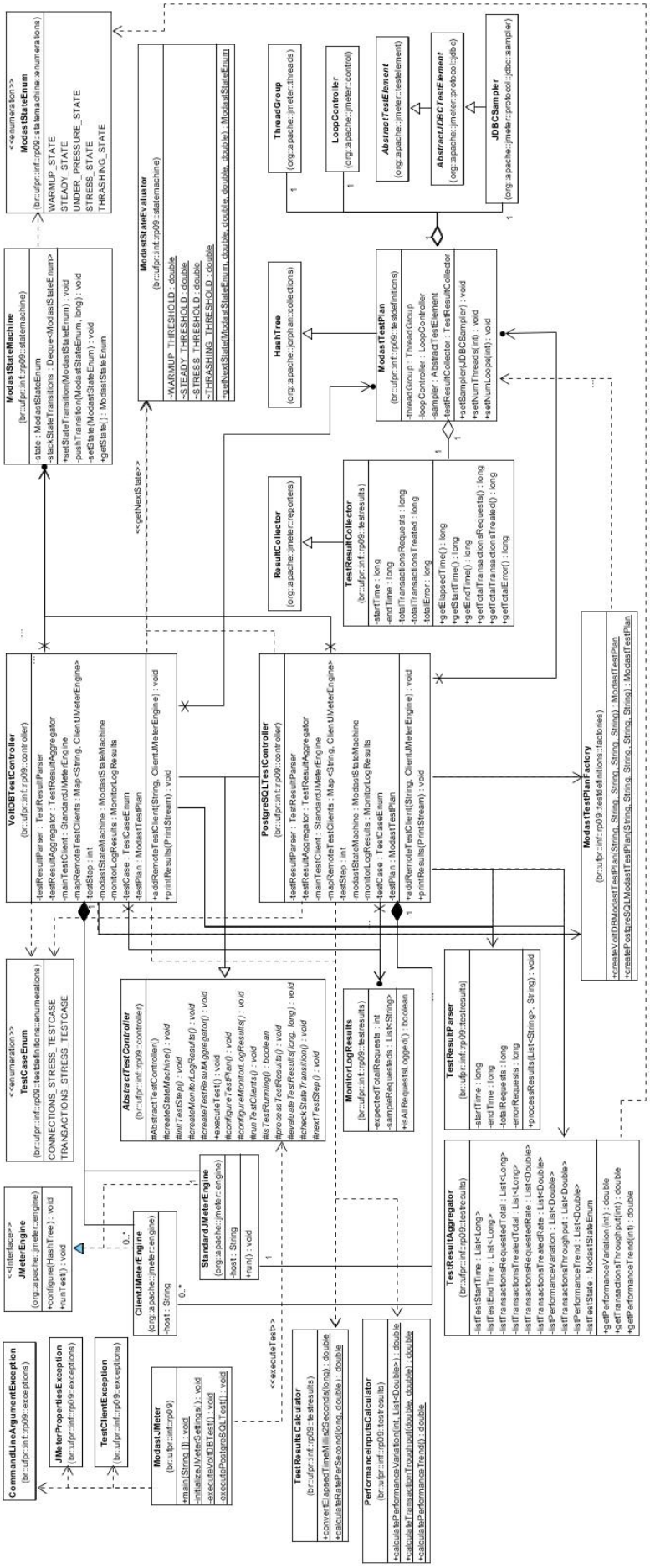
- -tc: c (teste de estresse no módulo de conexões) ou t (teste de estresse no módulo de processamento de transações);
- -tp: volt (teste do SGBD VoltDB) ou post (teste do SGBD PostgreSQL);
- -ip: endereço IP do servidor SGBD;
- -p: porta utilizada pelo SGBD;
- -db: nome da base de dados a ser testada;
- -u: nome do usuário com privilégios de acesso à base de dados;
- -pwd: senha do usuário;
- -s: número máximo de iterações de teste a serem executadas;
- -Xms: quantidade inicial de memória alocada para a heap (argumento opcional);
- -Xmx: quantidade máxima de memória alocada para a heap. Sugerimos ao menos 4GB (argumento opcional).

ANEXO V – Diagrama de Classes

O seguinte diagrama exibe as classes implementadas e utilizadas neste projeto e seus relacionamentos.

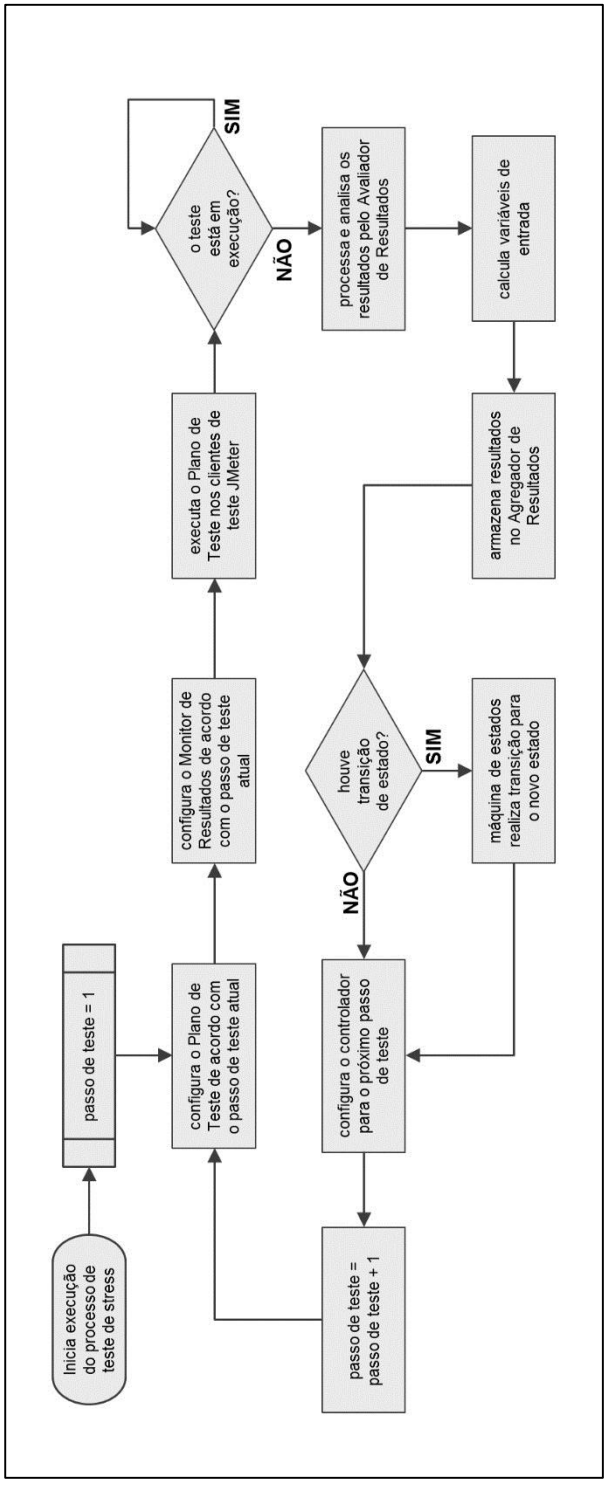
As classes pertencentes ao pacote “br.ufpr.inf.rp09” foram desenvolvidas e implementadas exclusivamente para esse projeto. As classes pertencentes “org.apache.jorphan” e “org.apache.jmeter” são implementações disponibilizadas pela ferramenta Apache JMeter.

Vários atributos e métodos foram omitidos neste diagrama, visando explicitar apenas os elementos básicos para entendimento da aplicação e seus relacionamentos.


CommandLineArgumentException
(br::ufpr::inf::rp09::exceptions)
JMeterPropertieeException
(br::ufpr::inf::rp09::exceptions)
TestClientException
(br::ufpr::inf::rp09::exceptions)
ModastJMeter
(br::ufpr::inf::rp09)
+main(String []) : void
-initializeJMeterSettings() : void
-executeVoltDBTest() : void
-executePostgreSQLTest() : void
<<Interface>>
JMeterEngine
(org::apache::jmeter::engine)
+configure(HashTree) : void
+runTest() : void
ClientJMeterEngine
(org::apache::jmeter::engine)
-host : String
StandardJMeterEngine
(org::apache::jmeter::engine)
-host : String
+run() : void
<<executeTest>>
TestResultsCalculator
(br::ufpr::inf::rp09::testresults)
+convertElapsedTimeMiliis2Seconds(long) : double
+calculateRatePerSecond(long, double) : double
PerformanceInputsCalculator
(br::ufpr::inf::rp09::testresults)
+calculateRatePerSecond(long, double) : double
+calculateTransactionThroughput(double, double) : double
+calculatePerformanceTrend() : double
<<enumeration>>
TestCaseEnum
(br::ufpr::inf::rp09::testdefinitions::enumerations)
CONNECTIONS_STRESS_TESTCASE
TRANSACTIONS_STRESS_TESTCASE
AbstractTestController
(br::ufpr::inf::rp09::controller)
#AbstractTestController()
#createStateMachine() : void
#initTestStep() : void
#createMonitorLogResults() : void
#createTestResultAggregator() : void
+executeTest() : void
#configureTestPlan() : void
#configureMonitorLogResults() : void
#runTestClients() : void
#isTestRunning() : boolean
#processTestResults() : void
#evaluateTestResults(long, long) : void
#checkStateTransition() : void
#nextTestStep() : void
MonitorLogResults
(br::ufpr::inf::rp09::testresults)
-expectedTotalRequests : int
-samplerRequestsIds : List<String>
+allRequestsLogged() : boolean
TestResultAggregator
(br::ufpr::inf::rp09::testresults)
-listTestStartTime : List<Long>
-listTestEndTime : List<Long>
-listTransactionsRequestedTotal : List<Long>
-listTransactionsTreatedTotal : List<Long>
-listTransactionsRequestedRate : List<Double>
-listTransactionsTreatedRate : List<Double>
-listPerformanceVariation : List<Double>
-listTransactionsThroughput : List<Double>
-listPerformanceTrend : List<Double>
-listTestState : ModastStateEnum
+getPerformanceVariation(int) : double
+getTransactionsThroughput(int) : double
+getPerformanceTrend(int) : double
TestResultParser
(br::ufpr::inf::rp09::testresults)
-startTime : long
-endTime : long
-totalRequests : long
-errorRequests : long
+processResults(List<String>, String) : void
VoltDBTestController
(br::ufpr::inf::rp09::controller)
-testResultParser : TestResultParser
-testResultAggregator : TestResultAggregator
-mainTestClient : StandardJMeterEngine
-mapRemoteTestClients : Map<String, ClientJMeterEngine>
-testStep : int
-modastStateMachine : ModastStateMachine
-monitorLogResults : MonitorLogResults
-testCase : TestCaseEnum
-testPlan : ModastTestPlan
+addRemoteTestClient(String, ClientJMeterEngine) : void
+printResults(PrintStream) : void
PostgreSQLTestController
(br::ufpr::inf::rp09::controller)
-testResultParser : TestResultParser
-testResultAggregator : TestResultAggregator
-mainTestClient : StandardJMeterEngine
-mapRemoteTestClients : Map<String, ClientJMeterEngine>
-testStep : int
-modastStateMachine : ModastStateMachine
-monitorLogResults : MonitorLogResults
-testCase : TestCaseEnum
-testPlan : ModastTestPlan
+addRemoteTestClient(String, ClientJMeterEngine) : void
+printResults(PrintStream) : void
<<getNextState>>
ModastStateMachine
(br::ufpr::inf::rp09::statemachine)
-state : ModastStateEnum
-stackStateTransitions : Deque<ModastStateEnum>
+setStateTransition(ModastStateEnum) : void
-pushTransition(ModastStateEnum, long) : void
-setState(ModastStateEnum) : void
+getState() : ModastStateEnum
<<enumeration>>
ModastStateEnum
(br::ufpr::inf::rp09::statemachine::enumerations)
WARMUP_STATE
STEADY_STATE
UNDER_PRESSURE_STATE
STRESS_STATE
THRASHING_STATE
ModastStateEvaluator
(br::ufpr::inf::rp09::statemachine)
-WARMUP_THRESHOLD : double
-STEADY_THRESHOLD : double
-STRESS_THRESHOLD : double
-THRASHING_THRESHOLD : double
+getNextState(ModastStateEnum, double, double, double, double) : ModastStateEnum
ResultCollector
(org::apache::jmeter::reporters)
TestResultCollector
(br::ufpr::inf::rp09::testresults)
-startTime : long
-endTime : long
-totalTransactionsRequests : long
-totalTransactionsTreated : long
-totalError : long
+getElapsedTime() : long
+getStartTime() : long
+getEndTime() : long
+getTotalTransactionsRequested() : long
+getTotalTransactionsTreated() : long
+getTotalError() : long
HashTree
(org::apache::jorphan::collections)
ModastTestPlan
(br::ufpr::inf::rp09::testdefinitions)
-threadGroup : ThreadGroup
-loopController : LoopController
-sampler : AbstractTestElement
-testResultCollector : TestResultCollector
+setSampler(JDBCSampler) : void
+setNumThreads(int) : void
+setNumLoops(int) : void
ThreadGroup
(org::apache::jmeter::threads)
LoopController
(org::apache::jmeter::control)
AbstractTestElement
(org::apache::jmeter::testelement)
AbstractJDBCTestElement
(org::apache::jmeter::protocol::jdbc)
JDBCSampler
(org::apache::jmeter::protocol::jdbc::sampler)
ModastTestPlanFactory
(br::ufpr::inf::rp09::testdefinitions::factories)
+createVoltDBModastTestPlan(String, String, String, String) : ModastTestPlan
+createPostgreSQLModastTestPlan(String, String, String, String) : ModastTestPlan

ANEXO VI – Fluxograma do Algoritmo de Execução de Teste

O diagrama a seguir representa o algoritmo utilizado na execução de uma iteração de teste da aplicação de integração do JMeter ao MoDaST, como descrito na seção 3.3.2.3.



5. REFERÊNCIAS BIBLIOGRÁFICAS

[1] SMITH, David Mitchell. Hype cycle for cloud computing. *Gartner Inc., Stamford*, 2011.

[2] XIA, Feng et al. Internet of things. *International Journal of Communication Systems*, 25(9):1101-1102, 2012.

[3] BUYYA, Rajkumar et al. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer System*, 25(6):599–616, 2009.

[4] SOUSA, Flávio R. C.; MOREIRA, Leonardo O.; MACHADO, Javam C. Computação em Nuvem: Conceitos, Tecnologias, Aplicações e Desafios. *II Escola Regional de Computação Ceará, Maranhão e Piauí (ERCEMAPI)*, Teresina: SBC, 1:150-175, 2009.

[5] SILBERSCHATZ, Abraham et al. Database system concepts, 6th edition *McGraw-Hill, New York*, 1997.

[6] STONEBRAKER, Michael et al. The end of an architectural era: (it's time for a complete rewrite). *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007.

[7] STONEBRAKER, Michael. New SQL: An Alternative to NoSQL and Old SQL for New OLTP Apps; *Communications of the ACM*. Retrieved, 2012.

[8] KULKARNI, Keerti. Testing Services through Cloud, *A White Paper, MicroFocus.*

[9] MEIRA, Jorge Augusto. Model-based stress testing for database systems, 2014.

[10] Apache JMeter; <http://jmeter.apache.org>

[11] PostgreSQL; <https://www.postgresql.org>

[12] VoltDB; <https://voltdb.com>

[13] TPC; <http://www.tpc.org>

[14] Merriam-Webster; <http://www.merriam-webster.com/dictionary/database>

[15] CODD, Edgar F. A relational model of data for large shared data banks, *Communications of the ACM*, 13(6):377-387, 1970.

[16] IBM at 100; *IBM100 – Relational Database;* <http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/reldb>

[17] ORACLE Help Center; <https://docs.oracle.com>

[18] PRABHAKARAN, Kothandaraman; WENCE, Don; CHRISTENSEN, Brian. Stress testing database storage, *U.S. Patent n. 6,859,758*, 22 fev. 2005.

[19] BENTLEY, John E. Software testing fundamentals - concepts, roles, and terminology, *Proceedings of SAS Conference*, 2005.

[20] BROMBERG, Richard N.; KUPCUNAS, Richard W. Application and method for benchmarking a database server, *U.S. Patent n. 5,819,066*, 6 out. 1998.

[21] PostgreSQL Documentation; <https//www.postgresql.org/docs>

[22] Rick Kuhn, Yu Lei, and Raghu Kacker. Practical combinatorial testing: Beyond pairwise. *IT Professional*, 10:19–23, May 2008.

[23] SERPRO; <http://www.serpro.gov.br>

[24] NEVEDROV, Dmitri. Using JMeter to Performance Test Web Services, *Published on dev2dev (http://dev2dev.bea.com)*, 2006.

[25] Apache JMeter API; <http://jmeter.apache.org/api>

[26] Integração, PAPPE. Manual de Utilização da Ferramenta JMeter. Manual Técnico. *Instituto de Informática, UFG*. Goiânia, 2013.

[27] Intel Xeon Processor X5675; <http://ark.intel.com/products/52577/Intel-Xeon-Processor-X5675-12M-Cache-3_06-GHz-6_40-GTs-Intel-QPI>

[28] Mellanox Technologies: InfiniBand Performance Benchmarks; <http://www.mellanox.com/page/performance_infiniband>

[29] MADSEN, Mark; BLOOR Robin. The Database Revolution. A Perspective On Database: Where We Came From And We're Going. *Bloor Group & Third Nature*, 2012.

[30] Transaction Processing Performance Council (TPC). TPC Benchmark B. Standard Specification. *Transaction Processing Performance Council*, 1994.

[31] GROLINGER, Katarina et al. Data management in cloud environments: NoSQL and NewSQL data stores. *Journal of Cloud Computing: Advances, Systems and Applications*, 2(1):1, 2013.