UNIVERSIDADE FEDERAL DO PARANÁ

JORGE AUGUSTO MEIRA

MODEL-BASED STRESS TESTING FOR DATABASE SYSTEMS

CURITIBA/LUXEMBOURG

2014

PhD-FSTC-2014-44
The Faculty of Sciences, Technology and Communication

Department of Informatics

DISSERTATION

Defense held on 17/12/2014 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG EN INFORMATIQUE

AND

Ph.D. AT UNIVERSIDADE FEDERAL DO PARANÁ IN COMPUTER SCIENCE

by

Jorge Augusto MEIRA
Born on 9 December 1983 in Ponta Grossa (Brazil)

MODEL-BASED STRESS TESTING FOR DATABASE SYSTEMS

Dissertation defense committee

Dr. Eduardo Cunha de Almeida, dissertation supervisor
A-Professor, Universidade Federal do Paraná

Dr. Yves Le Traon, dissertation 2nd supervisor
Professor, Université du Luxembourg

Dr-Ing. Stefanie Scherzinger
Professor, Ostbayerische Technische Hochschule Regensburg

Dr Marcos Sfair Sunye, Chairman
Professor, Universidade Federal do Paraná

Dr Gerson Sunyé
A-Professor, Université de Nantes

Dr Jacques Klein, Vice Chairman
Adjoint de Recherche, Université du Luxembourg

M514m Meira, Jorge Augusto
Modelo baseado em testes de estresse para sistema de banco de dados / Jorge Augusto Meira. – Curitiba, 2014.
139f. : il. [algumas color.] ; 30 cm.

Tese (doutorado) - Universidade Federal do Paraná, Setor de Tecnologia, Programa de Pós-graduação em Informática; Universidade de Luxemburgo, Faculdade de Ciências, Tecnologia e Comunicação. 2014.

Orientador: Eduardo Cunha de Almeida.-- Co-rientador: Yves Le Traon.
Bibliografia: p. 131-139

1.Banco de dados - Gerencia. 2. Benchmarking (administração). I. Universidade Federal do Paraná. II. Universidade de Luxemburgo. III. Traon, Yves Le. IV. Almeida, Eduardo Cunha de. V. Título.

CDD: 005.74

Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa do aluno de Doutorado em Ciência da Computação, Jorge Augusto Meira, avaliamos a tese de doutorado intitulada *"Model-Based Stress Testing for Database Systems"*, cuja defesa pública foi realizada no dia 17 de dezembro de 2014, às 10:00 horas, na Universidade de Luxemburgo, Campus Kirchberg, na sala *"Salle de Conseils"* . Após avaliação, decidimos pela:

(X) **aprovação** do candidato. ( ) **reprovação** do candidato.

Curitiba, 17 de dezembro de 2014

Professor Doutor Eduardo Cunha de Almeida
DINF/UFPR – Orientador

Professor Doutor Yves Le Traon
Universidade de Luxemburgo – Coorientador

Pesquisador Doutor Jacques Klein
Universidade de Luxemburgo – Membro Externo

Professora Doutora Stefanie Scherzinger
Universidade de Ciências Aplicadas de Regensburg – Membro Externo

Professor Doutor Gerson Sunyé
Universidade de Nantes – Membro Externo

Professor Doutor Marcos Sfair Sunye
DINF/UFPR – Membro Interno

“Freedom is just another word for nothing left to lose”

Kris Kristofferson

Model-based Stress Testing for Database Systems

by

Jorge Augusto Meira

Abstract

Database Management Systems (DBMS) have been successful at processing transaction workloads over decades. But contemporary systems, including Cloud computing, Internet-based systems, and sensors (i.e., Internet of Things (IoT)), are challenging the architecture of the DBMS with burgeoning transaction workloads. The direct consequence is that the development agenda of the DBMS is now heavily concerned with meeting non-functional requirements, such as performance, robustness and scalability [85]. Otherwise, any stressing workload will make the DBMS lose control of simple functional requirements, such as responding to a transaction request [62]. While traditional DBMS, including DB2, Oracle, and PostgreSQL, require embedding new features to meet non-functional requirements, the contemporary DBMS called as NewSQL [56, 98, 65] present a completely new architecture.

What is still lacking in the development agenda is a proper testing approach coupled with burgeoning transaction workloads for validating the DBMS with non-functional requirements in mind. The typical non-functional validation is carried out by performance benchmarks. However, they focus on metrics comparison instead of finding defects.

In this thesis, we address this lack by presenting different contributions for the domain of DBMS stress testing. These contributions fit different testing objectives to challenge each specific architecture of traditional and contemporary DBMS. For instance, testing the earlier DBMS (e.g., DB2, Oracle) requires incremental performance tuning (i.e., from simple setup to complex one), while testing the latter DBMS (e.g., VoltDB, NuoDB) requires driving it into different performance states due to its self-tuning capabilities [85]. Overall, this thesis makes the following contributions: 1) **S**tress **TE**sting **M**ethodology (**STEM**): A methodology to capture performance degradation and expose system defects in the internal code due to the combination of a stress workload and mistuning; 2) **Mo**del-based approach for **Da**tabase **S**tress **T**esting (**MoDaST**): An approach to test NewSQL database systems. Supported by a Database State Machine (DSM), MoDaST infers internal states of the database based on performance observations under different workload levels; 3) **U**nder **P**ressure **B**enchmark (**UPB**): A benchmark to assess the impact of availability mechanisms in NewSQL database systems.

We validate our contributions with several popular DBMS. Among the outcomes, we highlight that our methodologies succeed in driving the DBMS up to stress state conditions and expose several related defects, including a new major defect in a popular NewSQL.

Resumo

Sistemas de Gerenciamento de Bando de Dados (SGBD) têm sido bem sucedidos no processamento de cargas de trabalho transacionais ao longo de décadas. No entanto, sistemas atuais, incluindo Cloud computing, sistemas baseados na Internet, e os sensores (ou seja, Internet of Things (IoT)), estão desafiando a arquitetura dos SGBD com crescentes cargas de trabalho. A conseqüência direta é que a agenda de desenvolvimento de SGBD está agora fortemente preocupada em atender requisitos não funcionais, tais como desempenho, robustez e escalabilidade [85]. Caso contrário, uma simples carga de trabalho de estresse pode fazer com que os SGBD não atendam requisitos funcionais simples, como responder a um pedido de transação [62]. Enquanto SGBD tradicionais exigem a incorporação de novos recursos para atender tais requisitos não-funcionais, os SGBD contemporâneos conhecidos como NewSQL [56, 98, 65] apresentam uma arquitetura completamente nova.

O que ainda falta na agenda do desenvolvimento é uma abordagem de teste adequada que leve em conta requisitos não-funcionais. A validação não-funcional típica para SGBD é realizada por benchmarks. No entanto, eles se concentram na comparação baseada em métricas em vez de encontrar defeitos.

Nesta tese, abordamos essa deficiência na agenda de desenvolvimento, apresentando contribuições diferentes para o domínio de testes de estresse para SGBD. Estas contribuições atendem diferentes objetivos de teste que desafiam arquiteturas específica de SGBD tradicionais e contemporâneos. No geral, esta tese faz as seguintes contribuições: 1) **S**tress **TE**sting **M**ethodology (**STEM**): Uma metodologia para capturar a degradação do desempenho e expor os defeitos do sistema no código interno devido a combinação de uma carga de trabalho de estresse e problemas de configuração; 2) **Mo**del-based **Da**tabase **S**tress **T**esting (**MoDaST**): Uma abordagem para testar sistemas de banco de dados NewSQL. Apoiado por uma máquina de estado de banco de dados (DSM), MoDaST infere estados internos do banco de dados com base em observações de desempenho sob diferentes níveis de carga de trabalho; 3) **U**nder **P**ressure **B**enchmark (**UPB**): Um benchmark para avaliar o impacto dos mecanismos de disponibilidade em sistemas de banco de dados NewSQL.

Nós validamos nossas contribuições com vários SGBD populares. Entre os resultados, destaca-se em nossas metodologias o sucesso em conduzir o SGBD para condições de estresse e expor defeitos relacionados, incluindo um novo *major bug* em um SGBD NewSQL popular.

Acknowledgements

My first and greatest thanks are for my family, especially for my mum Sueli and my dad Orlando. Thank you for your endless support and for teaching me at my young age that "studying is the most important thing in the world". I might have taken it too seriously!

I also thank my brother – the one that inspired me to follow this path. Joel, you are my role model Doctor!

My deep gratitude goes to my thesis advisors, Eduardo Cunha de Almeida and Yves Le Traon. Defending my thesis without your inspiring guidance, lessons, advices, conversation and support would have been impossible.

I thank the members of the jury: Stefanie Scherzinger, Mark Sfair Sunye, Gerson Sunye and Jacques Klein for their review and valuable comments. I also thank "Fonds National de la Recherche Luxembourg" (project 902399), for the financial support.

My colleague and friend Edson Ramiro Lucas Filho here deserves a special mention. Ramiro, thank you for your time spent helping me run countless experiments and for helping me understand the results. Those late working nights were not in vain. Thank you, man!

For the friendship and all the time invested in correcting my English, I warmly thank Irma Hadzalic. You're an amazing person and it is my pleasure to know you. You know how special you are to me.

Last but not least, I would like to thank all my other friends who supported me and believed in me even when I didn't believe in myself. For those in Brazil - thank you for Skype conversations and for those in Luxembourg - thank you for beers we shared. You were all essential to me in going through this period of my life. If I had to mention all your names, the "acknowledgments" page would be meters long and I would probably forget one or two. Therefore, I am certain you all know I refer to you.

Versão brasileira

Primeiramente, eu gostaria de agradecer minha família, especialmente meus pais: minha Mãe Sueli e meu Pai Orlando. Obrigado por todo apoio que sempre me deram e por sempre me incentivarem, lembro desde meus primeiros anos de vida quando vocês diziam que estudar era o mais importante, e acho que levei isso um pouco a sério demais!

Agradeço também ao meu irmão Joel. Você é a pessoa que me inspirou a seguir esse caminho, você é o Doutor que eu sempre tive como espelho!

Obrigado também aos meus orientadores, Eduardo Cunha de Almeida e Yves

LeTraon, por todo apoio e tempo que dedicaram na arte que é orientar. Sem dúvida eu não teria defendido minha tese sem todos os ensinamentos, conselhos, conversas e apoio que vocês me deram em toda essa jornada.

Agradeço aos membros do júri: Stefanie Scherzinger, Marcos Sfair Sunye, Gerson Sunyé e Jacques Klein, pela revisão e valiosos comentários sobre minha tese. Agradeço igualmente ao "Fonds National de la Recherche Luxembourg" (projeto 902399), pelo apoio financeiro.

Não posso deixar de citar aqui o meu colega e amigo Edson Ramiro Lucas Filho. Ramiro, muito obrigado por todo o tempo que você dedicou para me ajudar a executar tantos experimentos e a entendê-los! Aquelas madrugadas trabalhando não foram em vão, meu muito obrigado.

Agradeço de coração a amizade e o tempo investido na correção gramatical da minha tese por essa pessoa fantástica que tive o prazer de conhecer, obrigado Irma Hadzalic, você sabe o quanto você é especial pra mim.

Por último, mas não menos importante, todos os amigos que estiveram comigo, me apoiaram e fizeram eu acreditar que era possível, quando muitas vezes nem eu acreditava mais. Obrigado pelas conversas via skype para aqueles que estavam no Brasil e todas as cervejas para aqueles que estavam em Luxemburgo. Vocês foram imprescindíveis para que eu vencesse mais essa etapa da minha vida! Sem Dúvida Gostaria de citar o nome de todos vocês, mas não o faço por dois motivos: Primeiro porque essa seção de agredecimento ficaria deveras extensa e segundo porque eu provavelmente esqueceria alguém, e realmente não poderia ser injusto com nenhum de vocês. No entanto, tenho certeza que vocês saberão que aqui me refiro a cada um de vocês.

Jorge

Contents

List of Figures

List of Tables

Resumo Estendido

Introdução

Processamento transacional de alto desempenho é um dos aspectos fundamentais para um processamento de dados bem-sucedido, uma vez que o volume de transações está ficando maior na maioria das áreas de aplicação. Ao longo dos últimos 40 anos os Sistemas de Gerenciamento de Banco de Dados (SGBD) tradicionas, como DB2, Oracle, PostgreSQL, foram bem sucedidos em processamento de transações. No entanto, o recente crescimento da carga de trabalho transacional (por exemplo, Internet, computação em nuvem, BigData) está pressionando esses SGBD para além das suas capacidades de desempenho, exigindo uma profunda revisão em suas arquiteturas [85]. A conseqüência direta é que agora o desenvolvimento do SGBD precisa cuidar de vários requisitos não-funcionais, tais como desempenho, robustez e escalabilidade. Caso contrário, qualquer carga de trabalho de estresse fará com que o SGBD não cumpra requisitos funcionais simples, como responder um pedido de transação [62]. Enquanto estes SGBD tradicionais exigem a incorporação sistemática de novos recursos, a fim de se adequar às exigências não funcionais, SGBD contemporâneos, como NewSQL [56, 98, 65], apresentam uma arquitetura completamente nova.

Por um lado, SGBD tradicionais enfrentam parcialmente vários requisitos não-funcionais com esquemas de replicação e particionamento de banco de dados em uma arquitetura orientada a disco. Por outro lado, NewSQL aparece como uma alternativa mais proeminente para fornecer processamento de transacções de alto desempenho. NewSQL tem uma arquitetura diferente para abordar requisitos não-funcionais, mantendo bancos de dados inteiros na memória e processamento de transações através

de um esquema de múltiplo single-threaded. Comparado ao SGBD tradicionais, a arquitetura de memória não requer configurações de parâmetros complexas [85].

No contexto de processamento transacional de alto desempenho, os SGBD são implantados em ambientes distribuídos e expostos a conjuntos de vários níveis de carga de trabalho, incluindo deslocamentos transitórios de carga de trabalho ou picos repentinos, que podem resultar em defeitos distintos. Por exemplo, existem muitos padrões de carga de trabalho anormais: planos de consulta de desempenho fracos [82], backpressure[1], escalonamento de bloqueio (para o modelo baseado em locks) [16], estimativas de desempenho fracas [52], degradação de desempenho [86]. No entanto, a causa raiz para esses defeitos não são fáceis de detectar e podem "iludir detectores de erros por anos de execução", são assim conhecidos como "Heisenbugs" [44].

Infelizmente, técnicas de teste existentes para SGBD são apropriados apenas para os requisitos funcionais [102, 29] e não podem ser aplicados para problemas relacionados a desempenho.

Considerando requisitos não funcionais, SGBD são historicamente avaliados por benchmarks, tais como TPC[2] e Yahoo! Cloud Serving Benchmark (YCSB). No entanto, eles se concentram apenas em métricas para comparação e não em encontrar defeitos. Portanto, eles são incapazes de identificar ou prever defeitos de desempenho, como gargalos de desempenho ou thrashing em várias situações [82, 86].

É necessário estabelecer uma abordagem de teste adequada para desafiar os SGBD de alto desempenho, enviando sistematicamente volumes crescentes de transações até condições de estresse.

Neste contexto, o teste de estresse é uma escolha natural. Testes de estresse são amplamente aplicados em diferentes áreas de conhecimento (por exemplo, Medicina, Engenharia, Economia). Na ciência da computação, mais precisamente em engenharia de software, testes de estresse são importantes para avaliar o software sob cargas de trabalho pesadas com o objetivo de determinar os limites do desempenho e expor defeitos relacionados.

[1] https://voltdb.com/docs/UsingVoltDB/DesignAppErrHandling.php
[2] http://www.tpc.org

Motivação

Enquanto os benchmarks transacionais impõem um volume limitado de operações para avaliação de desempenho, testes de estresse dão vários níveis de carga de trabalho para fins de detecção de defeitos. Os diferentes níveis de carga de trabalho podem ajudar a alcançar caminhos do código-fonte que não são normalmente exploradas pelas cargas de trabalho de benchmarks atuais [2]. Isto indica que o SGBD usa módulos de código adicionais para cada nível de carga de trabalho e é esperado aumento da cobertura de código. Embora uma cobertura de código maior não garanta necessariamente a detecção de defeitos de software, aumenta a probabilidade de os revelar, especialmente nos módulos relacionados a requisitos não-funcionais, que é o nosso objetivo.

Neste contexto, testes de estresse são fundamentais para explorar diferentes comportamentos de um banco de dados em teste (DUT). Em geral, ao aplicar testes de software podemos testar vários caminhos de execução de um programa manipulando os valores de entrada. Figura 1-1 mostra uma diferença conceitual entre uma condição estável e uma de estresse. Além disso, mostra os possíveis diferentes caminhos de execução. Como a maioria dos SGBD estão sendo expostos a condições de estresse nos dias de hoje, eles têm várias funções para lidar com diferentes níveis de carga de trabalho. Simples alterações dos valores de entrada não são suficientes para explorar as funções desenvolvidas para lidar com condições de estresse. Se essas funções não forem testadas adequadamente, não podemos detectar defeitos potenciais relacionados. Isto inclui defeitos funcionais e não-funcionais. Por exemplo, uma violação de acesso a memória pode ser listada como um defeito funcional. Defeitos não-funcionais em um SGBD são expostos por fragmentos de código que são funcionalmente corretos mas que podem, por exemplo, degradar drasticamente o desempenho.

Figura 1-2 descreve o fragmento de código de um defeito descoberto por uma das nossas contribuições e acatado pelos desenvolvedores do VoltDB [3]. O *branch* a partir da **Linha 426** é executado somente quando o número de conexões concorrentes é igual ou superior ao número máximo predefinido de conexões (ou seja, o número

[3]https://issues.voltdb.com/browse/ENG-6881

especificado na **Linha 253**). O defeito pode ser destacado na declaração de condição da **Linha 426**.

A variável `m_numConnections` é acessado por vários *threads* e o seu valor pode alcançar um número maior do que `MAX_CONNECTIONS`. No entanto, o *branch* a partir da **Linha 426** só deve deve ser executado se `m_numConnections` $\geq$ `Max_CONNECTIONS`. Caso contrário, o DUT não pode rejeitar novas conexões e um eventual consumo excessivo de memória pode levar os SGBD para um estado de thrashing.

A Figura 1-3 mostra a cobertura de código dos testes de estresse e do benchmark TPC-B[4] executado no PostgreSQL (ver Capítulo 4 para mais detalhes). O eixo X é o tempo decorrido em segundos, e o eixo Y à esquerda representa a proporção de linhas cobertas por cada caso de teste. O eixo Y representa a taxa de transferência do DUT em transações por segundo. Enquanto TPC-B contempla poucos níveis de carga de trabalho (sem pressão ou stress), um teste de estresse leva diferentes níveis de carga de trabalho para testar vários comportamentos do DUT. Isto aumenta a probabilidade de encontrar defeitos.

Contribuição

Nesta tese, apresentamos contribuições diferentes para o domínio ds teste de estresse para SGBD. Estas contribuições atendem diferentes objetivos de teste para desafiar arquiteturas específicas de SGBD, tais como, tradicionais e contemporâneos. Por exemplo, enquanto SGBD tradicionais requerem extensa configuração de parâmetros, SGBD contemporâneos implementam a abordagem "no knobs" [85]. Esta diferença muda drasticamente suas arquiteturas internas e requer metodologias de teste de diferentes. No geral, esta tese faz as seguintes contribuições:

- **Stress Testing Methodology (STEM)**: Uma metodologia para capturar a degradação do desempenho e expor defeitos do sistema devido à combinação de uma carga de trabalho de estresse e erros de configuração;

[4] http://www.tpc.org/tpcb/

- **Model-based Database Stress Testing (MoDaST)**: Uma abordagem para testar sistemas de banco de dados NewSQL. Apoiado por uma *Database State Machine (DSM)*, MoDaST infere estados internos do banco de dados com base em observações de desempenho sob diferentes níveis de carga de trabalho;

- **Under Pressure Benchmark (UPB)**: Um benchmark para avaliar o impacto dos mecanismos de disponibilidade em sistemas de banco de dados NewSQL. Embora UPB seja concebido como um benchmark, suas métricas avaliam a perda de desempenho do SGBD ao ativar seus mecanismos de disponibilidade, tais como, replicação.

Questões abertas em teste para SGBD

Teste de SGBD podem ser classificados em duas categorias: funcionais e não-funcional. Por um lado, o teste funcional certifica a capacidade do sistema em reproduzir uma saída adequada para uma entrada específica. Por outro lado, o teste não-funcional considera requisitos não-funcionais relativos à qualidade do sistema, tais como desempenho, robustez, segurança, escalabilidade.

Ao longo dos últimos anos, testes não-funcionais tornaram-se críticos devido ao recente crescimento da carga de trabalho que impacta diretamente sobre o desenvolvimento SGBD. Enquanto SGBD tradicionais exigem a incorporação sistematica de novos recursos, para se ajustar a esses requisitos, SGBD contemporâneos apresentam uma arquitetura completamente nova.

O principal desafio em testes de SGBD é estabelecer uma metodologia adequada, enviando coordenadamente volumes crescentes de transações afim de recriar cargas de trabalho de produção.

Abordagens atuais de teste para SGBD

Validação de SGBD é comumente realizada através de benchmarks. Abordagens diferentes de benchmarks foram propostas ao longo das últimas décadas, com foco em métricas para comparação (por exemplo, o tempo de resposta, taxa de tranferência, e

consumo de recursos) [53]. Alguns exemplos de benchmarks são: Débito/Crédito [34], AS3AP [93], TPC-like [2], SetQuery [67], YCSB [19]. Tipicamente benchmarks não buscam revelar defeitos do SGBD e nem se concentram na avaliação dos novos recursos de SGBD contemporâneos.

Ferramentas de teste de desempenho, tais como Hammerora [1], Oracle Application Testing Suite [6], e AppPerfect [7], fornecem um piloto de testes para submeter as operações com base em benchmarks do tipo TPC. Concentrando-se em avaliações funcionais, a ferramenta Agenda [30] fornece uma metodologia de testes para validar as propriedades ACID do SGBD.

Em relação a testes baseados em modelo, várias abordagens são focadas nas necessidades de avaliação de desempenho. Na verdade, em cenários específicos, eles são capazes de analisar requisitos funcionais, bem como requisitos não-funcionais, tais como, segurança [71, 32, 75], confiabilidade [40] e eficiência [60, 40, 78], mas nenhum deles específicos para SGBD.

Stress Testing Methodology (STEM)

O objetivo da STEM é capturar a degradação do desempenho e expor os defeitos do sistema no código interno, devido à combinação de uma carga de trabalho pesada e erros de configuração. Um defeito difere de uma degradação do desempenho no sentido de que o sistema não fornece o serviço esperado, conforme especificado (ou seja, os pedidos de transação não são aceitos, enquanto deviam ser). STEM segue uma abordagem incremental. A sequência de execução consiste de muitos passos para conduzir o SGBD de uma condição inicial até uma condição de estresse. A abordagem tem 3^3 combinações de configuração de *work_mem*, *max_conn* e *carga de trabalho*. Usamos *pairwise testing*[5] para limitar o número de combinações. O número final de combinações foi definido para 5 (ver Tabela 3.2) e formam a base dos passos da STEM:

[5] *"pairwise testing é um método combinatório em que todos os possíveis pares de valores dos parâmetros são cobertos por pelo menos um teste [57]."*

1. Configuração inicial SGDB com carga de trabalho inicial;

2. Ajuste do *buffer pool* com carga de trabalho de estresse;

3. Ajuste do SGBD com carga de trabalho base;

4. Ajuste do SGBD com carga de trabalho de estresse, até o limite de desempenho configurado no SGBD;

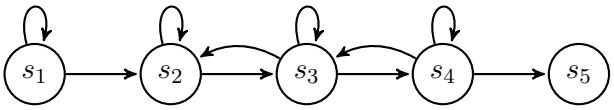
5. Ajuste do SGBD com carga de trabalho de estresse além do limite configurado (degradação de desempenho);

A primeira etapa visa buscar por defeitos relativos à qualquer funcionalidade SGBD, defeitos de instalação e configuração incorreta. O objetivo do segundo passo é estabelecer a linha base de degradação do SGBD. A terceira etapa tem como objetivo validar aspectos funcionais após a fase de ajuste. O propósito do quarto passo é procurar defeitos relativos ao limite de desempenho SGBD. A meta do quinto passo é empurrar o SGBD para além do seu limite de desempenho, reproduzindo condições de estresse.

Podemos tirar muitas conclusões interessantes a partir dos resultados experimentais após a aplicação STEM. Em primeiro lugar, testes de estresse requerem uma abordagem de teste distribuído, no qual testadores distribuídos são implantados para reproduzir um grande número de transações. Abordagens centralizadas, utilizadas nos trabalhos relacionados, limita o tamanho da carga, conduzindo assim a um teste funcional clássico. A execução de STEM é simples, mas em contraste com trabalhos relacionados, requer muitas máquinas para execução. Na verdade, um teste funcional clássico só iria apontar defeitos do SGBD em uma etapa inicial da STEM sem relação qualquer com condições de estresse. O defeito destacado pela STEM ocorre devido a uma limitação de tamanho de um vetor interno que é responsável por gerenciar os processos de back-end. Com o grande número de pedidos simultâneos, o vetor enche-se rapidamente, impedindo assim o tratamento de novos processos (incluindo novas solicitações de transacção).

Model-based Database Stress Testing (MoDaST)

Nossa abordagem *Model-based Database Stress Testing* (MoDaST) visa revelar defeitos não-funcionais potenciais no SGBD, especialmente NewSQL. MoDaST se concentra no teste de desempenho do SGBD com alterações dinâmicas dos níveis de carga de trabalho. MoDaST utiliza uma máquina de estado a *Database State Machine* (DSM), a fim de simular cargas de trabalho de diferentes níveis. A DSM impulsiona o SGBD em cinco estados observáveis: Warm-up, Estável, Sub-pressão, Estresse e Thrashing. A Figura abaixo mostra a DSM e a Definição 1 define formalmente a DSM e seus estados correspondentes.



Definition 1. *A máquina de estados de banco de dados denotada como T, é uma 5-tupla* $(\mathcal{S}, s_1, \mathcal{F}, \beta, \tau)$ *na qual:*

- $\mathcal{S} = \{s_1, s_2, s_3, s_4, s_5\}$ *é o conjunto de estados,*
- $s_1 \in \mathcal{S}$ *é o estado inicial,*
- $\mathcal{F} \subset \mathcal{S}$ *é o conjunto de estados finais, onde* $\mathcal{F} = \{s_5\}$ *na DSM,*
- β *é o conjunto de entrada de desempenho definidos em Definição 2,*
- τ *é a função de transição de estados.*

DSM tem três entradas diferentes de desempenho (β): 1) variação de desempenho, 2) rendimento da transação, e 3) tendência de desempenho, conforme descrito na Definição 2.

Definition 2. *A* **Entradas de Desempenho**, *denotada por* β, *é uma 3-tupla:* $\beta =< \Delta, \delta, \varphi >$, *onde* Δ *é a variação de desempenho ,* δ *é a taxa de transferência de transações, e* φ *é a tendência de desempenho.*

A principal diferença do STEM para MoDaST é que o foco da STEM é sobre a relação entre as cargas de trabalho e ajuste do sistema, enquanto a MoDaST concentra-se em conduzir o SGBD através dos estados de desempenho. NewSQL se baseia principalmente na abordagem "no knobs", o que significa menos interferência humana no ajuste do sistema. Portanto, STEM não pode ser aplicado neste caso, assim MoDaST apresenta uma abordagem de teste mais apropriado para NewSQL.

Embora a meta para MoDaST seja originalmente testar NewSQL, nós também testamos PostgreSQL, com ajustes em sua configuração básica. MoDaST pode inferir o estado interno do DUT com base no modelo de estados. Além disso, reafirmamos que a alta carga de trabalho pode levar para uma cobertura de código superior. Consequentemente, identificamos novos defeitos em ambos os banco de dados. Em particular, um dos defeitos foi confirmado e corrigido pelos desenvolvedores do VoltDB.

Para os engenheiros de teste, MoDaST se apresenta como uma ferramente conveniente para avaliar requisitos não-funcionais, incluindo, performance, estabilidade, e/ou escalabilidade. Os engenheiros de teste podem também constatar que a MoDaST aumenta a cobertura de código na tarefa de encontrar defeitos. Embora uma cobertura de código superior não garanta necessariamente a detecção de defeitos, pode apoiar a configuração de perfis de desempenho. Por exemplo, a maioria dos SGBD tem partes de código específicos para o tratamento de condições de estresse que só são executadas em determinada condições. MoDaST pode ajudar a exercitar essas partes de código.

Para DBAs, MoDaST pode ser uma ferramenta poderosa para previsão de estados de estresse e *Thrashing* em um monitoramento dinâmico. Essa previsão é particularmente útil para identificar limitações de configuração do DUT.

Under Pressure Benchmark (UPB)

UPB foca nos requisitos de disponibilidade, avaliando a perda de desempenho devido à replicação do banco de dados. Enquanto a replicação de dados está em funcionamento, o impacto sobre o desempenho pode variar dependendo tanto do ambiente de avaliação

como da implementação de diferentes SGBD. A UPB avalia os impactos da replicação no desempenho, com diferentes configurações (ou seja, fator de replicação e injeção de falhas) e cargas de trabalho. O benchmark UPB é composto por duas tarefas: 1) definir os cenários de disponibilidade e 2) executar os cenários.

Nós definimos cenários que representam o conjunto de estados possíveis de disponibilidade. Os cenários são escolhidos com base na combinação dos valores de duas variáveis, como descrito abaixo:

1. Índice de tolerância a falhas (K): quantidade de falhas sem interrupção do serviço. Valores possíveis para K são:
   - $k = 0$ (sem tolerância a falhas): o SGBD pára na presença de qualquer falha.
   - $K = 1, 2, ..., \frac{N}{2}$: N representa número de máquinas que compõem o SGBD. Neste caso, o SGBD suporta falhas em K nós.
2. Número de nós falhos (F).
   - $F = 0$: cluster sem falhas.
   - $F = 1, 2, ..., K$: cluster com F falhas. Os valores estão entre 1 e K.

No entanto, não é necessário ter um cenário para todas as combinações de valores, uma vez que alguns cenários não pode ocorrer na prática. A Tabela abaixo mostra possíveis valores das variáveis, a relevância de cada combinação e um comentário.

Combinação	**K**	**F**	**Relevância**	**comentário**
1	0	0	Yes	Base de comparação
2	0	Y	Inviável	$F > K$
3	X	0	Yes	Impacto do mecanismo de tolerância a falhas
4	X	Y	Yes	Impacto da injeção de falhas

Após a análise das combinações, três cenários são aplicados pela UPB:

- Cenário (1) - Combinação 1: Sem tolerância a falhas ($K = 0$), sem falhas ($F = 0$);

- Cenário (2) - Combinação 3: Com tolerância a falhas ($K > 0$), sem falhas ($F = 0$);
- Cenário (3) - Combinação 4: Com tolerância a falhas ($K > 0$), com falhas ($F > 0$ / $F < K$).

UPB difere do STEM e MoDaST, principalmente em seu foco. UPB é projetado para ser um benchmark, não um teste. No contexto de benchmarking, UPB também difere dos trabalhos relacionados ao nosso conhecimento. A UPB fornece um benchmark focado em questões centrais relacionadas a disponibilidade do SGBD, ou seja, mede a perda de desempenho do SGBD ao ativar mecanismos de replicação. Acreditamos que a UPB se encaixa nos requisitos para a avaliação NewSQL sobre situações críticas, como cargas pesadas e falhas. Além disso, a UPB fornece uma boa base para os administradores de banco de dados para tomar uma decisão sobre índices de replicação, com base no impacto no desempenho.

Nós validamos a UPB através de experimentos e avaliações de dois SGBD NewSQL: VoltDB e NuoDB. Verificamos que a replicação de dados tem um grande impacto no desempenho, como um efeito colateral da disponibilidade. O impacto pode ser considerado positivo ou negativo, dependendo o SGBD. Isto é mais evidente quando o SGBD está sob pressão.

Conclusão e Trabalhos Futuros

Neste capítulo, apresentamos as conclusões gerais desta tese. Primeiro apresentamos o resumo das nossas contribuições e em seguida apontamos as direções para trabalhos futuros.

Contribuição

Nesta tese, apresentamos três contribuições para atender diferentes necessidades em avaliação e validação de SGBD: 1) Nossa metodologia de testes de estresse para SGBD tradicionais para expor defeitos relativos à combinação de uma carga de trabalho

de estresse e problemas de configuração; 2) Uma abordagem de teste de estresse com base em modelo para NewSQL, que infere estados internos de desempenho do SGBD e expõe defeitos relacionados a esses estados.; 3) Um Benchmark para avaliar mecanismos de disponibilidade em SGBD NewSQL.

1. **Stress Testing Methodology (STEM)**: STEM revelou defeitos relacionados com a combinação de cargas de trabalho de estresse e problemas de configuração de SGBD tradicionais. Podemos tirar muitas conclusões interessantes a partir dos resultados. Em primeiro lugar, os testes de estresse requerem uma abordagem de teste distribuído para não limitar a geração da carga de trabalho. A abordagem incremental foi capaz de expor defeitos diferentes do SGBD. Assim, a STEM pode ser considerada uma ferramenta importante para avaliar a limitação de desempenho de SGBD tradicionais sob condições de estresse.

2. **Model-based Database Stress Testing (MoDaST)**: Nossa abordagem baseada em modelo (MoDaST) visa revelar defeitos não-funcionais potenciais no DUT, especialmente NewSQL. Os resultados experimentais mostraram que MoDaST consegue inferir estados internos do SGBD com base em uma máquina de estado (DSM). Além disso, descobrimos que, conforme o SGBD visita os estado de performance em direção ao estado de thrashing, a cobertura de código aumenta. Por conseguinte, a probabilidade de defeitos descobertas aumentar do mesmo modo.

3. **Under Pressure Benchmark (UPB)**: UPB avaliou o desempenho de SGBD NewSQL considerando mecanismos de disponibilidade. Acreditamos que a UPB se encaixa nos requisitos de avaliação de SGBD em condições críticas, como as cargas de trabalho sob pressão e falhas. Além disso, a UPB fornece uma boa base para os administradores de banco de dados tomarem decisões sobre índices de replicação, com base no impacto do desempenho. Verificamos que a replicação de dados tem um grande impacto no desempenho, tanto negativo como positivo, como um efeito colateral da disponibilidade.

Trabalhos Futuros

Focamos nosso trabalho futuro na MoDaST. Esta abordagem provou ser a mais adequado para representar o SGBD e revelar defeitos relacionados. Até agora, MoDaST pode ser aplicada de diferentes maneiras. Para os engenheiros de teste, MoDaST oferece um driver conveniente para avaliar os requisitos não-funcionais, incluindo, performance, estabilidade, ou escalabilidade. Enquanto nossos experimentos focam em desempenho, engenheiros de teste podem facilmente estender o caso de teste para outras avaliações.

A nossa abordagem pode ser conectada a qualquer SGBD, incluindo sistemas de código fechado. Engenheiros de teste também podem usar o fato de que nossa abordagem aumenta a cobertura de código, facilitando encontrar defeitos. Embora uma maior cobertura de código não garanta necessariamente a detecção de defeitos, os engenheiros podem de beneficiar da maior cobertura para criação de perfis de desempenho e configuração.

Para DBAs, MoDaST pode ser uma ferramenta poderosa para previsão e monitorament dinâmico de estados de "Estress" e "Thrashing".

Para trabalhos futuros, pretendemos aplicar MoDaST para *cloud hypervisors*, a fim de monitorar os sistemas *database-as-a-service (DBaaS)*. Para isso, estados adicionais podem ser criados, tal como um estado *alto estresse* entre estresse e thrashing, ou um estado *ocioso* caso o sistema esteja alocando mais recursos do que o necessário.

Chapter 1

Introduction

Scalable and high performance transaction processing is one of the key aspects for a successful data business processing, once the volume of incoming transactions gets bigger in most application areas. Over the last 40 years traditional Database Management Systems (DBMS), such as DB2, Oracle, PostgreSQL, have been successful at processing transactions. However, the recent growth of the transaction workload (e.g., Internet, Cloud computing, BigData) is challenging these DBMS beyond their performance capabilities requiring a thorough revision in their architectures [85]. The direct consequence is that now the development of the DBMS needs to look after several non-functional requirements, such as performance, robustness and scalability. Otherwise, any stressing workload will make the DBMS lose control on simple functional requirements, such as responding to a transaction request [62]. While these traditional DBMS require systematically embedding new features in order to fit non-functional requirements, contemporary DBMS, such as NewSQL [56, 98, 65], present a completely new architecture.

On the one hand, traditional DBMS partially tackle several non-functional requirements with replication and database partitioning schemes in a disk-oriented architecture. This architecture is strongly based on three main features: (1) complex multi-threaded transaction processing schedules, (2) distributed database consistency protocols, and (3) extensive performance tuning to make the DBMS deliver high transaction throughputs. On the other hand, NewSQL appears as the most

prominent alternative to deliver high performance transaction processing. NewSQL takes a different architecture to tackle non-functional requirements by keeping entire databases in-memory and processing transactions through a multiple single-threaded scheme. Compared to traditional DBMS, the in-memory architecture does not require extensive tuning [85].

In the context of high performance transaction processing, the DBMS are supposed to be deployed in distributed environments and to experience a various set of workload levels including transient workload shifts or sudden spikes, which can result in a number of different defects. For example, there are many abnormal workload patterns: poor performance query plans [82], backpressure[1], lock escalation (for lock-based mode) [16], poor performance estimations [52], performance degraded mode and load shedding [86]. However, the root cause for these defects are not easy to detect and may "elude the bugcatcher for years of execution", which is also called "Heisenbugs" [44].

Unfortunately, existing testing techniques for DBMS are only appropriate for functional requirements [102, 29] and cannot be applied here. With non-functional requirements in mind, DBMS are historically evaluated by benchmarks, such as TPC[2] and Yahoo Cloud Serving Benchmarks (YCSB). However, they only focus on metrics comparison rather than finding defects. Thus, they are unable to identify or predict performance defects, such as bottlenecks or performance thrashing in various situations [82, 86].

It now becomes evident that it is necessary to establish a proper testing approach to challenge the high performance DBMS by systematically submitting increasing volumes of transactions until stress conditions. In this context, stress testing is a natural choice. Stress testing is widely applied in different areas (e.g., Medicine, Engineering, Economics). In computer science, more precisely in software engineering, stress testing is important for evaluating the software upon stress workloads with the objective of determining the boundaries of performance and exposing potential defects.

[1] https://voltdb.com/docs/UsingVoltDB/DesignAppErrHandling.php
[2] http://www.tpc.org

1.1 Motivation

While the transactional TPC-like benchmarks impose a limited volume of transactions for performance assessment, stress testing gives several different workload levels for defect finding purposes. The difference can be spotted internally in the DBMS. With stress testing, the different workload levels can help reaching paths of the source code that are not usually explored by the current workloads of the TPC-like benchmarks. This indicates that the DBMS invokes additional code modules for each workload level and the increase of code coverage is expected. Although a higher code coverage does not necessarily guarantee the detection of software defects, it increases the probability to reveal them, specially in the modules related to non-functional requirements, which is our goal.

In this context, stress testing is critical to explore different behaviors of a Database Under Test (DUT). In general software testing, we may test several execution paths in a program by manipulating input values of the program. Figure 1-1 shows a conceptual difference between a stable and a stress condition; moreover, it shows, why both conditions can result in different execution paths. Since most of the DBMS are being exposed to stress conditions nowadays, they have several functions to deal with different levels of workload. Simply changing input values may not explore the functions for a high workload condition. If we do not test those functions properly, we cannot detect potential defects. These defects include non-functional and functional defects as shown. For instance, memory access violation (e.g., index out of bound or null pointer access) and type errors (e.g., wrong class casting) are listed as functional defects. Non-functional defects in a DBMS are exposed by code fragments that are functionally correct but dramatically degrade performance.

Figure 1-2 depicts the code fragment of a major defect spotted by one of our contributions and reported by the VoltDB developers [3]. The branch starting from **Line 426** can be executed only when the current number of connections is equal to the predefined maximum number of allowed connections (i.e., the number specified

[3]MoDaST spotted a major defect in VoltDB, details at: https://issues.voltdb.com/browse/ENG-6881

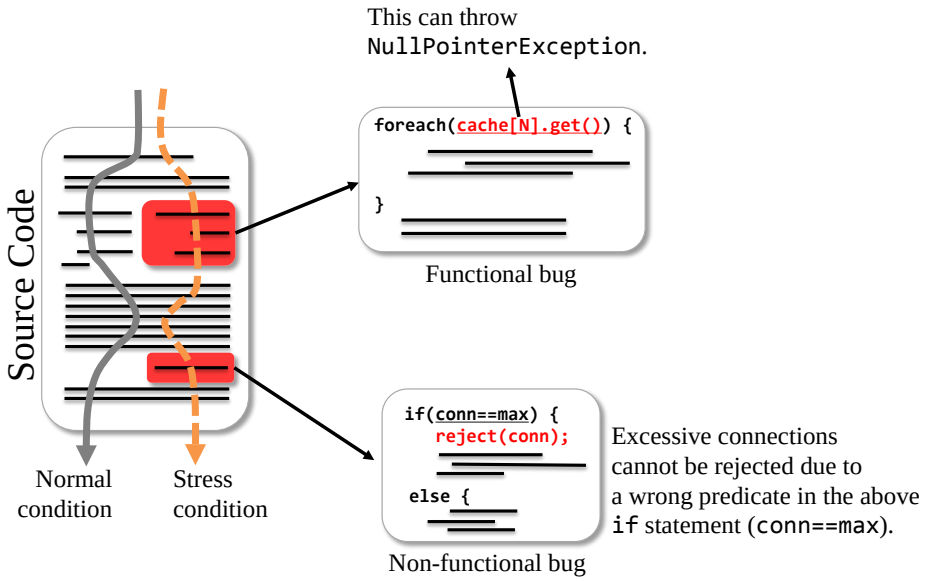


Figure 1-1: Conceptual execution paths under normal and stress conditions.

in **Line 253**). The defect can be spotted in the condition statement at **Line 426**.

The variable `m_numConnections` can be accessed by several threads and its value can be larger than `MAX_CONNECTIONS`. However, the branch starting from **Line 426** should be executed if `m_numConnections` $\geq$ `MAX_CONNECTIONS`. Otherwise, the DUT cannot reject new connections and the eventual excessive memory consumption may lead the DBMS to a thrashing state.

Figure 1-3 shows the code coverage of stress testing and the TPC-B benchmark[4] executed on PostgreSQL (see Chapter 4 for details). The X-axis is the time elapsed in seconds and the Y-axis on the left edge represents the proportion of covered lines by each testing case. The Y-axis represents the throughput of the DUT in transactions per second. While TPC-B commits fewer number of workload levels (without pressure or stress), a stress testing drives different workload levels to test various behaviors of the DUT. This increases the probability to find a potential defect.

[4]http://www.tpc.org/tpcb/

```
  ...

253  private final AtomicInteger MAX_CONNECTIONS
         = new AtomicInteger(800);
  ...

426  if (m_numConnections.get() == MAX_CONNECTIONS.get()) {
427    networkLog.warn(''Rejected connection from '' +
428      socket.socket().getRemoteSocketAddress() +
429      '' because the connection limit of '' +
                MAX_CONNECTIONS + " has been reached");
430   try {
431    /*
432    * Send rejection message with reason code
433    */
434    final ByteBuffer b = ByteBuffer.allocate(1);
435    b.put(MAX_CONNECTIONS_LIMIT_ERROR);
```

Figure 1-2: Example of defect in DBMS that can be identified only under stress conditions.

1.2 Contribution

In this thesis, we present several contributions for the domain of DBMS stress testing. These contributions fit different testing objectives to challenge each specific architecture of traditional and contemporary DBMS. For instance, while the earlier DBMS requires extensive tuning, the latter is said to be "no knobs" [85]. This difference dramatically changes their internal architectures and requires presenting different testing methodologies as well. Overall, this thesis makes the following contributions:

- **Stress Testing Methodology (STEM)**: A methodology to capture performance degradation and to expose system defects in the internal code due to the combination of a stress workload and mistuning;
- **Model-based approach for Database Stress Testing (MoDaST)**: An approach to test NewSQL database systems. Supported by a Database State Machine (DSM), MoDaST infers internal states of the database based on performance observations under different workload levels;
- **Under Pressure Benchmark (UPB)**: A benchmark to assess the impact of availability mechanisms in NewSQL database systems. Although UPB is de-

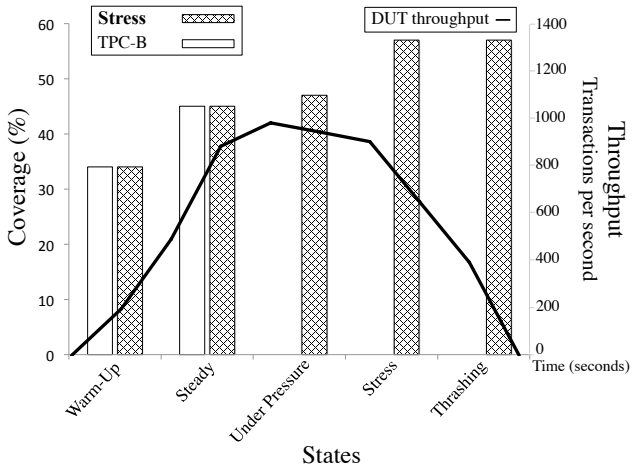


Figure 1-3: Example of inherent limitations of the existing techniques

signed as a benchmark, its metrics assess performance loss when DBMS activate their availability mechanisms, such as replication.

1.2.1 Stress TEsting Methodology (STEM)

The STEM claims that mistuning is crucial for detecting defects in traditional DBMS. Therefore, the goal of STEM is to capture the performance degradation and to expose system defects in the internal code due to the combination of a stress workload and mistuning. A defect differs from a performance degradation in the sense that the system does not provide the expected service as specified (i.e., transaction requests are not accepted as they should be). STEM follows an incremental approach: the DUT is configured from its simplest setup to the most complex one, which is done by tuning the setup gradually.

1.2.2 Model-based approach for Database Stress Testing (MoDaST)

A novel model-based approach to reveal potential non-functional defects in DUT, specially NewSQL. MoDaST focuses on testing performance of the DUT with dynamically changing workload levels. MoDaST leverages a Database State Machine (DSM) model in order to mimic different workload levels. The DSM drives the DUT

across five observable states: Warm-up, Steady, Under-Pressure, Stress, and Thrashing. The advantage compared to other testing tools is that the model allows users to infer and explore internal states of a DUT even if only black-box testing is available. In addition, users may change the workload levels in order to challenge the DUT for different test objectives. For instance, the testing objective can be detect performance loss conditions or thrashing states.

The main difference from STEM to MoDaST is that the earlier focuses on the relationship between workloads and system tuning, while the latter focuses on driving the system to particular performance states. NewSQL is mostly based on the "no knobs" approach, which means less human interference in system tuning. Therefore, STEM can not be applied here and MoDaST appears then as the most appropriated testing approach for NewSQL.

1.2.3 Under Pressure Benchmark (UPB)

In the UPB, we focus on availability requirements by assessing performance loss due to database replication. While data replication is up and running, the impact on performance may vary depending both on the evaluation environment and the proper implementation that differs in different DBMS. The UPB assesses the impacts of replication on performance with different configurations (i.e., replication factor and failed nodes) and workloads. The workload varies up to under pressure conditions, in terms of failures and bulk load.

UPB differs from STEM and MoDaST, mainly by its focus. UPB is designed to be a benchmark. To our knowledge, the UPB also differs from the related work in the benchmarking context. While the related work measures performance for finding the best tuning for a given workload, UPB measures performance loss when DBMS activate the availability mechanisms of replication.

1.3 Outline

The remainder of this thesis is organized as follows. In Chapter 2, we introduce basic concepts of DBMS and software testing to establish our context. We also present a survey of the current testing techniques for DBMS and model-based approaches by comparing them and showing why they do not fit stress testing for DBMS.

In Chapter 3, we present the STEM testing methodology and our validation through experimentation. In Chapter 4, we present MoDaST in three parts. First, we present the Database State Machine, its states and transitions. Second, we present the test driver used to proceed the evaluation. Third, we present experimental results in a popular NewSQL. In Chapter 5, we present the UPB and its evaluation methodology. Then, we present the experiments and discuss the results. In Chapter 6, we conclude this thesis and give suggestions for the future work.

Chapter 2

State of the art

2.1 Introduction

In this chapter we start presenting an overview of Database Management Systems (DBMS) to contextualize our work. Then, we introduce software testing concepts and the different approaches for testing DBMS. Next, we survey model-based approaches for general testing purposes. Finally, we compare these techniques and discuss inherent weaknesses.

2.2 Traditional Database Management Systems

The concerns about storage and digital data management have started as soon the digital data has been created. In early 1960s, the first general purpose Database Management System (DBMS) has been launched and its name was Integrated Data Store. The Integrated Data Store has influenced database systems through the 1960s and, has as well, strongly influenced the network data model for databases. In the same decade the Information Management System (IMS), by IBM, was developed. The IMS introduced an alternative data representation called hierarchical data model.

Later, in the early 1970s the IBM's research laboratory proposed the first Relational DBMS, based on a relational data model representation, the System R. This model was consolidated as dominant position in the 1980s [74] and was responsible

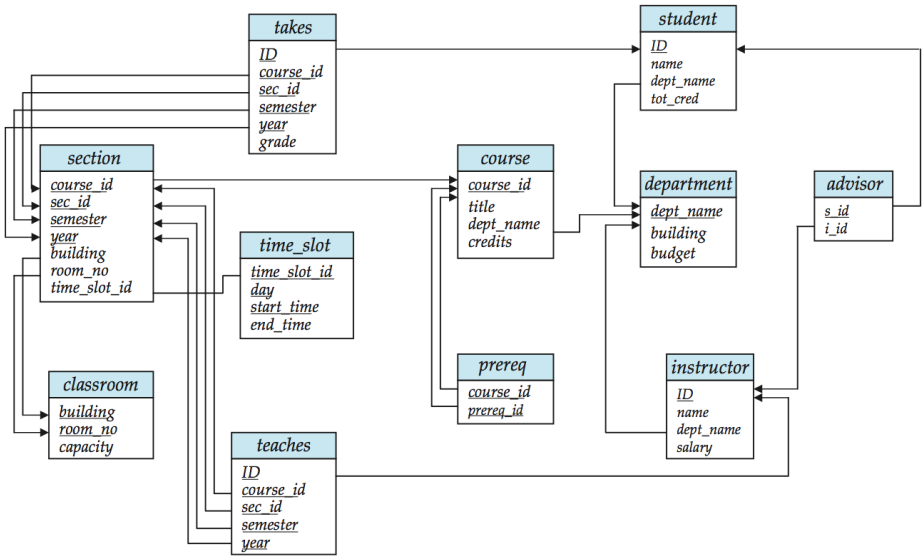


Figure 2-1: Schema diagram for the university database [80].

for a race in the industry for developing several DBMS based in the same model.

In the relational model the relationship between the data is represented by a collection of tables [80]. The relational model provides a high level abstraction of the low-level details of data storage. Basically, this model is composed of relations, tuples and attributes. The term *relation* refers to a table, the term *tuple* refers to a row and the term *attribute* refers to the columns of a table. The logical design of a relation is called *schema* (see Firgure 5-2).

The stored data is handled by query languages (e.g., SQL), which allows the user to execute different operations, such as insert, update, remove and select. From the point of view of the database user, a set of operations may be a part of a single unit database manipulation. For instance, when a customer from a bank uses an automatic teller machine to check his account, deposits an amount of money and then transfers it. This single unit database operation is referred to as transaction. A transaction is a group of SQL statements, which must be committed or rolled back (aborted) depending on its execution. If the transaction is executed without errors in any statement, the modified data is recorded on the database. If any statement that composes the transaction is not successfully executed, the transaction must be rolled

back and any change undone, in order to guarantee the ACID properties [39]:

- **Atomicity**: Each operation that composes a transaction must be executed successfully (i.e., commit), or none operation action is executed (i.e., rollback).
- **Consistency**: The execution of a transaction must keep the database consistency, which means the transaction does not violate any defined rule.
- **Isolation**: Concurrent transactions must not have an influence on the results from each other.
- **Durability**: Once a transaction is successfully committed, its changes on the database are permanent.

Once the data is potentially accessed by more than one transaction at the same time, the DBMS must use some mechanism to ensure **Isolation**. The general mechanism for concurrency control is based on data locks [80]. Locks are implemented in two levels: 1) Shared lock, in which the transaction have the rights to read the data, but cannot write; 2) Exclusive lock, in which the transaction can do both, read and write.

Internally to the DBMS, the management of locks is implemented within the concurrency control subsystem. The concurrency control is responsible of ensure that concurrent transactions will produce the same results as if they were executing one-at-a-time (i.e., the concept of Serializability). For that, a scheduler for synchronizing access to the database is necessary. Trivially, it is clear that shared locks cannot cause any concurrency problems in the DBMS as it is meant for read only operations in the database. However, when more than one transaction requires an exclusive lock for concurrent write operations in the same datum, the scheduler plays an important role in order to ensure **Isolation** and thus a consistent database [70]. Figure 2-2 shows an example of "serializable" and "not serializable" schedules. The loop means a "not serializable" schedule and must be avoided, for example, by delaying one of the conflicting transaction.

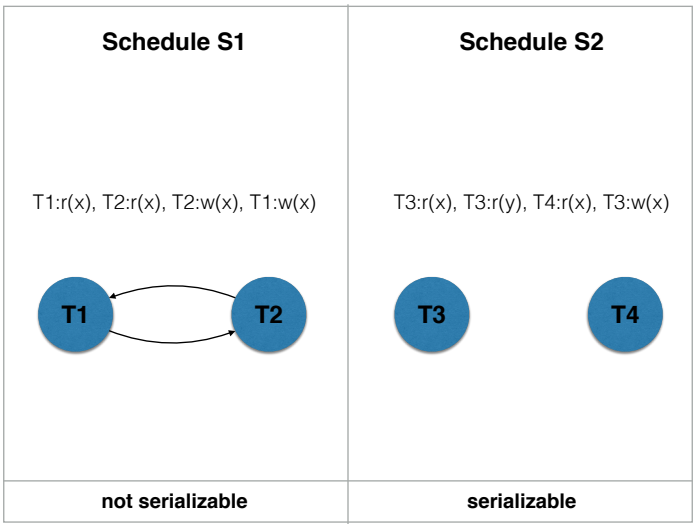


Figure 2-2: Transaction schedules: lock example

The high-concurrency has special impacts on Online Transaction Processing (OLTP) applications. OLTP transactions require "fast commit processing" for supporting high throughputs on data updates [80, 85]. It becomes especially critical nowadays, with the constant increase of connected devices.

Over more than 40 years, the efforts to increase transaction throughput lie on amendments of ancient mechanisms proposed in early 70s. In [85], Michael Stonebraker alerts about the data tsunami and shows to the community the necessity of rewriting the traditional DBMS architectures. Across the recent years traditional and contemporary DBMS are systematically making improvements and embedding new features in order to fit non-functional requirements, such as, performance, scalability, and robustness (the proper definition for non-functional requirements is given at Section 2.4.1).

On the one hand, traditional DBMS partially tackle several non-functional requirements with replication and database partitioning schemes in a disk-oriented architecture. This architecture is strongly based on three main features: (1) complex multi-threaded transaction processing schedules, (2) distributed database consistency protocols, and (3) extensive performance tuning (i.e., manual tuning knobs) to make the DBMS deliver high transaction throughputs. On the other hand, NewSQL appears as the most prominent alternative to deliver high throughputs on transaction

processing yet enforcing strong database consistency through the ACID properties. NewSQL presents a totally new architecture described later in Section 2.3.

2.2.1 The Tuning Knobs

Traditional DBMS were developed in a time when computing resources were expensive, making the tuning process specially important and supported by highly specialized professionals. Back to that time, the cost of human resources (HR) was cheap and computational resources were expensive. Nowadays, it is the other way around and HR is the major expense in the IT market [85].

Another striking problem is related to carrying out manual performance tuning. In major DBMS there are hundreds of tuning knobs[1,2] and finding the optimal configuration is a tedious and error-prone task. Small companies cannot afford hiring experts; for large companies, not even an expert can always figure out the optimal tuning [92]. In order to mitigate these problems, several approaches for DBMS self-tuning were proposed over the years [92, 101, 87].

In [101], the authors claim that specialized configuration of tuning knobs can avoid inconvenient outages during peak loads. More than that [87], there is no single configuration of the system to adequately handle every possible workload.

In this context, testing can be useful to validate whether a tuning setup reflects in boosting or spoiling performance for a given workload.

2.3 NewSQL

The NewSQL is a distributed in-memory database management system, which preserves the ACID properties and supports SQL language. In this thesis, we stick to the architecture of VoltDB [98], since it is similar to the other NewSQL, such as H-Store [56], NuoDB [65], MemSQL [63].

[1] A list of tuning knobs for PostgreSQL can be found here: https://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server

[2] A list of tuning knobs for Oracle can be found here: http://www.oracle.com/us/products/applications/aia-11g-performance-tuning-1915233.pdf

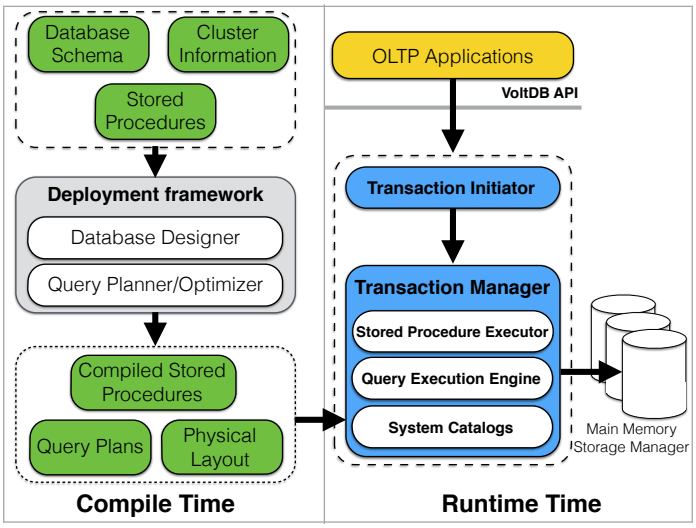


Figure 2-3: VoltDB architecture [99]

The VoltDB architecture (see Figure 2-3) is composed by two components. The "compile time component" is all related to the database setup, such as configuration files, schema definition, available resources, stored procedures' compilation. The "runtime time component" looks after the operating activities, such as, transaction management, resources management, data storage, data manipulation, and concurrency control.

Briefly, NewSQL takes a different architecture to tackle non-functional requirements by keeping entire databases in-memory and processing transactions through a multiple single-threaded scheme in distributed database partitions. Compared to traditional DBMS, another crucial point is its self-tuning capabilities [85].

In VoltDB, the concept of transaction is implemented as stored procedures. The stored procedures combine SQL statements and the features of a programing language. Figure 2-4 shows an example of stored procedure implemented in Java to proceed a flight reservation. First, the flight reservation procedure ensures that the flight number is valid to then execute and commit in the database. If the number is not valid, the transaction is aborted and data is rollbacked to the last consistent state. By using stored procedure, VoltDB guarantees the consistency of the database. From now on, we will refer to stored procedure as transaction.

```
final String getflight = "SELECT FlightID FROM Flight WHERE FlightID=?;";
final String makeres = "INSERT INTO Reservation (?,?,?,?,?,?);";

...

public VoltTable[] run( int servenum, int flightnum, int customernum )
   throws VoltAbortException {

   // Verify flight exists
   voltQueueSQL(getflightsql, flightnum);
   VoltTable[] queryresults = voltExecuteSQL();

   // If there is no matching record, rollback
   if (queryresults[0].getRowCount() == 0 ) throw new VoltAbortException();

   // Make reservation
   voltQueueSQL(makeressql, reservnum, flightnum, customernum,0,0);
   return voltExecuteSQL();
}
```

Figure 2-4: VoltDB: stored procedure [97]

2.3.1 Partitioning

Partitioning techniques can improve performance by promoting I/O parallelism on disk operations. According to [80] *"In its simplest form, I/O parallelism refers to reducing the time required to retrieve relations from disk by partitioning the relations over multiple disks."*

Although NewSQL is not a disk-oriented DBMS, partitioning also plays an important role for high transaction throughput. Instead of promoting I/O parallelism, the goal of partitioning in NewSQL is to span transactions across distributed machine nodes.

According to [54] *"The challenge becomes dividing the application's data so that each transaction only access one partition ... TPC-C benchmark can be partitioned by warehouse so an average of 89% of the transactions access a single partition"*.

For instance, in VoltDB relations are horizontally partitioned (see Figure 2-5) and allocated in different machine nodes. These partitions are automatically defined by VoltDB or by the database administrator (DBA), who identifies data dependencies and creates the partitions. With partitions allocated separately, the burden of transaction processing is balanced across the distributed machine nodes and the system is less prone to generate locks or CPU latches. In the next section, we describe how NewSQL DBMS accesses the partitions.

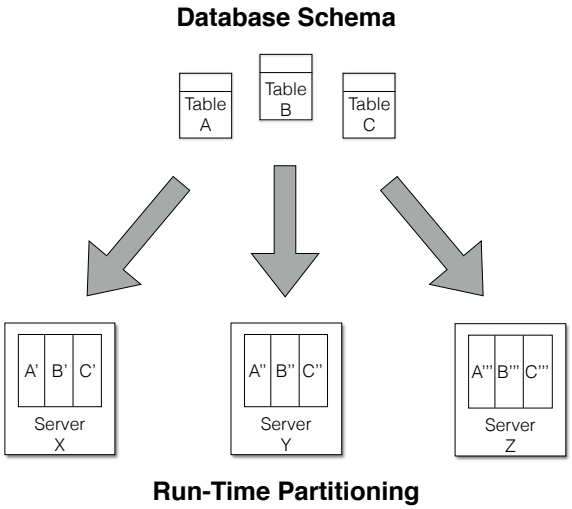


Figure 2-5: VoltDB partitioning [97]

2.3.2 Multiple single-threaded transaction processing scheme

Single-Threaded Transaction Processing schemes are also used to avoid serialization protocols, locking and CPU latches [99], which represent together 96% of performance overhead in OLTP (i.e., only 4% of useful work!) [55]. This scheme is based on an execution queue to host incoming transactions and consume them one-at-a-time. Figure 2-6 shows the transaction queue.

In order to improve transaction throughput, NewSQL embraces the multi single-threaded transaction processing. This means that the execution queue dispatches transactions to be processed at different machine nodes (assuming a distributed running environment). Every machine node is allocated as a single-threaded process.

However, one transaction can be divided into fragments that updates different partitions. In this case, transaction processing can span across multiple machines. For that, NewSQL presents two transaction processing modes [54]: "*Single Partition Transactions*" (SPT) and "*Multi-Partition Transactions*" (MPT).

As emphasized in [54] "*... an average of 89% of the transactions accesses a single partition*". The SPT processing looks after consistency for such "89%" of the transactions. Transactions are single-threaded processed in an individual node. Therefore, no hassles with locking or CPU latches. Moreover, the rest of the machine nodes are

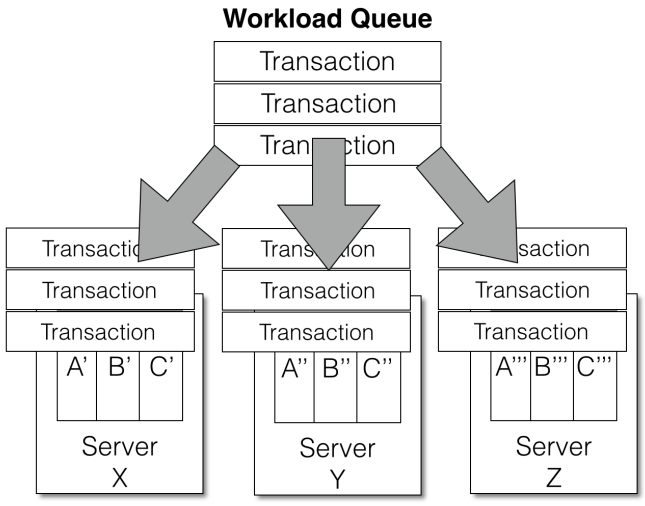


Figure 2-6: VoltDB transaction processing [97]

free to receive other transaction requests.

The MPT processing mode looks after the remaining "11%" of the transactions. In this mode, one machine node acts as the transaction coordinator and piggybacks the two-phase commit protocol (2PC). As in the disk-oriented DBMS, the 2PC requires a first phase of spanning transaction fragments across the slave nodes and acknowledging their response. The second phase starts when the coordinator receives all the responses. Then, it spans commit messages and waits for the final result.

2.3.3 The "No Knobs" Tuning Operation

As we presented in Section 2.2.1, the traditional DBMS are based on a huge number of complex tuning knobs. The DBMS vendors and the research community are investing efforts to provide mechanisms for self-tuning in order to minimize the human intervention. In [85], the authors claim that the self-tuning proposed by the DBMS vendors does not produce results anywhere near compared to skilled DBA. They propose to rethink the tuning process by creating DBMS with no visible knobs.

The idea of "No knobs" operation is not new and it is described as a major goal for contemporary DBMS: " ... *databases must be self-tuning. There can be no human-settable parameters, and the database system must be able to adapt as conditions*

change. We call this no knobs operation."[12]. In order to achieve this type of operation, VoltDB implements a new database design mechanism which automatically specifies partitioning, replication and indexed fields. The goal is to maximize single-partitioned transactions.

The partitions are spread across the cluster based on the primary key of the main table and assigning tuples of other tables to nodes based on the tuples of the main table they descend from. The horizontal partitioning and indexing options might also be specified by the DBA.

In addition, the DBA no longer needs to tweak tuning parameters as traditionally carried out in DBMS, such as PostgreSQL and Oracle. For instance, in PostgreSQL the maximum allowed concurrent connections is set by the knob *MAX_ CONNECTIONS*. Instead, VoltDB links the number of concurrent connections to the available resources of the operating system.

2.4 Software testing

Software testing consists of verifying whether the system behaves as expected in a given situation [41]. Software testing can also be defined as the exercise of finding defects in a software system, never the absence of them [13]. This task can be performed manually, but the main objective relies on an automated approach to implement a specific test design. In this sense an abstract model plays an important rule in order to provide repeatable means to generate a test suite. The exercise of software testing may investigate two main requirements: functional and non-functional. The requirements are related to the behavior aspects of a system [61].

2.4.1 Functional and non-functional requirements

According to the IEEE glossary [89], a functional requirement is "*A requirement that specifies a function that a system or system component must be able to perform*". We can generalize this definition as the capability of a system to provide the correct output for a given input. Thus, functional testing targets whether the System Under

```
insert into names ("Jorge");

commit;

select name from names where name = "Jorge";
```

Figure 2-7: Example of functional testing

Testing (SUT) implements the functionality for a specific requirement. Let's consider a database composed by only one table "names" with two attributes: "identification" (auto incremented integer) and name (string). A simple functional test to ensure the data durability might be proceeded by inserting a new "name" in this table and then trying to recover the same data. In this case a defect is characterized by a wrong answer or an empty field as result.

There is no consensus regarding to the definition of non-functional requirements, though [43, 9]. Several authors proposed different definitions along the last two decades [76, 64, 81]. Basically, non-functional requirements are related to "property, quality, aspect or constraints" with which the system must meet, such as performance, reliability, robustness, scalability, compatibility, usability, resilience, recovery [9]. Moreover, non-functional requirements are highly influenced by the environment where the system is allocated [42]. For instance, response time may increase fast during peak loads, impacting on the quality of service.

As an example, let's consider a DBMS configuration file, which allows tuning the system for a given workload. Lets take into consideration an hypothetical tuning knob in this file related to the maximum number of concurrent connections "max_conn". This knob ensures the maximum number of concurrent transactions treated by the DBMS[3]. A simple non-functional testing consists of setting up this knob and then submitting concurrent connections to and beyond the set limit (see Chapter 3). Once the system is unable to ensure the pre-set maximum number of concurrent connections it is considered a defect. This kind of defect has significant impacts on the quality of service, such as gradual performance loss. In our context, non-functional testing has an essential role to evaluate the SUT behavior under stress conditions.

[3]We simplified the configuration for concurrent connections for the sake of the example, but the proper configuration relies on more than one knob (see Chapter 3)

2.4.2 Stress Testing

Stress testing is a specific non-functional testing widely applied, not only in software engineering, but in several different fields of knowledge, for instance: medicine, engineering and economics:

Medicine [73]: *"... preferred stress modality in patients who are able to exercise to an adequate workload (at least 85% of age-adjusted maximal predicted heart rate and five metabolic equivalents) ..."*.

Engineering[4]: *"... We define a stress testing as a targeted reassessment of the safety margins of nuclear power plants in the light of the events which occurred at Fukushima: extreme natural events challenging the plant safety functions and leading to a severe accident ..."*.

Economics [14]: *"... Stress testing did not originate in finance, but in engineering. In its broadest sense, stress testing is a technique to test the stability of an entity or system. In finance, it was originally used to test the performance of individual portfolios or the stability of individual institutions under especially adverse conditions (micro stress tests). More recently, similar techniques have been employed to test the stability of groups of financial institutions that, taken together, can have an impact on the economy as a whole (macro stress tests) ..."*.

It is possible to highlight similarities between these examples with a stress testing in software engineering context. In [100], the authors present a step-up and step down methodology. The goal is to compare the effectiveness of these different approaches in an accelerated life tests (ALT) for components and materials. ALT is used to expose the system under extreme conditions in order to find failures in a short amount of time. The step-up stress ALT, is a special kind of ALT in which the "material limit" is reached step-by-step over time (up to the highest stress). Alternatively, the step-down stress ALT, the highest stress is used at first moment, then it is decreased step-by-step over time to the lowest level. In our context the system life time is not reachable in fact, such as in a material (i.e., electronic components), but it is possible to enforce the system thrashing.

[4] "Stress tests" specifications - http://www.wenra.org/

In [66], the authors present a study of the prognostic value of exercise stress test variables in elderly patients with coronary atherosclerosis and exercise-induced ischemia. The American Heart Association recommends the exercise stress test for sedentary elderly people before beginning a rigorous physical activity in order to identify coronary atherosclerosis (i.e., hardening of the arteries). The exercise stress test consists of five stages of progressive inclination each with a three minute duration and a speed of 3.6 km/h. For the next five stages, the speed was increased to 4.8 km/h. The objective is to attest the patient capacity to practice physical activities. In our testing context, a step-up stress approach applied in software engineering can validate the"system" capacity during heavy workload.

In economics [69], the authors describe the stress test as "... a technique that measures the vulnerability of a portfolio, an institution, or an entire financial system under different hypothetical events or scenarios". The stress test is used to evaluate two different aspects of financial institutions: solvency and liquidity. Basically, it is a "what if" exercise. The evaluation is based on the financial institution response in delicate situations, such as "If a large amount of deposits is withdrawn suddenly or funding markets (such as repos and commercial paper) freeze, the bank might face a liquidity shortage even if it is otherwise solvent". In our context, the similarity is related to answer unexpected situations, such as workload spikes.

In software engineering, stress testing is designed to impose a heavy workload to ensure the reliability of the system. For instance, by submitting a large number of HTTP requests or database transactions concurrently, to check for defects that impact on performance stability. Normally, the workload must go far beyond the system limits [41].

2.4.3 DBMS testing

In DBMS, performance testing validates the system from different angles. Commonly, validation is executed through a benchmark pattern to reproduce a production environment. Different benchmark approaches were presented along the last decades, but focusing on providing comparison metrics (e.g., response time, through-

put, and resource consumption) [53] rather than software testing, such as: Debit/-Credit [35], AS3AP [93], TPC-like [2], SetQuery [67], the cloud data service benchmark YCSB [19], and a data-analysis benchmark for parallel DBMS [72].

The TPC-B benchmark is a stress benchmark approach proposed by TPC for traditional DBMSs. It is presented as a stress approach in the sense that they aim to validate the integrity of transactions upon significant disk processing (i.e., I/O operations). The execution turns around after the SUT reaches its steady conditions, which is the performance boundary of TPC-B.

Designed for large-scale databases, the YSCB benchmarks three main non-functional requirements of distributed database stores: (1) Performance; (2) Scalability; (3) Availability. The YSCB includes a load generator to allow benchmarking different database stores, including relational and key-value. However, the load generator does not include fault-injection nor stress workloads. As well as TPC-Like benchmark, it relies in constraints of performance to ensure measurements in stable condition.

The R-cubed [106] benchmark was designed to assess availability of computational systems in general that was further extended to a benchmark suite called the System Recovery Benchmark (SRB). The suite bundles five different benchmarks: (1) Cluster - SRB-X; (2) Application - SRB-D; (3) Databases - SRB-C; (4) Hard disk (RAID) - SRB-B; (5) Operating system - SRB-A. However, just two of them were implemented to date (i.e., SRB-X and SRB-A).

DBench-OLTP [96] benchmark was also designed to assess availability of transactional systems. It broadly uses the TPC-C specification with two main extensions in order to mimic real system problems: fault-loads and measures related to system dependability (e.g., mean time to detect errors).

Several recent storage systems were developed and tested through similar approaches. In [21, 83, 82, 23], validation is performed by the TPC-Like benchmarks. In [18], the authors present PNUTS, a massively parallel and geographically distributed database system for Yahoo!'s web applications. They provide their own performance testing methodology focused on latency metrics with a very simple own workload supported by a centralized architecture.

Performance testing tools such as Hammerora [1], Oracle Application Testing Suite [6], and AppPerfect [7], provide a test driver to submit operations also based on a TPC-like benchmark. Their goal is to allow engineers writing test cases to mimic load conditions. The drawback is that the test cases are way too specific and cannot reflect a far more aggressive real-world production environment with workload spikes and shifts after a while in steady condition state [82, 86].

Focusing on functional assessments, the Agenda [30] tool provides a testing methodology to validate ACID properties of the DBMS. Therefore, Agenda does not generate test cases to tackle non-functional requirements.

Taking into consideration database applications, deadlock situations are explored in [48, 49]. Deadlocks are characterized as the situation when more than one instance of execution tries to lock and use the same resource, causing loops in the transaction schedules [17]. The authors propose an approach modeled by a lock graph that detects the potential loops. Once detected, a delaying mechanism is applied to break the deadlocks.

We also mention testing frameworks developed to ensure functional requirements, which may be applied in different systems, including Database Systems. Peerunit [25] is a framework to test distributed systems supported by a distributed testing architecture. Peerunit aims to test the system behavior, in terms of functionality, under different conditions of volatility. On the other hand, JUnit[5] uses a centralized architecture to carry out Java unit tests. Junit is widely used to perform functional testing.

Table 2.1 summarizes the DBMS testing approaches. We take into consideration four characteristics of each approach: The workload regime used, the targeted system requirement, the testing architecture and the requirement type. The limitations of the related approaches are presented in the section 2.6.

[5]http://junit.org/

Approach	Workload regime	Target	Testing architecture	Requirement
Das et . al. [23]	YCSB	Performance	- *	Non-functional
Soundararajan et. al [83]	TPC-like	Performance	Centralized	Non-functional
Curino et. al. [21]	TPC-like	Performance	- *	Non-functional
PNUTS	Own mixed workload	Performance	Centralized	Non-functional
DBench-OLTP	TPC-like	Performance	Centralized	Non-functional
HammerDB	TPC-like	Performance	Centralized	Non-functional
AppPerfect	TPC-like	Performance	Centralized	Non-functional
Agenda	TPC-like	ACID properties	Centralized	Functional
Oracle Application Testing Suite	TPC-like	Performance	Centralized	Non-functional
Peerunit	Key-value	data-manipulation	Distributed	Functional
JUnit	-	System specifications	Centralized	Functional

Table 2.1: Comparing DBMS testing requisites

*It is not clear whether used a centralized or distributed architecture.

2.5 Model-based Testing

Model-Based Testing (MBT) is a technique to automate the test case generation based on behavior models. The behaviors of a system, according to IEEE standards association [89][51], are related to the requirements and defined as following:

- a condition or capability needed by a user to solve a problem or achieve an objective.
- a condition or capability that must be met or possessed by a system, system component, product, or service to satisfy an agreement, standard, specification, or other formally imposed documents.
- a documented representation of a condition or capability as in 1 or 2.
- a condition or capability that must be met or possessed by a system, product, service, result, or component to satisfy a contract, standard, specification, or other formally imposed document. Requirements include the quantified and documented needs, wants, and expectations of the sponsor, customer, and other stakeholders.

2.5.1 The elements of MBT

According to [22], there are three key elements in MBT approaches: the model's notation, test generation algorithm and the testbed. Figure 2-8 illustrates these elements.

The model's notation can be supported by several different languages, graph, mathematical model, etc. The most popular notations used by model-based approaches are: UML and FSM. On the one hand, UML is a graphical modeling language composed by a set of diagrams, such as statechart diagram, class diagram, sequence diagram, test case diagram, etc. This language becomes a de-facto standard for object-oriented software design [37] as well as a widely used notation for model-based testing. On the other hand, the FSM is a mathematical model used to model

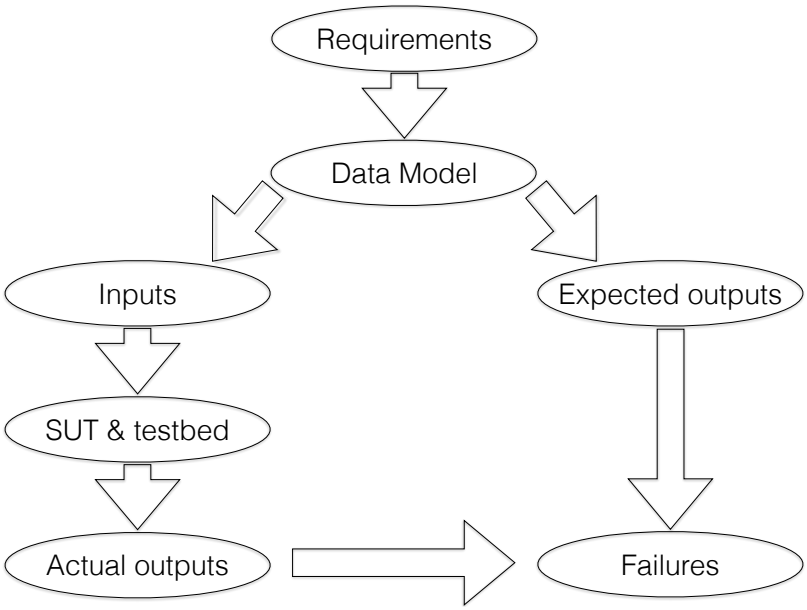


Figure 2-8: MBT elements [22]

systems in several areas [58], such as communication protocols [103], sequential circuits [50], pattern matching [95], etc. A deterministic FSM is described as a quintuple composed by an alphabet (i.e., inputs), a finite set of states, an initial state, a set of state-transitions and a set of final states.

2.5.2 The MBT approaches

MBT approaches are widely applied in software testing to validate functional as well as non-functional requirements of the SUT [31]. In [71, 32, 75] we have examples of approaches focusing on security requirements for reactive systems based on the environment description as a model. Efficiency requirements are explored in [60, 78]. These approaches are respectively based on TRIO specification and UML model to attest real time and object-oriented software. Examples of approaches focused on functional requirements are treated in [84, 38, 5, 8], all of them using FSM as behavior model.

The following approaches are presented with more details. They address non-functional requirements, more specifically, performance.

Garousi et al. [40] proposed a stress testing approach based on a UML model

to identify performance properties when data traffic on a network is maximal. By using the specification of a real-world distributed system, this approach can help to increase the probability of exhibiting network traffic-related faults in distributed systems. The authors argue that the proposed approach is more effective when compared with approaches based on operational profiles, in the matter that the probabilities of exhibiting network traffic-related faults are increased.

In [33], an "Automatic Performance Testing Method Based on a Formal Model for Communicating Systems" is presented. The goal is to check two performance requirements: throughput (in terms of number of users attended) and response time (requests to be processed within a second). The authors argue for a stress testing, in the sense that the throughput is raised to the maximum number of requests that the system is configured to deal, at the same time.

Shams et. al. [79] proposed "A Model-Based Approach for Testing the Performance of Web Applications". The authors use a FSM to model a set request sequences and then be able to create synthetic workloads with specific characteristics. They consider the impact of inter-dependent requests by the same user, this means, requests that depends the response of previous requests. They claim as the major benefit is the ability to create several specialized workloads that reflects the typical interactions of users with web applications.

In [105], the authors proposed a stress testing to analyze the modules dependencies of the SUT. Using an initial input, a FSM models the interaction of the modules of certain system. By manipulating the input and the interaction of the states of the FSM, the approach aims to model the best stress state representation for a given input.

The "Stress Test Model of Cascading Failures in Power Grids" [59] describes a model-based testing for power systems. Mainly, the approach force security constraints by applying stress conditions. The stress conditions are related to a sequence of failures (i.e., outages). The idea is to test the operating limits of power systems.

Barna et. al. [10, 11] proposes a "Autonomic Load-Testing Framework" for performance testing in components of software and hardware. By using a Layered queu-

ing model, the approach triggers bottlenecks related to processing capacity limits of resources. These bottlenecks are specially noticeable in components which queued requests, for instance web based application limited by a fixed number of threads for new connection requests. As soon as the queue reaches this limit, the queue becomes a bottleneck.

The "WALTy: A User Behavior Tailored Tool for Evaluating Web Application Performance" [77] uses a Customer Behavior Model Graph (CBMG) to generate specific workloads based on user profiles. The CBMG is created using log files, The tool is composed by two modules: First, the Builder Module which extract the information from the CBMG; Second, the Traffic module, responsible for traffic generation. The authors falsify the tool as a web mining workload generator.

In [3], the "MBPeT: A Model-Based Performance Testing Tool" is presented. This tool evaluates system performance based on loads generated in real-time execution. By monitoring different performance indicators and using a Probabilistic Timed Automata (PTA) as model, the authors seek for performance indicators for given workload, such as response time.

TestOptimal [88] is a commercial tool developed to provide a model-based testing on top different behavior models (i.e., UML, FSM, Extended FSM, Control Flow Graphs) and covers functional as well as non-functional testing (Performance testing). The developers argue a test case generation and test automation tool.

The main issues in the presented approaches are related to the limitations in expose the system to stress conditions gradually and represent the system based on its performance behaviors. Most of them may be applied for DBMS to assess quite specific points related to performance or functional requirements. For instance, create specific workloads to reach the exact amount of resources set by the tuning knobs [33]. However, it cannot reproduce stress without exceeding the limits imposed by the knobs.

Table 2.1 summarizes the MBT approaches. The comparison is based on 3 characteristics: The testing requirement assessed by the approach, the system under test and the behavior model used to generate the test cases.

Table 2.2: *Comparing Model-based testing techniques*

Approach	**Requirement**	**Target System**	**Behavior Model**
Garousi et al.[40]	Performance (Stress)	Distributed system	UML
Barna et. al. [10]	Performance	Software/Hardware components	Layered queuing
Liao et. al. [59]	Performance (Stress)	Power systems	UML
Yang et. al. [105]	Performance (Stress)	System modules interaction	FSM
Shams et. al. [79]	Performance	Web Applications	FSM
Eros et. al. [33]	Performance (Stress)	Communicating Systems	FSM
WALTy [77]	Performance	Web Mining	Graphs
MBPeT [3]	Performance	Web Applications	Probabilistic Timed Automata (PTA)
TestOptimal	Functional/Non-Functional	Any	UML, FSM, Graphs

2.6 Conclusion

Concerning the DBMS testing approaches, we can enumerate three limitations that avoid carrying out stress testing of DBMS, with special attention to NewSQL. The limitations are:

1. Centralized architecture: Centralized architectures present two problems: First, the testing scalability is limited by the resources of the testing machine; Second, the testing machine interferes with performance [62].

2. TPC-like workload regime: Most of the approaches focus on performance comparison based on the TPC-like workload regime, which imposes constraints of performance to ensure stable conditions. This creates limitations to exercise the system under stress workloads and to find defects.

3. Model-based approach: There is no approach to test non-functional requirements of the DBMS supported by a system model. It hinders the testing repeatability and the comprehension of the different behaviors presented by the DUT.

Moreover, we claim that the presented model-based testing approaches do not focus on performance requirements and defects related to stress workloads of real-world environments. Mainly, we argue they are not able to conduce the system to its performance limits in face of high concurrency environment. Mostly, the related approaches focus on specific test cases to ensure performance requirements. The approaches focused on stress testing are specifically designed to conditions not related to increasing workloads and performance loss, but special test cases to simulate punctual crashes [59] or peak loads [40].

Thus, to our best knowledge, the existing testing approaches can not deal with the challenges raised up by the high concurrency workloads and the new high-throughput databases, such as boundaries of performance and the specific behaviors in stress conditions. There are many aspects missing so far. An appropriate approach must be able to relate potential defects and performance loss to specific system behaviors.

Chapter 3

Stress Testing Methodology

3.1 Introduction

In this Chapter, we present STEM, a Stress Testing Methodology for exposing workload-related defects in traditional DBMS. In the specific case of workload-related defects, we claim that the defects can only be revealed from the combination of stressing workloads to performance mistuning (e.g., insufficient admission control). In STEM, we progressively increase the transaction workload along with several tuning adjustments (a type of step-stress testing [104]). In addition, STEM takes into account the various biases that may affect the test results, including the influence of the operating system and the testing coordination.

Besides, STEM leverages distributed testing techniques in order to generate stressing workloads and coordinate the stress testing campaign. This distributed architecture allows increasing the workload by several orders of magnitude up to stress conditions. We validate our methodology with empirical studies on two popular DBMS (one proprietary, one open-source) and detail defects that have been found.

The rest of the Chapter is structured as follows. Section 3.2 presents our research questions. Section 3.3 presents our stress testing methodology. Section 3.4 describes our validation through experimentation. Section 3.5 concludes this chapter.

3.2 Stress Testing for DBMS

Our methodology for stress testing DBMS is systematically and empirically evaluated, taking into consideration following research questions:

RQ0: Does the incremental methodology exercise distinct parts of the code?

- By presenting evidence that more source-code parts are exercised compared to the state of the art in functional testing, through code-coverage or log file analysis. The evidence can be presented in terms of blocks of code ignored by functional testing (e.g., loops, exceptions or conditions).

Stress testing requires an incremental test approach that drives the DBMS from an initial configuration setup towards a production setup in order to test the functionalities related to transactional workloads. Therefore, we require to gradually increment the workload up to a stress condition, which is beyond the DBMS's state of usage in a real workload environment. The objective is to search for defects related to different scales of concurrent transactions. Suppose that a DBMS is directly tested upon stressing load conditions instead of using an incremental test approach. The results will not show whether the defect appears at smaller scales. In fact, different types of defects appear while a system treats different workloads, because different parts of the source-code are exercised [27]. We can verify that different parts of the code are exercised either by code coverage analysis or by analyzing the message entries from the log files. The earlier is straightforward by executing the DBMS with its code instrumented by some coverage tool (e.g., GNU/gcov, EMMA). The latter treats the DBMS as a black box and requires to monitor both the response time on transactions and the entries from the log files. Indeed, a higher response time is expected, since the DBMS deals with more transactions and rejects the ones that overtake its configured capacity. Then, the DBMS issues more log entries informing rejected requests or errors that can be analyzed to prove that more functionalities were exercised, for instance, comparing the entries issued from different test steps.

RQ1: Does the OS impact on test results?

- The answer can be provided in two ways: (i) analyzing the resource consumption of the OS, providing evidences that such consumption works as expected (e.g., the consumption remains steady along testing); (ii) comparing the test results on different OSs (e.g., Linux, BSD, Windows) and providing evidences that the detected defect is the same in all of them.

Beyond the test driver, other software parts are not supposed to interfere with the test results, especially the OS. We make two assumptions when the same defect is exposed on different OS: (i) the OS is robust, and (ii) the OS is correct. These assumptions demand to continually monitor the OS health (i.e., CPU, memory, I/O, swapping, and networking). The main objective is to ensure that the OS responds to any possible request to the DBMS until the test ends.

RQ2: Does the DBMS use all the allocated resources upon stress workload?

- By comparing the inconsistencies between the declared DBMS setup and the effective usage of the DBMS resources.

This last research question is related to the DBMS defects exposed by the stress workload. Since we assume the OS to be robust and the DBMS to be tuned, the defects are the major cause for preventing the DBMS from allocating all the available resources. This also requires to monitor the health of both the DBMS and the OS.

3.3 STEM

In this section, we describe STEM, our methodology for stress testing transactional DBMS. STEM is based on the joint increment of two variables, concerning the DBMS knobs on one hand and the workload on the other hand. The DBMS knobs correspond to the setup parameters to be tuned for a workload. The workload corresponds to the number of submitted transactions and how they are submitted. The variation of these *dependable variables* is necessary for testing the SUT under different angles in order to find defects related to different workload levels and more specifically, defects related to the performance degradation boundaries.

3.3.1 Dependable Variables

The number of knobs both in the DBMS and OS is rather large. We only focus on the DBMS knobs since we assume that the OS has been completely tuned up to the hardware capacity. In fact, we pay special attention to the knobs that are impacted the most by transactional load spikes, including the buffer pool management ($work\_mem$), and the admission control (max_conn). Other knobs can also be explored, including group commit, and log flushing, however, we do not focus on them due to space constraints.

The DBMS knobs can accept a large domain of values, from their minimum up to maximum capacity. For the sake of clarity, we assume that each knob can only have three values related to the DBMS capacity: min, med, max.[1] To setup these values, we follow the rules of thumb from the DBMS documentation. Once the DBMS is tuned, the workload is created based on its capacity. Moreover, the workload is related to the way the tests are executed. To mimic real distributed clients and reproduce a stress workload, it is required to deploy many distributed testers, denoted by T, for submitting a large number of transactions. The number of distributed testers depends on the size of the workload (see Table 3.1).

Workload	**Condition**
Min	if equal to declared max_conn, with $\|T\| = 1$
Med	if equal to $\|T\| * max\_conn$, with $1 < \|T\| \leq 10$
Max	if equal to $\|T\|^2 * max\_conn$, with $1 < \|T\| \leq 10$

Table 3.1: Workload setup parameters.

The minimum size workload (min) is set equally to the max_conn configuration and executed by one tester, denoted by $t \in T$. The objective is to establish the performance baseline due to low concurrency. The medium size workload (med) is set to $|T| * max\_conn$ to increase the degree of concurrency. The objective is to evaluate the robustness of the DBMS. The maximum size workload (max) is set

[1] The process of identifying the variables and then decomposing their input domains into equivalent classes (from a testing perspective) is called "category-partition testing" in the software testing community [68].

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

Table 3.2: STEM's Execution Sequence.

to $|T|^2 *$ *max_conn* to stress the DBMS. In addition, the DBMS server and the tester machines must be separated from each other to avoid any interference on the performance results.

3.3.2 Execution Sequence

The execution sequence consists of many steps to drive the DBMS from an initial state up to the stress state (i.e., the incremental test approach). The steps are linked to many setup combinations for each knob. This consists of running the DBMS with all values assigned to dependable variables, which leads to 3^3 setup combinations of *work_mem*, *max_conn*, and *workload*. We used pairwise testing[2] to narrow the number of variations to 9 combinations. Moreover, we verified whether some of these combinations are considered useless, for instance, a DBMS must be started with a sufficient amount of memory to handle a large number of concurrent connections. That is, we cannot set *work_mem= min* and *max_conn=max*. As such, the final number of combinations was set to 5 (see Table 3.2) and form the basis of the steps taken by STEM, which are:

1. Initial DBMS setup upon initial workload;

2. Tuned buffer pool size upon stress workload;

3. Tuned DBMS upon a minimal amount of transactions;

[2] *"Pairwise testing is a combinatorial method in which all possible pairs of parameter values are covered by at least one test [57]."*

4. Tuned DBMS upon stress workload up to the DBMS performance limit;

5. Tuned DBMS upon stress workload beyond the DBMS performance limit (up to the performance degradation);

The first step aims at seeking for defects related to any DBMS functionality, installation defects, and misconfiguration, not necessarily related to the DBMS installation package. The goal of the second step is to establish the degradation baseline for the DBMS, which is done by setting the admission control to its minimum. The third step aims at validating functional aspects after a tuning phase. For instance, to validate whether the observed value along testing corresponds to the one expected after tuning. The purpose of the fourth step is to search for defects related to the DBMS performance limit (not yet the stress condition). To establish the DBMS performance limit, we monitor the health of the DBMS and compare the results with the performance baseline established in the second step. The objective is to avoid pushing the DBMS beyond such limit that defines the boundary of the medium size workload. The goal of the fifth step is to push the DBMS beyond its performance limit, reproducing stress load conditions.

The second, fourth, and fifth steps must be executed in a distributed manner to reproduce a large-scale environment. In this context, the test driver cannot have any scalability issue for managing a large number of transactions without perturbing the test execution [24].

3.3.3 Database Specification

In STEM, the database schema and the transaction specification follow the TPC-C benchmark. TPC-C provides 9 tables that implement a wholesale supplier application and 5 types of transactions including: entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. Transactions are submitted by emulated users and the results are displayed at Remote Terminal Emulators (RTE). TPC-C also specifies the pacing of transactions by the emulated users.

We couple our test driver with a distributed implementation of the TPC-C benchmark [28]. We chose this specific implementation, since it was implemented on top of peer-to-peer technology. Therefore, our test driver is allowed to scale-up the number of the emulated users.

3.3.4 Testing Architecture

The general distributed test driver architecture is composed of a test controller and distributed testers. The role of the controller is to synchronize the execution of test cases across the distributed testers (i.e., test drivers). The role of the testers is to invoke the instructions described within the test cases at the interface of the SUT. The instructions are typically calls to the SUT interface wrapped up in test case actions (e.g., transactions).

Testing a distributed system in realistic conditions requires the distribution of the testers. Therefore, the ability to distribute and coordinate a large number of testers multiplies the potential load we can generate to progressively push a system out of its boundaries. This is an important feature that we exploit with this methodology.

In STEM, the execution of the tests is based on the Peerunit[3] testing architecture [24, 27] for reproducing large-scale workloads. Peerunit is deployed on a set of nodes containing one controller and a set of distributed testers. In our implementation, a tester reproduces an emulated user submitting transactions to the DBMS interface. The controller synchronizes the execution of the transactions through a distributed message exchange algorithm deployed across multiple testers.

In Peerunit, test cases are implemented in Java and deployed on all available testers. Each test case is implemented as a class with a set of actions implemented as *annotated* methods, i.e., methods adorned with a particular metatag, or *annotation*, which provides coordination information. During a test, the DBMS is exercised by a sequence of actions. Usually, an action has an input data set and generates an output data set, which can be both stored in text files or in a database. For instance, inserting data into the database and then retrieving such data.

[3] http://peerunit.gforge.inria.fr/

3.4 Experimental Evaluation

In this section, we present an experimental evaluation of STEM. Our objective is twofold: (i) validate STEM with two popular DBMS, and (ii) answer our research questions. In the experiments, we used a leading commercial DBMS, denoted by DBMS-X, and PostgreSQL version 8.3.

We chose an older version of PostgreSQL to validate if STEM is able to reveal defects undetected in an established bug-list.

PostgresSQL is studied first for two main reasons: (i) the source code is accessible making it easier for us to play with it; (ii) the DBMS serves as the storage system of a number of cloud computing solutions where load peaks are expected. Some of these solutions include: Greenplum, EnterpriseDB Cloud Database, and "vPostgres".

During the PostgreSQL tests, we couple the experiments with the GNU/gcov code-coverage tool, to understand the consequences of each step on the execution of the DBMS. The coverage is important to demonstrate that an incremental test methodology helps reaching portions of the code that are usually not exercised with functional tests. The coverage analyzes three main packages: (i) *Freespace* that implements the seek for free space in disk pages; (ii) *Page* that initializes pages in the buffer pool; and (iii) *Manager* that implements the shared-row-lock.

This section is organized as follows. The next subsection describes our implementation in a cluster. Section 3.4.2 describes our experiments based on incremental testing. Section 3.4.3 discusses the influence of experimental variables with the couple PostgreSQL/FreeBSD. Section 3.4.4 discusses the experimental results.

3.4.1 Cluster configuration

All of our experiments were conducted on a cluster machine of the Grid5000 platform[4]. We used 11 "Sun Fire X2200 M2" machines connected by Gigabit Ethernet, where each node was configured with 2 duo-core AMD Opteron 2218 at 2.613GHz and 8GB

[4] `http://www.grid5000.fr`

Step	Number of Transact.	Completed Transact.	Rejected Transact.	Execution Time (sec)	Response Time (sec)	Linux CPU Usage (%)	Linux Memory Usage (MB)
1	100	100	0	1	≈ 0.5	25	100
2	10,000	585	9,415	8	≈ 1	57	100
3	2,000	1,404	596	2	≈ 1	55	300
4	20,000	2,199	17,801	15	≈ 2	87	450
5	200,000	5,583	194,417	51	≈ 5	90	500

Table 3.3: PostgreSQL's Result Overview.

Step	Number of Transact.	Completed Transact.	Rejected Transact.	Execution Time (sec)	Response Time (sec)	Linux CPU Usage (%)	Linux Memory Usage (MB)
1	100	100	0	1	≈ 0.2	50	515
2	10,000	957	9,043	6	≈ 0.8	80	1,000
3	2,000	2,000	0	2	≈ 0.6	70	825
4	20,000	13,479	6,521	8	≈ 1	80	1,000
5	200,000	30,784	169,252	20	≈ 1.1	80	1,000

Table 3.4: DBMS-X's Result Overview.

of main memory. We used one node to run exclusively the DBMS server and ten nodes to run the clients, where each client is managed by a Peerunit tester[5]. To avoid performance interference, the clients and the server were executed on separate nodes, even for the smaller experimentations. To evaluate a possible interference of the OS on the results (Research Question RQ1), the server node ran the experiments with two different OS: GNU/Linux Debian 5.0.9 and FreeBSD 8.2. All client nodes ran GNU/Linux Debian 5.0.9. In all experiments reported, each tester was configured to run in its own Java Virtual Machine (JVM). The cost of test coordination was negligible (see [24] for a complete discussion on test coordination overhead).

3.4.2 Incremental testing

We conduct the experimentation through the 5 steps of STEM for searching load-related defects. Before every step execution, the database is recreated and reloaded to ensure independence between test executions and to be in compliance with repeatable automated testing. Tables 3.3 and 3.4 summarize the results presenting the workload, the number of completed and rejected transactions, the testing execution time, the average response time for a single transaction and the resource consumption of the GNU/Linux OS. Both tables present only the GNU/Linux's results, since DBMS-X

[5]The implementation is available for download at: http://web.inf.ufpr.br/ealmeida/research/tpc-c-c3sl

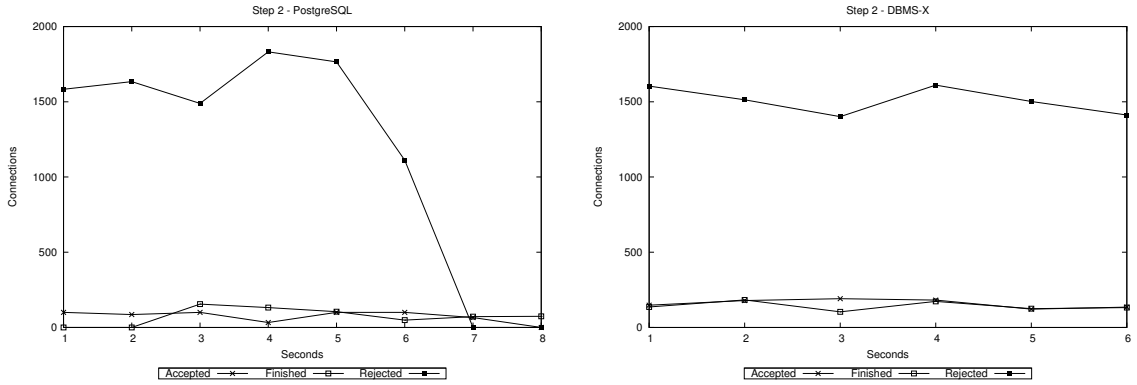


Figure 3-1: Number of Different Connections vs. Elapsed Time in Degradation Baseline Test (Step 2).

presented a steady resource consumption during testing.

3.4.2.1 Step 1

Objective: Ensure correctness of the installation.

Parameters: $|T| = 1$; $Max\_conn = 100$ (default); $Workload = 100$;

The objective of this step is to ensure the proper installation of the DBMS, which remains with its default values. In addition, we test the DBMS functionalities related to the transactional workload (i.e., the classical functional test). The expected result for this test step is the DBMS accepting and completing all the requested transactions.

In PostgreSQL, the default values for *Maximum Connections* and *Workload* parameters are both set to 100, with a similar configuration for DBMS-X. Both DBMS pass this first test step in which 100 transactions are submitted and all of them are accepted and completed successfully, with an average response time of 0.5 seconds for PostgreSQL and 0.2 seconds for DBMS-X.

3.4.2.2 Step 2

Objective: Establish the degradation baseline.

Parameters: $|T| = 10$; $Max\_conn = 100$ (default); $Workload = 10,000$;

In the second step, we allocate a larger amount of memory and workload for searching the performance boundary and the response time degradation. The expected result is the DBMS accepting the concurrent requests up to their setup values and refusing the exceeding ones, occasionally issuing a default error message[6].

In this step, PostgreSQL takes approximately 8 seconds to execute while DBMS-X takes 6 seconds. This execution time relates the execution of the entire test step: the submission of all the requests and the retrieval of all the responses from the DBMS, which can be either a transaction commit/abort or a connection rejection.

During the test, ten testers ($|T| = 10$) submit a number of 10,000 transaction requests. The DBMS were configured to accept at most 100 concurrent requests. Figure 3-1 shows that both DBMS correctly accept a number of requests within their setup value, as expected. In spite of the high concurrency, no accepted transaction is aborted.

In PostgreSQL, we observe that the workload has a direct impact on the code coverage (see Figure 3-5). Indeed, as the workload increases, the code coverage of three packages also increases. First, in the "freespace" package, when the load increases, we observe that PostgreSQL takes more time to find free space. Second, in the "page" package, which initializes more pages. Third, in the "transaction manager" package, where the increase of code coverage is expected due to the larger number of concurrent transactions waiting for the tuples to be unlocked.

In both DBMS, we do not observe any performance degradation. All transactions finish in less than one second. This behavior is also expected, since the raise of the load is not sufficient to overtake any resource of the system, as we will see in Section 3.4.3. Since this is the initial phase of STEM and the DBMS behave as expected, the computed response time becomes the *baseline time* for the subsequent steps.

[6]In PostgreSQL, when the number of requests exceeds the configured value, it issues the following message: *"Sorry too many clients already"*.

3.4.2.3 Step 3

Objective: Ensure the correctness of the tuning knob.

Parameters: $|T| = 1$; $Max\_conn = 2,000$; $Workload = 2,000$;

In the third step, the objective is to ensure that the tuning knobs are correct, based on the results of the previous step. Tuning consists of setting the DBMS knobs to their maximum values, in function of the available resources.

In DBMS-X, little intervention was required to tune CPU, memory and storage. In fact, all the major commercial DBMS provide automatic resource management features to minimize human intervention [4]. A required intervention was tuning the *Maximum Connections* knob to cope with 2,000 concurrent requests. The results show that DBMS-X passes the test accepting and committing all transactions with low response time degradation.

In the case of PostgreSQL, the *Maximum Connections* knob was fixed in 2,000 concurrent connections, which is the expected value for this test. In fact, the setup value of this knob depends on some of the *Buffer Pool* variables, which correspond to the size of the shared buffers used by each new connection. We set the shared buffers parameter with respect to the rules of thumb of the DBMS.

From this third step on, we set the shared buffers parameter to the maximum value of 32,768. Every time this parameter is set, we require to evaluate the correlated kernel parameter *SHMMAX*. This kernel parameter specifies the largest shared memory segment size for both OS, Linux and FreeBSD. According to the PostgreSQL documentation, the following formula is recommended to calculate this value:

$$SHMMAX = (250kB + 8.2kB * shared\_buffers + 14.2kB * max\_conn)$$

In this third step, PostgreSQL took 2 seconds to treat all the 2,000 submitted transactions, preserving the same response time observed at step 2. However, it generated an important number of 596 error messages during the execution: 231 in

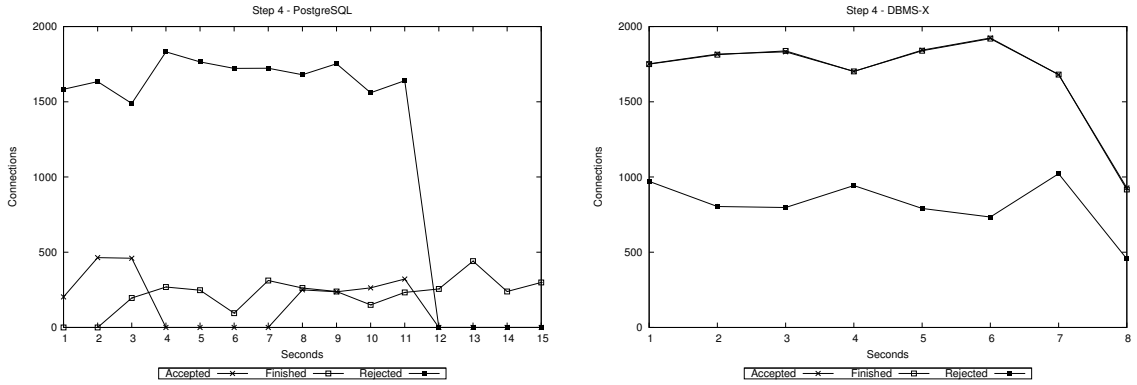


Figure 3-2: Number of Different Connections vs. Elapsed Time Under Robustness Test (Step 4).

the first second and 365 in the second one. We do not observe any degradation on the OS resource consumption. From the functional point of view, this is considered as a *defect*, since the observed number of connections differs from the expected one.

Indeed, we observe an increasing number of new backend processes created by each new transaction request. To create a backend process, besides getting a piece of shared-memory, PostgreSQL manages the state of each process within an array. When it takes a large number of requests, due to the *Maximum Connection* setup, the backend processes fill out the array and eventually overtake its limit. Consequently, the new backend processes are dropped and the objective of *Maximum Connections = 2,000* is never reached (Research Question RQ2).

Another important result is the test boundary established in this step. When $|T| = 1$, the submitted load is bounded by the resources of the lone tester machine, which can only submit 1,000 requests per second without compromising its performance. While we could increase the $Workload$, the results of PostgreSQL would not be different from the ones presented here. To effectively reach a larger number of $Workload$ requests and reach a stress load, a distributed test approach is necessary.

3.4.2.4 Step 4

Objective: Robustness Test.

Parameters: $|T| = 10$; $Max\_conn = 2,000$; $Workload = 20,000$;

In this step, we explore the robustness of the DBMS upon a stress load with *Maximum Connections = 2,000*. As for the second step, the expected result is the DBMS finishing all the accepted transactions with the remaining average response time around 1 second.

From the previous results, we conclude that PostgreSQL cannot handle $2,000$ concurrent transactions. The entire test ran during 15 seconds and the submission of requests lasts up to 11 seconds. Figure 3-2 shows a nonlinear behavior of PostgreSQL, which initially accepts around 200 requests and takes 3 seconds to treat them. PostgreSQL accepts almost 500 requests and treats them in the following seconds. Analyzing the OS health, we diagnosed that the system becomes CPU-bound while handling the submitted load. However, the memory usage remains low, since the accepted requests never reach the expected *Maximum Connections*. At the end, the average response time is twice as much as the *baseline time* (Step 2).

We also diagnosed that the accepted requests produce more modifications in disk pages. When some request performs a modification, PostgreSQL grants an exclusive lock. Then, the backend process, which is responsible for such request, requires to lock other backends, trying to extend the disk pages concurrently. This process is executed at every locking attempt, causing an important number of context swaps, one for each request.[7] From the presented slopes, the number of accepted requests are not too different from Step 2. However, the behavior of PostgreSQL becomes chaotic without any stability in the response time. From the testing point of view, this is a *defect*: PostgreSQL should have a similar behavior as the one observed at Step 2, rejecting the exceeding requests and taking care of the accepted ones.

In DBMS-X, the entire test run in 8 seconds with a stable response time for transactions. DBMS-X finishes all the accepted transactions within an average response time of 1 second (in Figure 3-2 the behavior of the "Finished" slope follows the "Accepted" one). However, DBMS-X only accepts 67.39% of the concurrent requests, never reaching the expected *Maximum Connections*, even with the same resource consumption of the earlier steps. When DBMS-X gets closer to the expected *Maximum*

[7]The requests are managed into the buffer manager.

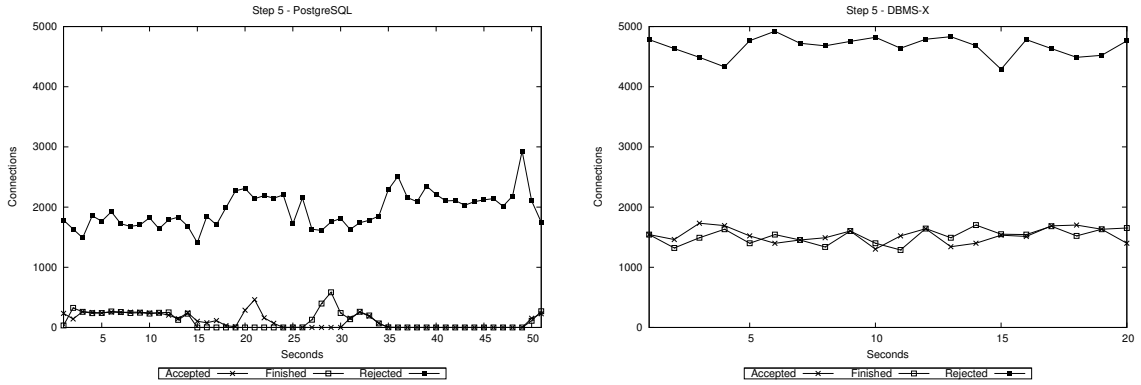


Figure 3-3: Number of Different Connections vs. Elapsed Time Under Stress Test (Step 5).

Connections, it starts issuing a network error message. In DBMS-X's documentation, this message is due to two main reasons: a network misconfiguration (e.g., wrong SQL URL, port, or IP address) or a network backend process issue (e.g., offline process, network port is closed, network or hardware failure). In earlier steps this message is never issued, therefore from a functional point of view, such inability of reaching the expected *Maximum Connections* is a *defect*.

3.4.2.5 Step 5

Objective: Stress Test.

Parameters: $|T| = 10$; $Max\_conn = 2,000$; $Workload = 200,000$;

Figure 3-3 illustrate the elapsed times for the experiments. In PostgreSQL, similarly to Steps 2 and 4, we observe that the number of accepted and finished transactions were much smaller than the rejected ones (2.79% accepted/finished with an average response time of 5 seconds). We notice that it accepts and finishes a uniform number of transactions until the 14th second when a thrashing state begins (i.e. in this state, a large amount of computer resources is used to do a minimal amount of work, with the system in a continual state of resource contention).

In addition, we observe at the 35th second that PostgreSQL enters a failure state and refuses any new connection attempt. From code coverage analysis, we notice that two particular code parts are more exercised compared to Step 4: the free space

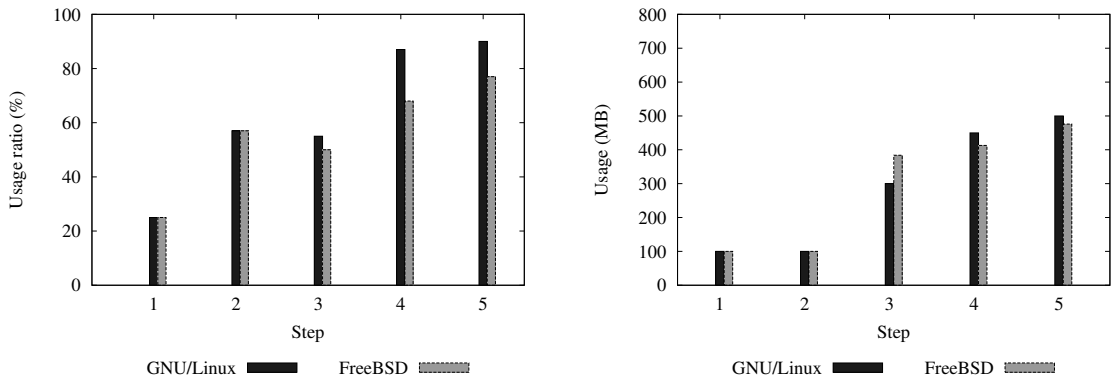


Figure 3-4: PostgreSQL's Resource Consumption at Each Step.

check on pages and the buffer page code. When a transaction is about to update the database, PostgreSQL tries to find free space in data files either by seeking pages with enough free space or extending the data file itself with more pages. At stressing stages, PostgreSQL is not capable of managing a large number of requests for finding such free space. Hence, it starts issuing "InvalidBuffer" messages. This behavior is considered as a *defect*, since the disk has available space with low throughput of 18M of writing operations, which is not too different compared to Step 3 with 14M of writing throughput. Furthermore, this result validates our claim that STEM is helpful to push the SUT to its performance limit and reveal load-related defects (Research Question RQ0).

In DBMS-X, we observe a similar response time compared to Step 4 of 1.1 seconds, even accepting only 15.39% of the requests. DBMS-X shows the same behavior as Step 4 rejecting new requests. It also does not reach the expected *Maximum Connections* of 2,000, but provides a low response time for transactions and never switches to a thrashing state.

3.4.3 Influence of Experimental Variables

Along with the experimentation, we analyze the health of the OS as well as the network traffic to evaluate whether they influence the testing results (Research Question RQ1). We use the "dstat", "slabtop", and "netstat" tools to provide complete information about CPU, memory (including kernel cache information), network, and disk

consumption. In the DBMS-X tests, we observe a steady resource consumption, especially at Steps 2, 4, and 5 that provide heavier workloads. Therefore, we consider they did not interfere in the test results.

In the PostgreSQL tests, we execute it on top of two different OS (GNU/Linux and FreeBSD), since we do not observe a steady resource consumption. Both OS are known to be robust upon heavy load conditions. Table 3.3 shows that the system becomes CPU-bound only upon peak conditions at Steps 4 and 5, with PostgreSQL rejecting a large number of requests and delaying the response time. Figure 3-4 shows that the CPU consumption increases from Step 1 to 5, 360% on GNU/Linux and 308% on FreeBSD. This growth is not only due to admission control, but also to manage a larger number of accepted transactions. PostgreSQL uses a multiprocess model to handle transaction requests. Naturally, the more requests it receives, the more CPU it consumes.

We investigate whether some other OS resources are contributing to the faulty behavior. In fact, the memory consumption increases at different steps (see Figure 3-4), but the overall consumption is not high compared to DBMS-X. Actually, the CPU was the only resource affected by the incremental workload due to context switches in transaction management, while the consumption of the other resources remained constant during the tests. In addition, networking results present a small deviation with only a small impact of received and sent packages. For instance, the amount of sent packages from Steps 3 to 5 increases from 250K to 320K. The I/O throughput does not increase too much as well, with writing operations ranging from 14M at Step 3 to 18M at Step 5. With memory essentially free, low I/O throughput and networking, we observe the testing results are not influenced when STEM is incremented.

3.4.4 Discussion

Our experiments validated Research Question RQ0, which states that we can analyze the test results through log files (DBMS-X) or code coverage (PostgreSQL).

Through log files, we treat the DBMS as a black-box by analyzing its output traces

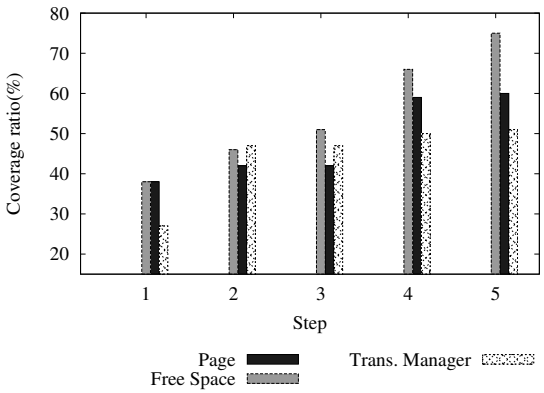


Figure 3-5: PostgreSQL's Code Coverage.

with respect to the documentation. In addition, we analyze the expected performance metrics compared to the observed one at different steps. In this way, we identified the DBMS-X's defect on the network backend process.

The code-coverage approach provides a fine grained manner to reach and understand the source of the defect. We also use it for validating that STEM exercises parts of the system that would never be covered otherwise. As illustrated by Figure 3-5, code-coverage increases mainly at Steps 4, and 5 where a large number of transactions were submitted. In particular, STEM allows to force many robustness mechanisms to be exercised and tested. These limit cases are precisely those that may occur under real large-scale conditions.

STEM presents three advantages compared with the current testing approaches that are similar to Steps 1 to 3 executed directly and without an incremental approach. First, the methodology for implementing and executing the tests is simple. Second, we can diagnose the root of the defects, and under which precise execution conditions the defect can be reproduced. The isolation of the execution conditions that may provoke a defect is especially crucial for diagnosing a problem and fixing it. Third, the incremental approach was able to expose different defects at each step.

Research Question RQ1 considers that different workloads exercise different parts of the code. This is verified for PostgreSQL along with all test steps in two ways, one with the original source-code and another with an instrumented source-code, which generated code coverage information. After analysis, this information revealed that

the more the workload increases, the more code is exercised. In DBMS-X, we analyzed whether different log file entries appear. Our results show that the network backend process only issues error entries when treating large workloads.

Research Question RQ2 considers that, if the DBMS manifests an unexpected behavior on different OS, or the OS resource consumption remains steady, then the OS has no effect on the test results. This is verified for PostgreSQL, which had similar behavior on Linux and BSD. Moreover, system monitoring tools revealed a steady resource consumption for network, disk I/O, and memory. One could argue that Linux and BSD share part of their code, specially network, and a same defect could be present on both OS. Whether this could be possible, it is rather unlikely, since the PostgreSQL unexpected behavior revealed in Step 4 is not reproduced by DBMS-X.

Research Question RQ3 considers that the DBMS under test should not use all allocated resources. The rationale behind this question is that we should not stress the OS, otherwise identifying the source of an abnormal behavior would be complex. This is verified at the stress test (Step 5), where 200,000 transactions are sent. Monitoring tools reveal that both DBMS never reach their configuration limits, whereas the OS performance presents low impact on the overall results.

3.5 Conclusion

In this chapter, we presented STEM, a stress testing methodology for traditional DBMS. We can draw many interesting findings from the experimental results after applying STEM. First, stress testing requires a distributed testing approach, where multiple test drivers are deployed across a cluster machine to submit a large number of transactions. The single driver approach, used in the related work, bounds the size of the load, thus leading to a classical functional test. The execution of STEM is straightforward, but in contrast to related work, requires many machines to execute. Actually, a classical functional test would only discover the PostgreSQL's defect found at an early step of STEM without any trace of a lower/upper bound number of

accepted or rejected transactions. The defect happens due to a size limitation of an internal array that is responsible for managing backend processes. Upon concurrent requests, the array fills out quickly, thus preventing the treatment of new processes (including new transaction requests). A better approach would bound the size of such array by a related DBMS configuration parameter as expected from the DBMS documentation.

Second, traditional DBMS are tightly coupled with performance tuning in order to boost performance. STEM looks after this matter by its incremental testing approach. That is, STEM combines incremental tuning to address burgeoning transaction workloads.

Finally, the incremental approach was able to expose different defects in PostgreSQL at each step and a defect within the network backend process of a leading commercial DBMS. When DBMS-X gets closer to the expected maximum number of concurrent connections, it starts issuing an unexpected network error message. From its documentation, such message is due to network misconfiguration (e.g., wrong SQL URL, port, or IP address) or a network backend process issue (e.g., offline process, network port is closed, network or hardware failure).

Chapter 4

Model-based approach for Database Stress Testing

4.1 Introduction

In this chapter we present the **Mo**del-based approach for **Da**tabase **S**tress **T**esting (MoDaST), a novel model-based approach to reveal potential non-functional defects in DBMS, specially NewSQL. MoDaST focuses on testing performance of DBMS with dynamically changing workload levels. MoDaST leverages a state machine model in order to mimic different workload levels. This model drives a database system across five observable states: Warm-up, Steady, Under-Pressure, Stress, and Thrashing. The advantage compared to other testing tools is that the state machine model allows users to infer and explore internal states of a DUT even if only black-box testing is available. In addition, users may change the workload levels in order to challenge the DUT for different test objectives. For instance, the testing objective can be detect performance loss conditions or thrashing states.

The main difference from STEM to MoDaST is that the earlier focuses on the relationship between workloads and system tuning, while the latter focuses on driving the system to particular performance states. NewSQL is mostly based on the "no knobs" approach, which means less human interference in system tuning.

The remainder of this Chapter is organized as follows. Section 4.2 describes our

model-based approach for database stress testing. Section 4.6, we report the results of the experiments. Section 4.7 concludes with future directions.

4.2 MoDaST

MoDaST consists of the database state machine (DSM) and a test driver. DSM represents a set of observable states of a DBMS and its transition function (Figure 4-1 shows an overview). The test driver defines workload models of each state and commences performance testing by giving a specific amount of workload to a DUT.

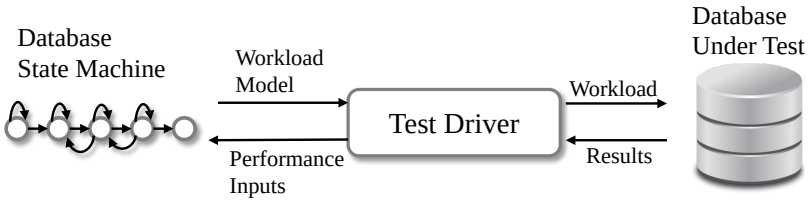


Figure 4-1: Overview of MoDaST.

Then, the driver observes the current performance data of the DUT and figures out state transitions by giving the data to DSM. The remainder of this section describes the details of DSM, test driver, and performance degradation prediction.

4.2.1 The Database State Machine (DSM)

The Database State Machine (DSM) models how a DUT behaves at given workload levels. In particular, DSM focuses on representing observable states of a DUT with respect to performance (i.e., performance behaviors). The behaviors of a DUT can be represented by the following five states: Warm-up (s_1), Steady (s_2), Under Pressure (s_3), Stress (s_4), Thrashing (s_5). Figure 4-2 shows DSM and Definition 3 formally defines of DSM and its corresponding states.

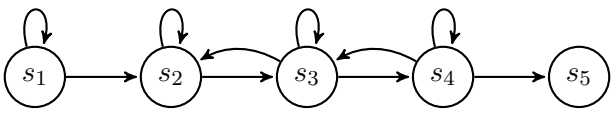


Figure 4-2: The Database State Machine (DSM)

Definition 3. *The Database State Machine (DSM) denoted as T, is a 5-tuple* $(\mathcal{S}, s_1, \mathcal{F}, \beta, \tau)$ *where:*

- $\mathcal{S} = \{s_1, s_2, s_3, s_4, s_5\}$ *is a set of states,*
- $s_1 \in \mathcal{S}$ *is the initial state,*
- $\mathcal{F} \subset \mathcal{S}$ *is the set of final states, where* $\mathcal{F} = \{s_5\}$ *in DSM,*
- β *is the set of performance inputs defined in Definition 4,*
- τ *a state transition function defined by Definition 8.*

The detailed information about every state is available in Section 4.2.1.2. To describe each state in detail, it is necessary to define the performance inputs, β, of DSM. Based on the performance inputs, DSM determines state transitions, τ, in a DUT. β (performance inputs) and τ (transition function) are defined in Definitions 4 and 8, respectively.

4.2.1.1 Performance Inputs

DSM takes three different performance inputs from a DUT to infer its current internal state. The inputs, β, is the set of 1) performance variation, 2) transaction throughput, and 3) performance trend as described in Definition 4.

Definition 4. *The* **Performance Inputs**, *denoted by* β, *is a tuple of three performance variables:* $\beta =< \Delta, \delta, \varphi >$, *where* Δ *is the performance variation (Definition 5),* δ *is the transaction throughput (Definition 6), and* φ *is the performance trend (Definition 7), respectively.*

Performance variation, denoted by Δ, represents the stability of the SUT. MoDaST makes n observations and computes the dispersion of the number of transactions treated per second for each observation (see y-axis in Figure 4-3). For example, if $\Delta \to 0$ after n observations, the DUT is processing a steady number of incoming transactions.

Definition 5. *Performance variation, Δ, is the dispersion of the number of treated transactions per second and formally defined as:*

$$\Delta = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(y_i - \mu)^2}, \tag{4.1}$$

where $\mu = \frac{1}{n}\sum_{i=1}^{n}(y_i)$

Transaction throughput, denoted by δ, is the proportion between the number of transactions treated and requested per second. This metric defines the upper bound number of transactions in concurrent execution with steady behavior. For example, if $\delta \to 1$ across a number of performance observations, the DUT is successfully treating most of the requested transactions.

Definition 6. *Transaction throughput, denoted by δ, is the proportion of the transactions treated per second (y) by the number of transactions requested (z):*

$$\delta = \frac{y}{z} \tag{4.2}$$

Performance trend, denoted by φ, is a metric explaining the expected performance slope of the DUT within a certain timeframe (i.e., implemented by a sliding window). To compute the performance slope, we use the least square method [15] to approximate the running time (x-axis) and the throughput (y-axis). As shown in Figure 4-3, φ is the distance between the current observation timestamp and the expected timestamp when the DUT's throughput converges to 0 (i.e., $\delta = y/z = 0$, where $z \neq 0$). Section 4.2.2 describes how to compute the distance in detail.

Definition 7. *Performance trend is a function defined as*

$$\varphi = x' - x \tag{4.3}$$

where x is the current time and x' represents the point that the tangent line crosses the time axis.

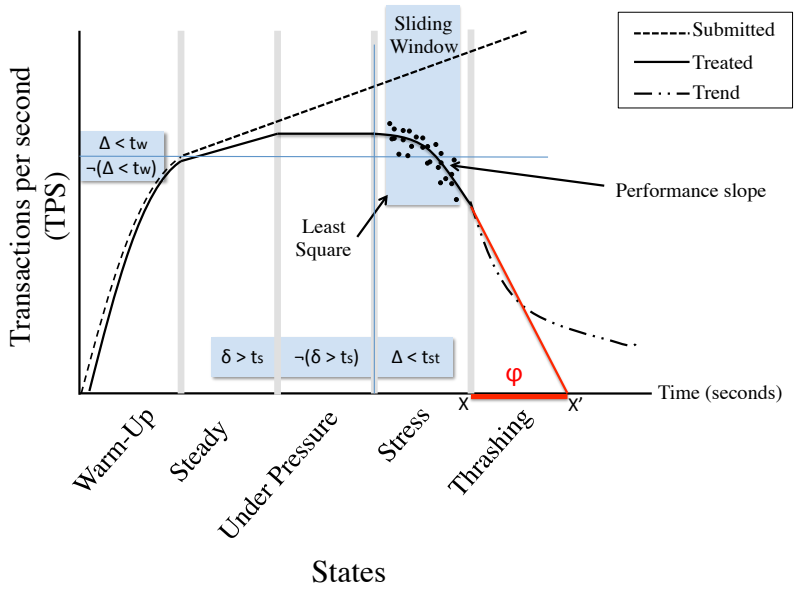


Figure 4-3: The relationships between performance inputs and states of the DSM.

4.2.1.2 States

DSM has five states as shown in Definition 3. These states represent internal performance behaviors of a DUT. The following paragraphs describe each state in detail. Different DBMS may assume different values for each threshold.

State 1 — Warm-up: This state is the startup process[1] of the DUT. In this state, the DUT initializes internal services such as transaction management service. Although some transactions can be treated during the state, its performance is not stable since the DUT focuses on initialization and filling memory caches. DSM defines the Warm-up state by using performance variations (Δ in Definition 5) since it reflects the internal memory status of a DUT.

DSM assumes that a DUT is in the *Warm-up* state if Δ is not converging to 0 after the startup of the DUT. In other words, $\neg(\Delta < t_w)$, where t_w is the warm-up threshold value. Otherwise (i.e., $\Delta < t_w$ holds), the transition to the next state (i.e., **Steady**) is triggered. Section 4.5 explains how to determine the value.

State 2 — Steady: a DUT goes to this state if its performance variation, Δ, is converging to 0. Once the DUT is in this state, it never comes back to the *Warm-up*

[1]Microsoft SQL Server - Buffer Management, http://technet.microsoft.com/

state again since all the internal services are already initialized and running. In addition, the memory cache of the DUT is filled to provide an expected performance, which indicates that the DUT can correctly treat most of incoming transaction requested by clients in time. Specifically, this can be represented as $\delta > t_s$, where t_s is the steady threshold value.

State 3 — Under Pressure: This state implies that a DUT is on the limit of performance. The DUT goes to the state if δ approaches to zero, which means that a set of unexpected workload is coming to the DUT. The unexpected workload includes workload shifts and sudden spikes (e.g., Black Friday or Christmas) that affect performance [82, 86, 36]. In this state, the DUT can still deal with the similar amount of transactions processed in the previous state (*Steady*). However, it cannot properly treat a certain amount of transactions in time since the total amount requested by clients is beyond the limit of the DUT. Although this situation can be transient, it might need an external help from the DB administrator (DBA) to go back to *Steady*. For example, DBA can scale up the DUT's capability or reject a certain amount of the incoming transactions until the workload decreases to an acceptable amount (i.e., $z \rightarrow y$ and $\delta > t_s$).

State 4 — Stress: a DUT goes into this state when the number of transactions requested by clients is beyond the performance limit. This state is different from the *Under Pressure* state since the performance variation (i.e., Δ) increases. The DUT in this state is highly vulnerable to crash if no external help is available. For example, DBA should consider additional solutions such as adopting database replica, adding more cluster machines, or killing long running transactions (normally triggered by bulk loads). If an appropriate solution is performed, the DUT can go back to the *Under Pressure* state and $\Delta < t_{st}$, where t_{st} is the stress threshold value.

State 5 — Thrashing: This state represents that a DUT uses a large amount of computing resources for a minimum number of transactions. The DUT experiences resource contention and cannot deal with any new transaction in this state. Once a DUT goes to this state, it is not possible to come back to the previous state since any external intervention is useless. DSM detects the transition to the *Thrashing* state if

Table 4.1: Threshold values for state transitions.

	Target State				
States	s_1	s_2	s_3	s_4	s_5
s_1	$\neg(\Delta < t_w)$	$\Delta < t_w$	-	-	-
s_2	-	$\delta > t_s$	$\neg(\delta > t_s)$	-	-
s_3	-	$\delta > t_s$	$\neg(\delta > t_s)$	$\Delta > t_{st}$	-
s_4	-	-	$\neg(\Delta > t_{st})$	$\Delta > t_{st}$	$\varphi < t_{th}$
s_5	-	-	-	-	$\varphi = 0$

$\varphi < t_{th}$, where t_{th} is the thrashing threshold value. Predicting the thrashing state is explained in Section 4.2.2.

4.2.1.3 State Transitions

The state transition function, τ, determines whether a DUT changes its internal state based on observed performance data. This function takes *performance input* $\beta = (< \Delta, \delta, \varphi >)$ from the test driver and returns the next state $s \in S$ as described in Definition 8.

Definition 8. *The state transition function, τ, is defined as:*

$$\tau : \mathcal{S} \times \beta \rightarrow \mathcal{S} \tag{4.4}$$

where $\forall s \in \mathcal{S}, \exists p \in \beta$ and $\exists s' \in \mathcal{S} | (s, p) \rightarrow (s')$.

In each state, the DSM examines the current values of performance inputs and compares the values with the threshold values[2] (i.e., t_w, t_s and t_{st}). Table 4.1 summarizes the performance constraints for state transitions.

4.2.2 Predicting the thrashing state

In addition to performance testing, MoDaST allows predicting thrashing states, before the DUT crash. This indicates the time remaining until the DBMS *out-of-service* while allows DBA taking precautions to avoid the service crash. Another advantage

[2]The values used in the experiments are specified in the GUI since it is variable depending on the DUT.

is to forecast resource contention or over-consumption, since DBMS may waste computing resources in the thrashing state.

The first step to predict the *Thrashing* state is computing the performance slope. MoDaST uses the Least Squares method [15] that approximates the relationship between independent (x) and dependent variables (y) in the form of $y = f(x)$. x is the time each second and y denotes the corresponding throughput at time x. It allows the computation of the three required coefficients (i.e., a_0, a_1 and a_2) for the quadratic function (see Equation 4.5). Our approach computes the coefficients by using recent p observations[3] of (x, y) (i.e., sliding window). Once we have the quadratic function, it is possible to estimate the performance slope as shown in Figure 4-3.

$$f(x) = a_0x^2 + a_1x + a_2 \tag{4.5}$$

Once the performance slope is identified, it is possible to determine the tangent line. The tangent line can be computed by applying the derivative $f'(x)$ considering the current observation x_i. Using the tangent projection in the axis x, MoDaST can estimate the performance trend, φ, according to Definition 7. If the value is converging to the thrashing threshold (t_{th}), we can assume that DUT may crash at any moment (i.e., transition from the stress to thrashing state).

4.2.3 Test Driver

The Test Driver (TD) is built on top of the PeerUnit distributed testing framework [25]. PeerUnit allows building and executing distributed test cases, which are key features for stress testing. Figure 4-4 presents the internals of PeerUnit and the stub for implementing the TD test case. Basically, a test case is a Java `TestingClass` class that is executed by distributed nodes (called *Testers*) and coordinated by distributed test harness algorithms [26, 25]. Locally at each "Tester", the `TestRunner` class is the execution engine for reading and invoking code statements from the test cases. A *Tester* can be accessed by the `Tester` interface when a test case extends

[3] p is defined by the least squares correlation coefficient [15]

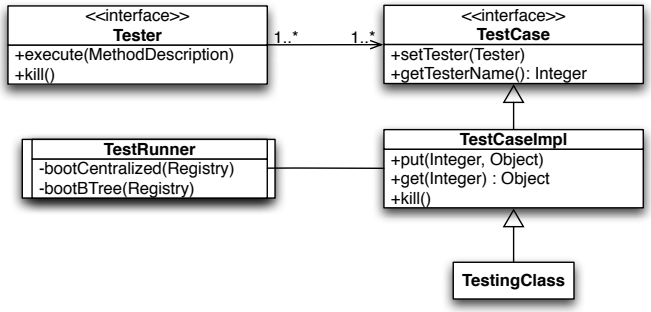


Figure 4-4: H-Tester Test Driver based on the PeerUnit testing framework [25].

the `TestCaseImpl` class. The `TestCaseImpl` class provides a set of generic operations to take advantage of the features provided by PeerUnit. For instance, it provides an interface for exchanging messages among *Testers* (i.e., `put(k,v)` and `get(k)`) or dropping DUT nodes (i.e., kill()). This interface is particularly convenient for gathering performance metrics from both the DUT nodes and the running *Testers*. TD also offers log traces for "post-mortem" analysis, such as: exceptions and memory leak traces.

Listing 4.1: `TestingClass` code snippet of MoDaST

```
1  public class DSM extends TestCaseImpl{
2    /* Controls the workload with the workload distribution
3    and provides access to the DUT*/
4    Distribution dist = new DistributionFactory('zipf');
5    Statistics stats;
6    Stack<Threshold> succ, pred;
7    /*Set a timeout for the test*/
8    Integer testTerm = gui.getTestTerm();
9
10   @BeforeClass(...)
11   public void startUp() {
12     /*Start gathering statistics*/
13     stats.start();
14     ...
15     /*Create the states stacks and inform the thresholds given by the GUI*/
16     succ = new Stack<Threshold>();
17     succ.addAll(gui.getThresholds());
18     pred = new Stack<Threshold>();
19     /*In this call we start the DUT and the workload submission*/
20     dist.run();
```

```
21    }
22
23    @TestStep(timeout = testTerm, ...)
24    public void transitionMonitor() {
25      Threshold actual=succ.pop();
26      while(!succ.empty())
27        /*Check the statistics if the actual state overtakes any threshold */
28        if(isStateTransition(actual, stats)){
29          /* If moving to the next state, store the actual state
30          otherwise retrieve former states*/
31          if (moveForward()){
32            pred.push(actual);
33          }else{
34            succ.push(actual);
35            succ.push(pred.pop());
36          }
37          actual=succ.pop();
38        }
39        ... // sleep a while
40      }
41      assert.Fail();
42    ...
```

Listing 4.1 presents the code snippet for the TD `TestingClass`. The method `startUp()` is meant to initiate the internal services, the DUT and the workload submission. At each iteration, a given workload is submitted to the DUT (**Line 20**) and bounded by the performance constraints.

The method `transitionMonitor()` reads monitoring information from the DUT and the underlying OS to check the performance constraints. Once the DUT cannot hold one of the constraints any longer (**Line 28**), TD considers that a state transition has happened. TD controls the state transitions by pushing and popping state labels into two stacks `Stack<Threshold>`: one for the transited states and another for the upcoming states (**Lines 31 to 37**). For instance, coming back from *Under pressure* to *Steady* state requires popping the state label from the `prev` stack and pushing to the `succ` stack.

The TD generates two different types of workload as shown in Table 4.2. Since the performance can be affected by both the number of connections and transactions, it is necessary to test both.

Table 4.2: Workload cases for the test driver. Workload case #1 creates a high workload in terms of connections; Workload case #2 focuses on transaction flooding.

	Workload Case #1			Workload Case #2		
step	conn	trans	tps	conn	trans	tps
1	10	1	10	10	10	100
2	20	1	20	10	20	200
⋮	⋮	⋮	⋮	⋮	⋮	⋮
n	c	1	$c * 1$	10	x	$10 * x$

- **Workload case #1**: The goal of this case is to impose a high workload to the **connection** module of the DUT. The number of connections is gradually increased for each step. In this case, the driver submits only one transaction per connection.

- **Workload case #2**: The goal of this case is to submit a high workload to the **transaction** module of the DUT instead of the connection module. The number of transactions is gradually increased for each step. In this workload case, the driver submits an increasing number of transactions per connection, where the number of connections is fixed.

4.3 Research Questions

We established the following research questions to systematically evaluate the effectiveness of our approach.

- **RQ0**: Does DSM properly reflect performance behaviors of a DUT?

- **RQ1**: How much does our approach cover the source code of a DUT (i.e., code coverage)?

- **RQ2**: Does our approach find defects?

- **RQ3**: Can our approach predict performance degradation (e.g., the thrashing state)?

RQ0 is a baseline question since our approach assumes that a DUT follows DSM described in Section 4.2.1. If the DUT does not follow DSM, the subsequent research questions may not remain effective. In particular, DSM affects both 1) exploring the code coverage of each state and 2) predicting the thrashing state up front.

RQ1 is selected since our motivation described in Section 1.1 assumes that some execution paths in DBMS source code can be explored only when a certain amount of workload is requested. In addition, it is necessary to examine the code coverage of each module since it is important to figure out which module has functionality for stress conditions.

RQ2 is correlated to **RQ1**; if our approach can explore more lines of a DUT's source code by submitting different levels of workload, we may find new defects located in functions dealing with stress conditions. The defects may include both non-functional and functional.

RQ3 examines whether our approach can predict the thrashing state before a DUT crashes. This RQ is necessary because performance prediction is one of the advantages when using MoDaST. If the prediction is available, DBA can apply several solutions for preventing DBMS crashes.

4.4 Subjects

Table 4.3 lists the subjects used in our experiments presented in Section 4.5. We applied our approach to two DBMS; VoltDB and PostgreSQL. MoDaST is specially designed to assess non-functional requirements of NewSQL, however it can also be applied in traditional DBMS without taking into consideration the tuning knobs. Section 4.5 shows that is possible to infers the DBMS states in both cases, with interesting findings related to the fact that traditional DBMS is easily pushed to the performance limits without any tuning process.

Table 4.3: DBMS used in our experiments. "Size" represents the lines of code in thousands (KLOC). "Versions" is the version of DBMS selected in the experiments. "Feature" represents the storage strategy of each DBMS.

Subject	Size (KLOC)	Versions	Features
VoltDB	403	4.5	In-Memory
PostgreSQL	757	8.3	Disk oriented

4.5 Experimental evaluation

We conducted several experiments to answer the research questions explained in Section 4.3 based on the following procedure. The experiment procedure has four steps: 1) Submit the workload case described in Section 4.2.3, 2) Analyze the execution results based on the RQs, 3) Collect the code coverage data, and 4) Proceed to the comparative study.

The experiments are executed on a HPC platform [94]. We used two different configurations: 1) 11 machines for PostgreSQL (one DBMS server and ten testers) and 2) 13 machines for VoltDB (three DBMS server and ten testers). Each machine has dual Xeon X5675@3.07GHz with 48GB of RAM, which runs Debian GNU/Linux and connected by a Infiniband QDR (40Gb/s) network. Our approach is implemented in Java.

We used the database schema and the transaction specification defined by TPC-B benchmark[4]. TPC-B provides four tables that implement a hypothetical bank. The bank has one or more branches and each branch has multiple tellers. The bank has many customers, each with an account. The transaction represents a customer's deposit or withdrawal.

To collect the code coverage information of PostgreSQL, we used the GNU/Gcov tool[5], which is supported by default by the DBMS. For VoltDB, the code coverage is measured by JaCoCo[6], since the DBMS is implemented in Java.

The threshold values are specified in Table 4.4. Since VoltDB is an in-memory database, the warm-up process is basically instant and not substantial. The thrashing

[4]http://www.tpc.org/tpcb
[5]https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[6]http://www.eclemma.org/jacoco/

Table 4.4: Threshold values for the state transitions. VoltDB does not need values for the warm-up and thrashing states since this DBMS does not experience these states.

	PostgreSQL	VoltDB
t_w	0.1	–
t_s	0.9	0.9
t_{st}	0.1 * tps_{up}	0.1 * tps_{up}
t_{th}	1	–

state is not observed on the VoltDB. The t_s threshold is limited by 90% of the transaction rate acceptance and the t_{st} is limited by 10% of the transaction acceptance rate compared to the previous state (i.e., *Under Pressure*). For the t_{th}, we used one second. The slide window is set to 60 observations (i.e., $p = 60$).

4.5.1 Comparative Study

Several techniques aim to analyze the performance behaviors of database systems. These techniques are known as benchmarks, as so, they focus on performance metrics in order to compare different database systems 2.4.2. Thus, we conducted a comparative study between MoDaST and a baseline technique to attest the effectiveness of our approach. For that, we selected the TPC-B benchmark[4] as the baseline technique for two reasons: 1) it is considered a standard stress benchmark and 2) its "Residence time" constraint (i.e., "90% of the transactions with a response time less than or equal to two seconds") is similar to the steady state in DSM. TPC-B imposes performance constraints to ensure a stable condition to the DUT.

4.6 Results

In this Chapter, we present the results of experiments described in Chapter 4. This section explains the four results: A) performance results, B) code coverage, C) defects, and D) thrashing prediction.

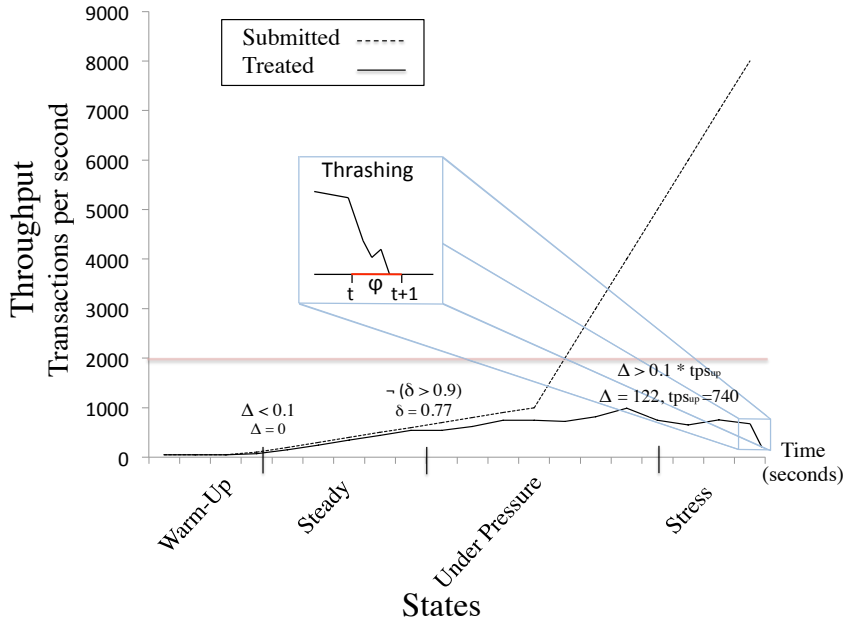


Figure 4-5: Performance results of PostgreSQL.

4.6.1 Performance Results

PostgreSQL experienced all the states of DSM as shown in Figure 4-5. It presented an unexpected behavior concerning the ability to maintain a stable performance. During the execution of the workload case #1, the test driver increased the number of connections sequentially as described in Section 4.2.3. According to the specification of PostgreSQL, it can process 2,000 concurrent connections (i.e., defined by the MAX_CONNECTION configuration). However, the DUT could not deal with any workload greater than 1,000 concurrent connections as shown in Figure 4-5[7]. For the workload case #2, the test driver increased the number of transactions with a fixed number of connections. PostgreSQL's behavior was more stable in this case and did not reach the thrashing state. However, it stayed either in of the under pressure or stress states.

VoltDB presented consistent results in terms of throughput stability. Thus, the DUT was mostly either in the steady or under pressure states for both workload cases (see Figure 4-6). However, the connection module was forced into stress state trigger-

[7]The thrashing state is only observable in the workload case #1.

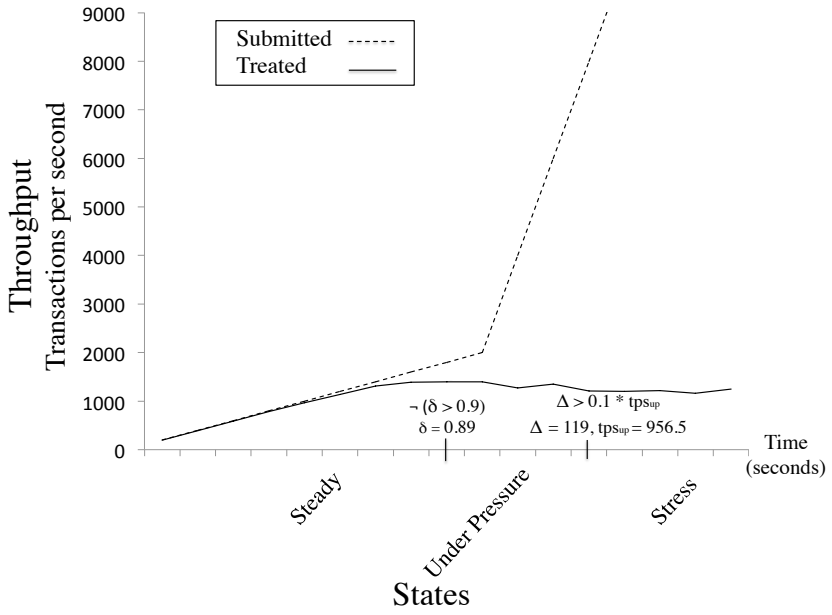


Figure 4-6: Performance results of VoltDB.

ing a backpressure condition[8] when applying the workload case #1. This condition occurs when a burst of incoming transactions was sustained for a certain period of time. This period must be sufficient to make the planner queue full. More information about this condition will be described in Section 4.6.3.

The baseline approach considers a performance constraint to ensure the DBMS on the steady state during measurement time. This constraint is defined as the "Residence Time". It limits the test with the condition: "90% of the transaction with a response time less or equal to 2 seconds". Thus, it is not possible to explore any stress condition of the DUT, since it never reached the under pressure nor the stress states. In both workload cases, the baseline approach only contemplates the steady state, by definition.

Answer to RQ0: MoDaST drove a DUT into each state of the DSM while the baseline technique explored only two initial states.

[8]http://voltdb.com/download/documentation/

4.6.2 Code Coverage

Figure 4-7 shows the code coverage results of PostgreSQL. Three packages presented a significant impact on the following modules: (i) *Freespace* that implements the seek for free space in disk pages; (ii) *Page* that initializes pages in the buffer pool; and (iii) *Manager* that implements the shared-row-lock. The coverage increased mainly during the two last states: *stress* and *thrashing*. It occurs because those packages are responsible for: 1) managing disk page allocation; 2) the transaction lock mechanism to ensure ACID properties. This implies that the DUT executed a set of functions dedicated to stress conditions since the DUT needs to allocate extra resources and to deal with the high concurrency when the number of transactions increases.

VoltDB showed a notable improvement of code coverage results as shown in Figure 4-7, even though it was not significant compared to that of PostgreSQL. We observed the improvement in two packages: *org.voltdb* and *org.voltdb.sysproc*. These packages contain classes that manage the performance: the maximum number of connections[9] and transactions[10]. The package "org.voltdb.sysproc" is related to the basic management information about the cluster.

The above-mentioned VoltDB classes were not covered when applying the TPC-B since it was not able to generate race conditions. Basically, the warm-up and steady states did not generate any concurrent condition. Similarly, the TPC-B could not discover more execution paths of PostgreSQL compared to our approach.

Answer to RQ1: MoDaST allows to explore a larger part of source code of DUTs than the baseline technique since a certain part of source code can be executed only when pushing the DUT to a high workload.

4.6.3 Defects

During the experiments, we found two potential defects (one from PostgreSQL and one from VoltDB) and one new unreported major defect (VoltDB)[11].

[9] http://zip.net/bwpsFS
[10] http://zip.net/byptyG
[11] http://zip.net/bvptxQ

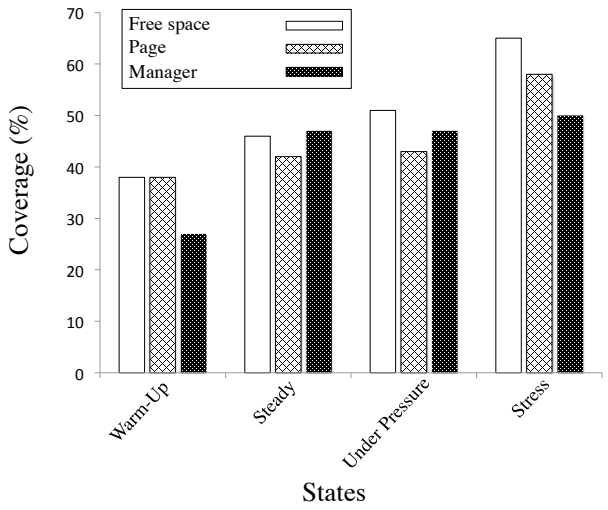


Figure 4-7: Code coverage results of PostgreSQL. This focuses on three major modules: Free Space, Page, and Manager.

We identified a performance defect of PostgreSQL, which is related to the inability to deal with the incoming connections, mainly in the workload case #1. Actually, the defect can be triggered either by transaction or connection flooding. PostgreSQL implements a backend process to deal with the incoming clients. Each client keeps one connection with the database. Thus, for each incoming connection, the DUT starts a new backend process.

Moreover, each connection holds one or more transactions, which proceed to modifications in the database. The modifications are made by *insert* and *update* operations that compose each transaction. To ensure the consistency of the database, the DUT must apply table locks. The DBMS configuration allows to set the number of concurrent connections (i.e., MAX_CONNECTIONS) up to the resources limitations. In our experiments, the maximum value set for MAX_CONNECTIONS was 2,000. Despite of this limit, the DUT was not able to reach the number of 2,000 open connections at any time. As the number of connections/transactions increases, the DUT spends most of the computational power dealing with the table locks instead of creating new backend processes. From the testing point of view, we consider a potential *defect*.

The VoltDB experienced a backpressure condition by applying the workload case

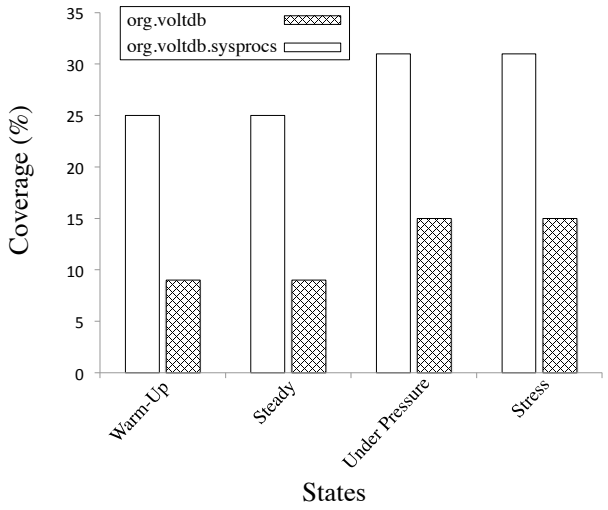


Figure 4-8: Code coverage results of VoltDB. This focuses on `org.voltdb` and `org.voltdb.sysprocs` packages. These packages are related to the concurrency control and server management.

#2. The increasing number of submitted transactions, via JDBC interface[12], fulfills the planner queue limit (i.e., 250) and raised up the message below:

- (GRACEFUL_FAILURE): 'Ad Hoc Planner task queue is full. Try again.'

This can be considered a potential defect[13], once the planner queue is waiting for VoltDB planner. The planner became full and started to reject incoming operations. This is essentially a different kind of backpressure conditions. It could be solved by setting the planner queue limit according to the server available resource.

The code coverage also enabled to reveal a functional defect. Figure 1-2 shows the code fragment where the defect was identified (Line 426). We reported this defect[14] to the developer community of VoltDB. This branch of code is responsible for ensuring that the DUT does not accept more concurrent connections than the maximum constraint allowed by the server resources. The defect rose up when our approach led the DUT to the stress state, which exposed it to a race condition in the connection module. The solution for this defect is to ensure that, even during race conditions, the number of concurrent connections never goes beyond the limit.

[12]http://docs.oracle.com/javase/7/docs/technotes/guides/jdbc/
[13]http://zip.net/bmps8J
[14]http://zip.net/byptRy

Basically it should be guaranteed in the condition statement (i.e., IF) by replacing "==" by ">=". VoltDB developers created a bug report[15] as a major defect and already committed a patch[16] to fix it immediately after our reporting.

Answer to RQ2: The MoDaST found and reproduced three potential defects and one of them is confirmed as a major defect by the developers of VoltDB.

4.6.4 Thrashing Prediction

Our approach could predict the thrashing states of PostgreSQL. As shown in Figure 4-5, the DUT went to the thrashing state after being in the stress state. However, due the instability of the DUT, it crashed immediately (<< one second) after detecting $\varphi < t_{th}$. Thus, it is almost impossible to take any action to avoid such a state.

VoltDB never went to the thrashing state under the two workload cases. This implies that $\varphi >> t_{th}$ and the DUT was highly stable. It does not mean that our approach was not effective. Rather, MoDaST correctly performed thrashing prediction for a stable DBMS. Due to the limitation of the experiment resources, we could not significantly scale up the number transactions, but MoDaST may generate a huge amount of workload to make the DUT go to the trashing state. This remains as future work.

Answer to RQ3: The thrashing prediction was not accurate for two reasons: the high instability of the PostgreSQL turned it to the thrashing instantly; the high stability of VoltDB did not bring it to the thrashing.

4.7 Conclusion

In this Chapter, we presented a novel model-based approach for database stress testing, MoDaST. It leverages a state machine to figure out the current internal state of the DUT based on performance data. We evaluated MoDaST on two popular DBMS: PostgreSQL and VoltDB. Although the goal for MoDaST is originally for testing

[15] https://issues.voltdb.com/browse/ENG-6881
[16] http://goo.gl/8FJJy7

NewSQL, we candidly tested PostgreSQL, with its setup tuned directly to its prime, to show that MoDaST can be applied to any DBMS. The experiment results show that MoDaST can successfully infer the current internal state of both DUTs based on the state model. In addition, we found out that submitting a high workload can lead to a higher code coverage. Consequently, we identified new defects in both database systems. In particular, one of the defects is already confirmed and fixed by a VoltDB developer since it is significant. A good deal of benefits can be drawn from MoDaST.

For test engineers, MoDaST offers a convenient driver to assess non-functional requirements, including, performance, stability, or scalability. MoDaST can be plugged to any DUT, including open-source and closed-source systems due to its black-box nature. Test engineers will also find that the MoDaST stress testing model increases code coverage in the task of finding defects. Although a higher code coverage does not necessarily guarantee detection of defects, engineers can benefit from higher code coverage to set up performance profiling, such as pinpointing performance limitations. For instance, most DBMS have specific code paths for treating stress conditions (e.g., limiting concurrent connections), but those paths cannot be executed unless a certain condition is met. MoDaST may help exercising exactly such condition.

For DBAs, MoDaST could be a powerful tool for "Stress" and "Thrashing" states prediction in a dynamic monitoring. This prediction is particularly useful to pinpoint performance limitations of DUT setups in running environment, such as machine clusters. This is convenient when DBAs want testing new system releases before deploying the database system into production.

Chapter 5

Under Pressure Benchmark

5.1 Introduction

In this Chapter, we present the Under Pressure Benchmark (UPB) to measure performance loss imposed by the availability replication mechanism of NewSQL. UPB measures the performance loss with different degrees of replication and failures. The UPB methodology increases the evaluation complexity from a stable system scenario up to a faulty system scenario upon different load sizes and replicas. The scenarios are designed to evaluate the following different settings: (1) no fault tolerance, no failures; (2) fault tolerance, no failures; (3) fault tolerance with failures. For each scenario, the benchmark generates a set of partial metrics measuring the NewSQL performance. These partial metrics are summarized to provide a unique availability index. We conducted experiments applying our benchmark on two different high-throughput in-memory DBMS: VoltDB and NuoDB.

UPB differs from STEM and MoDaST, mainly on its focus. UPB is designed to be a benchmark. In the benchmarking context, UPB also differs from the related work to our knowledge. While the related work measures performance for finding the best tuning for a given workload, UPB measures performance loss when DBMS activate the availability mechanisms of replication. Therefore, UPB sticks to our goal of assessing non-functional requirements of DBMS. Here, they are performance and availability.

This Chapter is structured as follows. Sections 5.2 and 5.3 present our benchmark and methodology, respectively. Section 5.6 describes our validation through experimentation. Section 5.8 presents the conclusion.

5.2 UPB

The UPB benchmark is composed by two tasks: 1) defining the availability scenarios and 2) executing the scenarios.

5.2.1 Defining the availability scenarios

We define scenarios that represent the set of possible states wrt. availability. The scenarios are chosen based on the combination of values of two variables, as described below:

1. Fault tolerance index (K): quantity of "failed nodes" supported by the cluster without service outage. The possible K values are:
    - $K = 0$ (no fault tolerance): the service stops in presence of any node failure.
    - $K = 1, 2, ..., \frac{N}{2}$: N represents the nodes that compose the DBMS cluster. In this case, the DBMS supports failures in K nodes. The values vary between 1 to $\frac{N}{2}$.
2. Number of failed nodes (F).
    - $F = 0$: cluster without failures.
    - $F = 1, 2, ..., K$: cluster with K "failed nodes". The failed values are between 1 to K.

However, it is not necessary to have one scenario for all combinations of variable values, since some scenarios cannot occur in practice. Table 5.1 shows possible values of the variables, a short explanation and the relevance of each combination.

Since some values depend on the scenario, X represents the values assumed by K ($1 \leq X \leq \frac{N}{2}$) and Y represents the values assumed by F ($1 \leq X \leq K$) in different scenarios.

Table 5.1: *Scenarios*

Combination	**K**	**F**	**Relevance**	**Comment**
1	0	0	Yes	Basis to scenarios comparison
2	0	Y	Unfeasible	$F > K$
3	X	0	Yes	Performance impact of fault tolerance mechanism
4	X	Y	Yes	Performance impact of failures

Following the combination analysis, there are three interesting scenarios that will be applied by UPB:

- Scenario (1) - combination 1: No fault tolerance ($K = 0$), no failures ($F = 0$);
- Scenario (2) - combination 3: Fault tolerance ($K > 0$), no failures ($F = 0$);
- Scenario (3) - combination 4: Fault tolerance ($K > 0$), failures ($0 < F \leq K$).

Since the values of K and F vary, each scenario has one or several steps to evaluate all the possibilities.

According to [90], the overall performance of the DBMS varies about 2 percent during the steady state. Before reaching such state, it is necessary to pass through a warming period, preventing any unstable behavior.

Similarly to the TPC-C benchmark, the warming period is set up by the responsible for applying the benchmark. The UPB performance measures must be the transactions per second (*tps*) average during a period of 2 minutes in a steady state. This period is defined empirically, with the objective to measure the system behavior and to normalize possible punctual performance instabilities. The performance is measured by monitoring the number of *tps*, collected during the steady state.

To guarantee an accurate measurement, each client has a Q configuration to limit their workload throughput (i.e., transactions per second). It prevents from having a stressed environment on the client side (i.e., client contention), that could generate

an unstable performance. The maximum workload a client is able to submit with no contention on the client side (i.e., the maximum value allowed for Q) is represented by L_c. To define L_c we apply an iterative method:

1. Define the low load (Q) respecting the latency ϕ;

2. Measure the performance (tps):
   $IF\ Q * 0.95 \leq tps\ THEN\ Q = Q * 1.10$, repeat step 2 $ELSE\ L_c = Q * 0.90$;

To exemplify this method, let's assume that an hypothetic client is set to submit initially 10 tps ($Q = 10$). The cluster is evaluated with this load and it processes all the transactions. The load of the client is increased by 10% until the difference between the cluster performance and the Q configuration of the client is less than 5%. Consider that the load is increased until $Q = 174$ tps, but the cluster processes just 150 tps. In this case, the difference between the performance expected and achieved is higher than 5%. The L_c is this limit achieved decreased by 10%. In this illustration, 156 tps.

5.2.2 Executing the scenarios

UPB is executed iteratively, in three steps, one per scenario. Each step may be executed more than one time depending on the max value of the fault tolerance index(K). The partial metrics are denoted by $T_{K,F}$ (F represents the "failed nodes").

Step 1:

In this step the fault tolerance and failures are not taken into account. In real environments this configuration is not recommended, but in our methodology it is important to set a basis for comparison. This configuration is used to define the max performance reached by the cluster, since there is no additional costs related to replication and/or failures (i.e., K=0, F=0). The max performance is represented by $T_{K,F}$ (see Algorithm 1).

Step 2:

The fault tolerance mechanisms have an impact on the system performance even

input : Q, a set of client workloads; L_c, maximum workload per client
output: $T_{K,F}$

foreach $q \in Q$ **do**
  $q = L_c * 0.10$;
  while $(Qtotal + q) * 0.95 \leq tps$ **and** $q \leq L_c$ **do**
    $lastQ = q$;
    $q = q * 1.10$;
  end
  if $(Qtotal + q) * 0.95 \leq tps$ **then**
    $Qtotal = Qtotal + q$;
  else
    $T_{K,F} = Q_{total} + lastQ$;
    return $T_{K,F}$
  end
end

Algorithm 1: Baseline

without node failures. This impact varies depending on the implementation and strategies used by each DBMS. The step 2 aims to verify and to measure these impacts.

The fault tolerance index is configured with different values($K > 0$). The node failure index(F) is not used. The goal is to measure the data replication impact on performance. For each K value, the max performance is calculated, using the Algorithm 1, and represented by:

$$D_{K,0} = (1 - \frac{T_{K,0}}{T_{0,0}}) * 100$$

Step 3:

This step measures the performance during a failure state, with a fault tolerance mechanism activated. The DBMS must be operational during node failures. In this case, the performance may be impacted by read workloads (the replicated data is used to maximize the throughput) and write workloads (the replicated data must be synchronized, implicating in additional costs depending on the K value).

The number of failures vary up to the number of replicated nodes: $1 \leq F \leq K$

For each F value, the performance is calculated, using the Algorithm 1, and related

to $T_{K,0}$ obtained in the step 2. Thus, performance degradation is defined by:

$$D_{K,F} = (1 - (\frac{T_{K,F}}{T_{K,0}})) * 100$$

The final degradation metric(DF_K) is given by summarization of the metric $D_{K,F}$

$$DF_K = \frac{\sum_{F=1}^{K} (\frac{1}{F} * D_{K,F})}{\sum_{F=1}^{K} (\frac{1}{F})}$$

We used a weighted average to calculate the global degradation considering that simultaneous failures are less common than unique failures. Thus, the weight of simultaneous failures on the metric are lower.

5.3 Overall Metrics

We define two overall metrics that summarize the partial metrics calculated. Each metric is the average value obtained for one kind of metric, over the K index, as shown in Table 5.2. We do not use step 1 to calculate an overall metric, since it is used only as basis for comparison.

Table 5.2: Overall Metrics

Metric	Step	Description
$D_T = \frac{\sum_{i=1}^{K} D_{i,0}}{K}$	2	Performance degradation with fault tolerance
$D_F = \frac{\sum_{i=1}^{K} DF_i}{K}$	3	Performance degradation during failures

These two metrics are the final results of the performance analysis of UPB.

5.4 The Environment Outline

The benchmark called Under Pressure Benchmark (UPB) evaluates one important mechanism related to DBMS availability [70]: replication. While data replication is up and running, the impact on performance may vary depending both on the evaluation environment and the proper implementation that differ from DBMS (e.g., asynchronous or synchronous). The UPB seeks to assess the efficiency and performance impact of replication with different configurations and workloads. The workload varies up to extreme situations in terms of faults and bulk load.

A set of partial metrics computes performance along the execution of the UPB, for instance, the number of transactions per second with or without node failures. At the end, a final metric sums up the partial metrics to present the overall performance of the DBMS under pressure. We present the main characteristics of UPB as follows:

- Portability: UPB can be applied to any DBMS with minor adaptations.
- Adaptability: The UPB methodology allows to use different workloads.
- Flexibility: Since UPB provides a set of partial metrics, one may use subsets for specific proposes, such as, analyzing different configurations or applications.
- Extensibility: The methodology supports to be extended to consider different DBMS features, by incorporating new metrics.

5.5 Architecture

The UPB architecture is designed to fit a classical DBMS architecture. It is composed by client machines and a cluster of distributed machine nodes to run the DBMS(see Figure 5-1).

5.5.1 Clients

The connection between the DBMS and the clients depends on the DBMS. The clients connect and submit transactions to any cluster node or all the cluster nodes at the

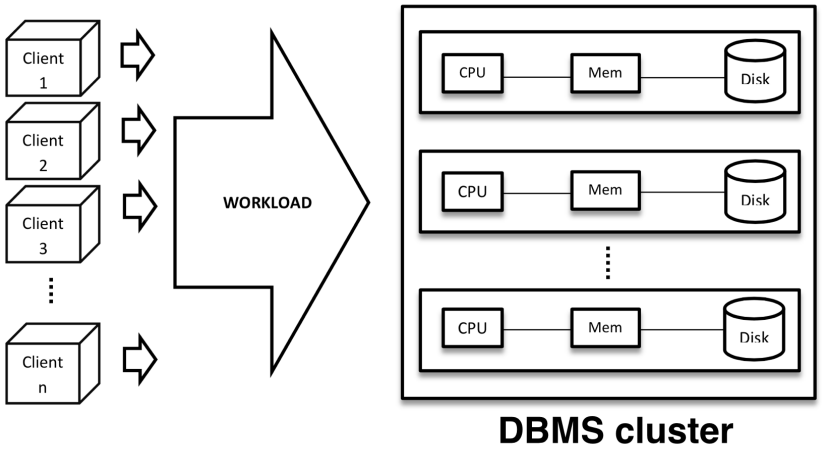


Figure 5-1: UPB architecture

same time with the DBMS connector (i.e. driver) taking care of load balance.

UPB has a stepwise methodology to drive its execution, in which the number of clients submitting transactions grows at each step. Two configuration knobs are required to limit the number of clients and their throughput (in transactions per seconds). The number of clients may vary from DBMS, even if they are running in the same environment, and a tuning procedure must be taken to draw the best fit number and avoid bungles to spoil the final result. The tuning procedure includes leveraging the results of the partial metrics to figure out the proper configuration.

For an Online Transaction Processing (OLTP) application running in a real environment, there is no throughput limitation for clients, since they are distributed all over the Wide Area Network (WAN) and do not run in the same machine. Otherwise, mimicking a large number of clients may saturate the client machine increasing latency of the transaction requests. As consequence, the DBMS may never reach its performance boundary and the final performance result may be spoiled as well. To scale out the benchmark and avoid any client contention, we implemented our benchmark to run across distributed machines [62].

5.5.2 Fault tolerance

Data replication is the main mechanism for fault tolerance in DBMS with two main strategies implemented by the DBMS: asynchronous updating of replicas (i.e., optimistic strategy) and synchronous updating (i.e., pessimistic strategy). For those strategies, a replication factor defines the number of nodes that could be out of operation without data loss or service outage.

The replication factor is configured following the availability requirements given by the application. According to the VoltDB documentation[1], to size the hardware based on data replication, the number of cluster nodes must be a multiple of the number of copies.

The UPB goal is to evaluate the performance impact on the DBMS while a number of its nodes is unavailable. It does not evaluate the impact of different failures. In this context, our understanding follows the definition of *fail-fast* presented in [46] in which any system module should operate perfectly or stop immediately. It means that any failure in a node, whether hardware or the DBMS itself, is enough to immediately disrupt the node.

5.5.3 Workload

In this work we leverage the YCSB [20] workload generator, but the UPB accepts any workload based on the transaction concept. Transactions could be composed by simple database operations (e.g., reads, writes, deletes) or complex business operations, such as, the transactions of the TPC-C and the TPC-E [2]. The only constraint is that at least 90% of the transactions must complete in less than 1 second. This requisite is normally used by well-known benchmarks to cope with real applications [90, 91]. From now, we denote this constraint as ϕ.

[1] http://docs.voltdb.com/PerfGuide/
[2] http://www.tpc.org/

5.6 Experimental evaluation

In this section we present experiments performed by applying the UPB in two different DBMS. Our goal is to validate the proposed benchmark and its methodology as a robust approach to compare availability. The experiments are conducted by following the three steps of UPB.

5.6.1 Experimental Setup

The experimental evaluation was performed in two in-memory DBMS: VoltDB and NuoDB. In order to provide a fair comparison, all the experiments were performed in the same environment, the Grid'5000 testbed [3]. The experimental setup is described bellow:

- Intel Xeon E5440 QC (2.83 GHz / 4 MB), 2 sockets, 4 cores per socket
- Memory 8 GB
- Network Myri-10G (10G-PCIE-8A-C)
- Debian GNU/Linux Lenny x64
- Java^{TM}SE Development Kit 7, Update 17
- YCSB-0.1.4
- DBMS: VoltDB v2.8.4.1 Community Edition, NuoDB Starlings Release 1.1

We used 6 server-class machines running the DBMS and three to run the clients. In order to avoid any interference, the clients and servers were run in separate machines.

To load the database and generate the workload, we used the YCSB framework [20]. The database schema is composed by a single table with 11 columns (see Figure 5-2).

Each column stores 1,000 alphanumeric characters, totalizing 1,100 bytes per record. In our experiments, the database was loaded with 1 million tuples. The

[3] http://www.grid5000.fr

```
CREATE TABLE usertable (

        YCSB_KEY VARCHAR(1000) NOT NULL,
         FIELD1 VARCHAR(1000), FIELD2   VARCHAR(1000),
         FIELD3 VARCHAR(1000), FIELD4   VARCHAR(1000),
         FIELD5 VARCHAR(1000), FIELD6   VARCHAR(1000),
         FIELD7 VARCHAR(1000), FIELD8   VARCHAR(1000),
         FIELD9 VARCHAR(1000), FIELD10 VARCHAR(1000),
        PRIMARY KEY (YCSB_KEY)

);
```

Figure 5-2: Database schema

workload was based on read operations that perform a select using the primary key and columns projection (see Figure 5-3). The YCSB workload engine has been configured to use a Zipfian query distribution [47].

```
SELECT  FIELD1, FIELD2, FIELD3, FIELD4, FIELD5,
        FIELD6, FIELD7, FIELD8, FIELD9, FIELD10
FROM    USERTABLE
WHERE   YCSB_KEY = :1
```

Figure 5-3: Read operation

Each client machine runs 32 threads to generate and to submit the workload. For the DBMS cluster, VoltDB cluster has 36 partitions divided into 6 nodes. The NuoDB implementation doesn't support data partitioning. Every NuoDB node can play three different roles: Broker(i.e., manages access and control of transaction engines and storage managers), Transaction Engine (TE)(i.e., provides access to a single database. It handles requests from clients, caches data, and coordinates transactions) and Storage Manager (SM)(i.e., each storage manager is associated with exactly one database). Thus, each layer can scale independently.

The NuoDB architecture supports one or more TEs, on a single node or across the cluster. The NuoDB Documentation[4] recommends to add TEs to improve perfor-

[4]http://doc.nuodb.com/

mance. Thus, during the experiments, we maximize the number of TEs to improve the requests management. For each database, one SM is required (i.e., depending the number of replicas the number of SMs is increased). The node Broker is fixed in one node. Therefore, the NuoDB cluster assumes different configurations based on the fault tolerance index(K), as shown in Table 5.3.

Table 5.3: NuoDB configuration

Configuration	**Node 1**	**Node 2**	**Node 3**	**Node 4**	**Node 5**	**Node 6**
K=1	Broker	TE	TE	TE	TE	TE & SM
K=2	Broker	TE	TE	TE	TE & SM	TE & SM
K=3	Broker	TE	TE	TE & SM	TE & SM	TE & SM
K=4	Broker	TE	TE & SM	TE & SM	TE & SM	TE & SM

5.6.2 Step 1

UPB has two parameters that determine how the experiments should be executed: Warm-up time and maximum workload per client (L_c). These parameters are related to the environment and must be defined in the first run. The way to obtain these parameters is quite similar for any DBMS. For this reason, we only present the results from VoltDB.

After that it is possible to obtain the maximum performance of the cluster without fails ($F = 0$) and configured with $K = 0$.

5.6.2.1 Parameters definition

To get the workload limit per client (L_c), we evaluate the cluster performance by setting a client with different Q values. The Table 5.4 presents the sustained performance of the VoltDB cluster and indicates the variation between the measured performance and Q configuration of the clients (maximum throughput a client could submit).

When Q parameter is less than 90.000 tps, the client is capable to submit the workload determined by Q. In these situations, the difference between Q and the real performance is pretty close (less than 0,04%). However when Q is higher than 90.000,

Table 5.4: *Defining the workload limit per client (L_c)*

Client Configuration (Q)	**VoltDB Performance (tps)**	**Variation**
70.000	69.999	0,0%
80.000	80.005	0,0%
90.000	90.039	0,0%
100.000	94.543	5,5%
110.000	94.266	14,3%

the clients can't submit more than 95.000 tps. In that situation the difference between the performance and Q is higher than the 5% (threshold defined in methodology). Because that, the maximum Q configuration acceptable by a VoltDB client (L_c) is 90.000 tps.

5.6.2.2 Cluster Performance with K=0 and F=0

To get $T_{0,0}$ it is necessary to run the workload several times, increasing the Q until the difference between Q total and the cluster performance is higher than 5%. The results of the VoltDB runs are presented on Table 5.5. The column Q_{total} indicates the sum of Q of all clients on the environment, considering that the Q configuration of each client must be equal or less than L_c parameter determined previously.

Table 5.5: *VoltDB runs to determine $T_{0,0}$ (No fault tolerance ($K = 0$), no failures ($F = 0$))*

Q_{total}	**VoltDB Performance (tps)**	**Variation**
180.000	180.052	0,0%
220.000	218.839	0,5%
240.000	237.564	1,0%
250.000	238.740	4,5%
255.000	237.378	6,9%
260.000	239.018	8,1%

The VoltDB results indicate that the cluster performance increase continuously until 238.740 tps. After that, the difference between Q_{total} and Cluster Performance is higher than 5%. This behaviour of performance stabilization around 238.000 tps is due a backpressure situation[5]. In all of the VoltDB runs the ϕ requirement has been

[5]http://docs.voltdb.com/UsingVoltDB/

met.

The $T_{0,0}$ of the others DBMS is obtained in a similar way than VoltDB. The results are presented on Table 5.6.

Table 5.6: *Parameters defined for each DBMS*

DBMS	$T_{0,0}$
VoltDB	238.740 tps
NuoDB	112.692 tps

5.6.3 Step 2

On Step 2 we evaluated the DBMS with different K configurations. The results of $T_{K,0}$ and the performance degradation of the fault tolerance systems ($D_{K,0}$) are presented on Table 5.7.

Table 5.7: *The performance degradation using fault tolerance mechanism.*

	VoltDB		**NuoDB**	
Environment	$T_{K,0}$	$D_{K,0}$	$T_{K,0}$	$D_{K,0}$
K=0	238.740	-	112.692	-
K=1	156.659	34.38%	103.340	8.30%
K=2	112.388	52.92%	103.669	8.00%
K=3	101.899	57.32%	101.166	10.22%

* The negative value means a performance increasing, no degradation.

The VoltDB results indicate that the backpressure state was achieved in the experiments. This means that the maximum performance has been reached.

The performance degradation on VoltDB happens because when we increase the replication level, we are dividing the available partitions among the duplicate copies. For example, on our environment when $K = 1$ there are 18 partitions to process transactions and the other 18 partitions store copies of the data. In the VoltDB architecture, the performance will be proportionally decreased as replication is increased.

While NuoDB and VoltDB present the same behavior with performance decreasing as long as K increases.

5.6.4 Step 3

The performance of the DBMS evaluated with faults ($F > 0$) and the performance degradation when compared to a non-fault cluster are indicated on Table 5.8.

Table 5.8: *DBMS performance in a faulty environment - The degradation is based on non-fault cluster.*

	VoltDB		**NuoDB**	
Environment	$T_{K,F}$	$D_{K,F}$	$T_{K,F}$	$D_{K,F}$
K=1 and F=1	152.072	2.928%	111.660	−8.05%*
K=2 and F=1	110.491	1.687%	106.244	−2.48%
K=2 and F=2	109.781	2.319%	109.729	−5.84%
K=3 and F=1	92.8194	8.910%	103.469	−2.27%
K=3 and F=2	86.837	14.781%	103.922	−2.72%
K=3 and F=3	76.003	25.413%	104.304	−3.10%

* The negative value means a performance increasing, no degradation.

The results on a faulty environment indicate that VoltDB performance degradation is less than 3% when K is configured as 1 or 2. But when $K = 3$, performance degradation increases considerably, varying between 8.9% and 25.4%.

Due to its particular implementation, NuoDB behaves differently when increasing performance as the number of faults increase. This is due to the peer-to-peer messaging infrastructure used to route tasks to nodes. Therefore, the fewer node replicas are running the fewer tasks and messages are routed. In addition, we observe that NuoDB has the stablest performance degradation upon faults, which may be also inherited from the resilience of the P2P backend.

Table 5.9: *This summarizes the performance degradation results in a faulty environment.*

Metric	**VoltDB**	**NuoDB**
DF_1	2.928	-8.05*
DF_2	1.897	-3.6*
DF_3	13.511	-2.475*

* The negative value means a performance increasing, no degradation.

Table 5.9 summarizes the performance results in a faulty environment. They

corroborate our observations that VoltDB suffers with faults, while NuoDB actually improves performance due to its P2P nature almost reaching the baseline results ($K = 0$, $F = 0$).

5.7 Final Comparison and Discussion

Based on the partial metrics presented above, it is possible to calculate the final metrics for each DBMS and to compare the availability of them based on two different aspects. The final metrics are presented on Table 5.10.

Table 5.10: *Overall metrics - This summarizes the partial metrics. D_T is the average of performance degradation metric (with fault tolerance), over the K index. D_F is the average of performance degradation metric (during failures), over the K index.*

Metric	**VoltDB**	**NuoDB**
D_T	48.21	8.84
D_F	6.11	-4.70*

* The negative value means an increase on performance, not degradation.

VoltDB had the best overall performance throughput in a faultless scenario, the same was not observed while faults are injected (reflected by the D_F metric). In contrast, NuoDB had some performance impact to maintain the replicas, but presented a surprisingly D_F that is a direct result from its P2P core. Therefore, NuoDB presented the best results in environments with faults.

One may argue that one DBMS is better than the other based on the presented results. While this may be true considering the execution scenarios to test replication and resilience, the DBMS makes a different set of tradeoffs to improve availability, which may lead to situations where one of them will be more appropriate than the other. In this context, we claim that the UPB can be an important tool to help choosing the more appropriate DBMS while heating the debate on availability solutions.

5.8 Conclusion

We presented the Under Pressure Benchmark (UPB) for evaluating NewSQL when supporting availability through replication. The UPB methodology increases the evaluation complexity from a stable system scenario up to a faulty system scenario, including (1) no fault tolerance, no failures; (2) fault tolerance, no failures; (3) fault tolerance with failures. To the best of our knowledge, UPB is the first benchmark of its kind.

The UPB provides a focused benchmark to deal with a central issue related to DBMS availability. We believe that the UPB fits the requirements for evaluating NewSQL upon critical situations, such as heavy loads and failures. Moreover, the UPB provides a good basis for database administrators to take decision about replication indexes, based on performance impact.

We validated our benchmark through experimentation and evaluation of two NewSQL: VoltDB and NuoDB. We have verified that data replication has a large impact on performance, as a side-effect of availability. The impact could be considered negative or positive, depending on the DBMS. This is more evident while the DBMS is under high-throughput load.

Chapter 6

Conclusion and Future Work

In this chapter we present the general conclusions of this thesis. First, we recall the main issues on DBMS testing. Second, we summarize the related work. Then, we present the summary of our contributions. Finally, we point the directions for future work.

6.1 Issues on DBMS testing

DBMS testing can be divided into two categories: functional and non-functional. On the one hand, the functional testing is related to the system ability to reproduce a proper output for a given input. On the other hand, the non-functional testing takes into consideration non-functional requirements concerning the system quality, such as performance, robustness, security, scalability.

Over the last years, non-functional testing became critical due to the recent growth of the transaction workload (e.g., Internet, Cloud computing, BigData) that impacts directly DBMS development. While traditional DBMS require systematically embedding new features, in order to fit these requirements, contemporary DBMS present a completely new architecture.

The main challenge on DBMS testing is to establish a proper testing approach to evaluate the high performance DBMS by systematically submitting increasing volumes of transactions recreating the production workload.

6.2 Current DBMS testing approaches

DBMS validation is commonly performed through benchmarks. Different benchmark approaches were proposed along the last decades, but focusing on providing comparison metrics (e.g., response time, throughput, and resource consumption) [53] rather than on finding defects, such as: Debit/Credit [35], AS3AP [93], TPC-like [2], SetQuery [67], the cloud data service benchmark YCSB [19], and a data-analysis benchmark for parallel DBMS [72]. In addition, current benchmarks do not focus on all new features of contemporary DBMS, such as availability (i.e., replication).

Performance testing tools such as Hammerora [1], Oracle Application Testing Suite [6], and AppPerfect [7], provide a test driver for submit operations based on a TPC-like benchmark. Focusing on functional assessments, the Agenda [30] tool provides a testing methodology to validate ACID properties of the DBMS. Therefore, Agenda does not generate test cases to tackle non-functional requirements.

Regarding the model-based testing, several approaches are focused on performance evaluation needs. In fact, in specific scenarios, they are able to examine functional as well as non-functional requirements of the SUT, such as security [71, 32, 75], reliability [40], efficiency [60, 40, 78] and performance [10, 11, 79, 33], but none of them is appropriate for DBMS stress testing.

6.3 Contribution

In this thesis, we presented three contributions to address different needs on DBMS evaluation and validation: 1) Stress testing methodology for traditional DBMS to expose defects related to the combination of a stress workload and mistuning; 2) Model-based approach for Database stress testing specialized in NewSQL, which infers internal states of the database and point defects based on performance observations under different workload levels; 3) A benchmark to assess availability mechanisms in NewSQL database systems.

1. **Stress Testing Methodology (STEM)**: STEM revealed load-related defects from the combination of stressing workloads and mistuning in traditional DBMS. We can draw many interesting findings from the results. First, stress testing requires a distributed testing approach, where multiple test drivers are deployed across a cluster to submit a large number of transactions. The single driver approach, used in the related work, bounds the size of the load, thus leading to a classical functional test. Next, the incremental approach was able to expose different defects in PostgreSQL at each step and a defect within the network backend process of a leading commercial DBMS. When DBMS-X gets closer to the expected maximum number of concurrent connections, it starts issuing an unexpected network error message. This approach turns out to be an important tool to assess performance limitations of the traditional DBMS under stress conditions.

2. **Model-Based approach for Database Stress Testing (MoDaST)**: A novel model-based approach to reveal potential non-functional defects in DBMS, specially NewSQL. MoDaST enabled pinpointing conditions of performance loss, a technique to predict thrashing states that can damage the system execution and reveal defects related to different states of performance. The experiment results have shown that MoDaST can successfully infer the current internal state of the DUT based on the state model, our Database State Machine (DSM). In addition, we found out that as the DBMS has visited the state machine in direction of the thrashing state, the code coverage increased. Consequently, the probability of finding defects increases in the same way. In particular, we found one defect that has been confirmed as a major defect by VoltDB developers and already applied to the last version of VoltDB.

3. **Under Pressure Benchmark (UPB)**: UPB evaluated the performance of NewSQL considering availability through **replication**. The UPB provided a focused benchmark to deal with a central issue related to NewSQL availability. We believe that the UPB fits the requirements for evaluating the DBMS upon

critical conditions, such as under pressure workloads and failures. Moreover, the UPB provides a good basis for database administrators taking decision about replication indexes based on performance impact. We have verified that data replication has a large impact on performance, as a side-effect of availability. The impact could be considered negative or positive, depending on the NewSQL. This is more evident when the DBMS is under pressure conditions.

6.4 Future work

We focus our future work on MoDaST. This approach proved to be the most appropriate to represent DBMS and to reveal related defects. So far, MoDaST might be applied in several ways. For test engineers, it offers a convenient driver to assess non-functional requirements, including performance, stability or scalability. While our experiments only focus on performance, test engineers can easily extend the test case for other assessments.

Our approach can be plugged to any DUT, including open-source and closed-source systems due to their black-box nature. Test engineers will also find that the stress testing model increases code coverage in the task of finding defects. Although a higher code coverage does not necessarily guarantee defects detection, engineers can benefit from higher code coverage to setup performance profiling, such as tuning setup and performance limitations. For instance, most DBMS have specific functions for stress conditions (e.g., for access control), but those functions cannot be executed unless a certain condition is met. Therefore, MoDaST may help detecting such conditions for building up performance profiles.

For DBAs, MoDaST could be a powerful tool for "Stress" and "Thrashing" states prediction in a dynamic monitoring. This prediction is particularly useful for recking performance limitations of DUT setups in running environment, such as machine clusters. This is convenient when DBAs want to test new system releases and to tune setups before deploying the database system into production.

For future work, we plan on applying MoDaST to cloud computer hypervisors in

order to monitor cloud database-as-a-service systems (DBaaS). For that, additional states may be attached, such as a *High Stress* state between the stress and thrashing states, or a *Idle* state in case the system allocated more resources than needed.

Bibliography

[1] Hammerora: The open source load test tool. `http://hammerora.sourceforge.net/`.

[2] Transaction Processing performance council, 2013. `http://www.tpc.org`.

[3] Fredrik Abbors, Tanwir Ahmad, Dragos Truscan, and Ivan Porres. Mbpet: A model-based performance testing tool. In *VALID 2012, The Fourth International Conference on Advances in System Testing and Validation Lifecycle*, pages 1–8, 2012.

[4] Mohammed Abouzour, Ivan T. Bowman, Peter Bumbulis, David DeHaan, Anil K. Goel, Anisoara Nica, G. N. Paulley, and John Smirnios. Database self-management: Taming the monster. *IEEE Data Eng. Bull.*, 34(4):3–11, 2011.

[5] Anneliese A. Andrews, Jeff Offutt, and Roger T. Alexander. Testing web applications by modeling with fsms. *Software and Systems Modeling*, 4:326–345, 2005.

[6] Oracle Application Testing Suite, 2014. `http://www.oracle.com/technetwork/oem/app-test/index.html`.

[7] AppPerfect, 2014. `http://www.appperfect.com/`.

[8] Thomas Arts and Simon J. Thompson. From test cases to fsms: augmented test-driven development and property inference. In Scott Lystig Fritchie and Konstantinos F. Sagonas, editors, *Erlang Workshop*, pages 1–12. ACM, 2010.

[9] Vikas Bajpai and Ravi Prakash Gorthi. On non-functional requirements: A survey. In *Students' Conference on Electrical, Electronics and Computer Science*, pages 1–4, 2012.

[10] Cornel Barna, Marin Litoiu, and Hamoun Ghanbari. Autonomic load-testing framework. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 91–100. ACM, 2011.

[11] Cornel Barna, Marin Litoiu, and Hamoun Ghanbari. Model-based performance testing. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 872–875, 2011.

[12] Phil Bernstein, Michael Brodie, Stefano Ceri, David DeWitt, Mike Franklin, Hector Garcia-Molina, Jim Gray, Jerry Held, Joe Hellerstein, H. V. Jagadish, Michael Lesk, Dave Maier, Jeff Naughton, Hamid Pirahesh, Mike Stonebraker, and Jeff Ullman. The asilomar report on database research. *SIGMOD Rec.*, 27(4):74–80, December 1998.

[13] Robert V. Binder. *Testing Object-oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[14] Claudio Borio, Mathias Drehmann, and Kostas Tsatsaronis. Stress-testing macro stress testing: does it live up to expectations. 2011.

[15] Shalabh C. Radhakrishna Rao, Helge Toutenburg and Christian Heumann. Linear models and generalizations, least squares and alternatives, 3rd edition. *AStA Advances in Statistical Analysis*, 93(1):121–122, 2009.

[16] Ji-Woong Chang, Kyu-Young Whang, Young-Koo Lee, Jae-Heon Yang, and Yong-Chul Oh. A formal approach to lock escalation. *Inf. Syst.*, 30(2):151–166, April 2005.

[17] Edward G Coffman, Melanie Elphick, and Arie Shoshani. System deadlocks. *ACM Computing Surveys (CSUR)*, 3(2):67–78, 1971.

[18] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoos hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[19] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[20] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of ACM symposium on Cloud computing*, pages 143–154, New York, NY, USA, 2010.

[21] Carlo Curino, Evan Jones, Raluca Ada Popa, Nirmesh Malviya, Eugene Wu, Samuel Madden, Hari Balakrishnan, and Nickolai Zeldovich. Relational cloud: A database service for the cloud. In *5th Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2011.

[22] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 285–294, New York, NY, USA, 1999. ACM.

[23] Sudipto Das, Shashank Agarwal, Divyakant Agrawal, and Amr El Abbadi. Elastras: An elastic, scalable, and self managing transactional database for the cloud. Technical report, CS, UCSB, 03/2010 2010.

[24] Eduardo Cunha de Almeida, João Eugenio Marynowski, Gerson Sunyé, Yves Le Traon, and Patrick Valduriez. Efficient distributed test architectures for large-scale systems. In *ICTSS 2010: 22nd IFIP Int. Conf. on Testing Software and Systems*, Natal, Brazil, November 2010.

[25] Eduardo Cunha de Almeida, João Eugenio Marynowski, Gerson Sunyé, and Patrick Valduriez. Peerunit: a framework for testing peer-to-peer systems. In Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto, editors, *ASE*, pages 169–170. ACM, 2010.

[26] Eduardo Cunha de Almeida, Gerson Sunyé, Yves Le Traon, and Patrick Valduriez. Testing peers' volatility. In *23rd IEEE/ACM ASE*, pages 419–422, 2008.

[27] Eduardo Cunha de Almeida, Gerson Sunyé, Yves Le Traon, and Patrick Valduriez. Testing peer-to-peer systems. *Empirical Software Engineering*, 15(4):346–379, 2010.

[28] Murilo R. de Lima, Marcos Sfair Sunyé, Eduardo Cunha de Almeida, and Alexandre Ibrahim Direne. Distributed benchmarking of relational database systems. In Qing Li, Ling Feng, Jian Pei, Xiaoyang Sean Wang, Xiaofang Zhou, and Qiao-Ming Zhu, editors, *APWeb/WAIM*, volume 5446 of *Lecture Notes in Computer Science*, pages 544–549. Springer, 2009.

[29] Yuetang Deng, Phyllis Frankl, and David Chays. Testing database transactions with AGENDA. In *Proceedings of the 27th International Conference on Software Engineering*, pages 78–87. ACM, 2005.

[30] Yuetang Deng, Phyllis Frankl, and David Chays. Testing database transactions with agenda. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 78–87, New York, NY, USA, 2005. ACM.

[31] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A survey on model-based testing approaches: A systematic review. In *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, WEASELTech '07, pages 31–36, New York, NY, USA, 2007. ACM.

[32] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Lutess: A specification-driven testing environment for synchronous software. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 267–276, New York, NY, USA, 1999. ACM.

[33] Levente Ers and Tibor Csndes. An automatic performance testing method based on a formal model for communicating systems. In *IWQoS*, pages 1–5. IEEE, 2010.

[34] Anon et al. A measure of transaction processing power. In Michael Stonebraker, editor, *Readings in Database Systems, First Edition*. Morgan Kaufmann, 1985.

[35] Anon et al, Dina Bitton, Mark Brown, Rick Catell, Stefano Ceri, Tim Chou, Dave DeWitt, Dieter Gawlick, Hector Garcia-Molina, Bob Good, Jim Gray, Pete Homan, Bob Jolls, Tony Lukes, Ed Lazowska, John Nauman, Mike Pong, Alfred Spector, Kent Trieber, Harald Sammer, Omri Serlin, Mike Stonebraker, Andreas Reuter, and Peter Weinberger. A measure of transaction processing power. *Datamation*, 31(7):112–118, April 1985.

[36] Alessandro Gustavo Fior, Jorge Augusto Meira, Eduardo Cunha de Almeida, Ricardo Gonves Coelho, Marcos Didonet Del Fabro, and Yves Le Traon. Under pressure benchmark for ddbms availability. *JIDM*, 4(3):266–278, 2013.

[37] Martin Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

[38] G. Friedman, Alan Hartman, Kenneth Nagin, and T. Shiran. Projected state machine coverage for software testing. In *ISSTA*, pages 134–143, 2002.

[39] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[40] Vahid Garousi, Lionel C. Briand, and Yvan Labiche. Traffic-aware stress testing of distributed systems based on uml models. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 391–400, New York, NY, USA, 2006. ACM.

[41] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2002.

[42] Carlo Ghezzi and Amir Molzam Sharifloo. Verifying non-functional properties of software product lines: Towards an efficient approach using parametric model checking. *Software Product Line Conference, International*, 0:170–174, 2011.

[43] Martin Glinz. On non-functional requirements. In *15th IEEE International Requirements Engineering Conference, RE 2007, October 15-19th, 2007, New Delhi, India*, pages 21–26, 2007.

[44] Jim Gray. Why do computers stop and what can be done about it? In *Symposium on reliability in distributed software and database systems*, pages 3–12. Los Angeles, CA, USA, 1986.

[45] Jim Gray, editor. *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993.

[46] Jim Gray and Daniel P. Siewiorek. High-availability computer systems. *IEEE Computer*, 24(9):39–48, September 1991.

[47] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. Quickly Generating Billion-Record Synthetic Databases. *ACM Special Interest Group on Management of Data*, 23(2):243–252, May 1994.

[48] Mark Grechanik, B. M. Mainul Hossain, and Ugo Buy. Testing database-centric applications for causes of database deadlocks. In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, ICST '13, pages 174–183, Washington, DC, USA, 2013. IEEE Computer Society.

[49] B. M. Mainul Hossain, Mark Grechanik, Ugo Buy, and Haisheng Wang. Redact: preventing database deadlocks from application-based transactions. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *ESEC/SIGSOFT FSE*, pages 591–594. ACM, 2013.

[50] Michael S Hsiao, Elizabeth M Rudnick, and Janak H Patel. Application of genetically engineered finite-state-machine sequences to sequential circuit atpg. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(3):239–254, 1998.

[51] Ieee standards association, 2014. `http://standards.ieee.org`.

[52] Raj Jain. *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling.* Wiley professional computing. Wiley, 1991.

[53] Raj Jain. The art of computer systems performance analysis: Techniques for experimental design, measurement, simulation and modeling (book review). *SIGMETRICS Performance Evaluation Review*, 19(2):5–11, 1991.

[54] Evan P. C. Jones, Daniel J. Abadi, and Samuel Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD Conference*, pages 603–614, 2010.

[55] Evan Philip Charles Jones. *Fault-tolerant distributed transactions for partitioned OLTP databases*. PhD thesis, Massachusetts Institute of Technology, 2012.

[56] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

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

[58] David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[59] Huaiwei Liao and Marija Ilic. Stress test model of cascading failures in power grids. In *North American Power Symposium (NAPS), 2011*, pages 1–5. IEEE, 2011.

[60] Dino Mandrioli, Sandro Morasca, and Angelo Morzenti. Generating test cases for real-time systems from logic specifications. *ACM Trans. Comput. Syst.*, 13(4):365–398, November 1995.

[61] Aditya P. Mathur. *Foundations of Software Testing*. Addison-Wesley Professional, 1st edition, 2008.

[62] Jorge Augusto Meira, Eduardo Cunha de Almeida, Gerson Sunyé, Yves Le Traon, and Patrick Valduriez. Stress testing of transactional database systems. *JIDM*, (3):279–294.

[63] Memsql, 2014. http://www.memsql.com/.

[64] John Mylopoulos, Lawrence Chung, and Brian Nixon. Representing and using non-functional requirements: A process-oriented approach. *IEEE Transactions on Software Engineering*, 18:483–497, 1992.

[65] Nuodb, 2014. `http://www.nuodb.com/`.

[66] Marcelo Eidi Ochiai, Oto C. E. Gebara, Joatista Serro-Azuland La B. Pinto; Amit Nussbacher, Humberto Pierri, and Mauricio Wajngarten. Exercise stress test: prognostic value for elderly patients with stable coronary atherosclerosis. 2006.

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

[68] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31(6):676–686, June 1988.

[69] Hiroko Oura and Liliana B Schumacher. Macrofinancial stress testing-principles and practices. *International Monetary Fund Policy Paper*, 2012.

[70] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[71] Ioannis Parissis and Farid Ouabdesselam. Specification-based testing of synchronous software. *SIGSOFT Softw. Eng. Notes*, 21(6):127–134, October 1996.

[72] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, pages 165–178, New York, NY, USA, 2009. ACM.

[73] Steven C Port. Imaging guidelines for nuclear cardiology procedures. *Journal of nuclear cardiology*, 6(2):G47–G84, 1999.

[74] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2003.

[75] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th International Conference on Software Engineering*, ICSE '92, pages 105–118, New York, NY, USA, 1992. ACM.

[76] Suzanne Robertson and James Robertson. *Mastering the Requirements Process*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.

[77] Giancarlo Ruffo, Rossano Schifanella, Matteo Sereno, and Roberto Politi. Walty: A user behavior tailored tool for evaluating web application performance. In *Network Computing and Applications, 2004.(NCA 2004). Proceedings. Third IEEE International Symposium on*, pages 77–86. IEEE, 2004.

[78] Bernhard Rumpe. Model-based testing of object-oriented systems. In *In: Formal Methods for Components and Objects, International Symposium, FMCO 2002, Leiden. LNCS 2852*. Springer Verlag, 2003.

[79] Mahnaz Shams, Diwakar Krishnamurthy, and Behrouz Far. A model-based approach for testing the performance of web applications. In *Proceedings of the 3rd International Workshop on Software Quality Assurance*, SOQUA '06, pages 54–61, New York, NY, USA, 2006. ACM.

[80] Abraham Silberschatz, Henry Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., New York, NY, USA, 5 edition, 2006.

[81] Ian Sommerville and Gerald Kotonya. *Requirements Engineering: Processes and Techniques*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[82] Ahmed A. Soror, Umar Farooq Minhas, Ashraf Aboulnaga, Kenneth Salem, Peter Kokosielis, and Sunil Kamath. Automatic virtual machine configuration for database workloads. *ACM Trans. Database Syst.*, 35(1):7:1–7:47, February 2008.

[83] Gokul Soundararajan, Daniel Lupei, Saeed Ghanbari, Adrian Daniel Popescu, Jin Chen, and Cristiana Amza. Dynamic resource allocation for database servers running on virtual storage. In *Proccedings of the 7th conference on File and*

storage technologies, FAST09, pages 71–84, Berkeley, CA, USA, 2009. USENIX Association.

[84] Keith Stobie. Model based testing in practice at microsoft. *Electron. Notes Theor. Comput. Sci.*, 111:5–12, January 2005.

[85] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[86] Adam J. Storm, Christian Garcia-Arellano, Sam S. Lightstone, Yixin Diao, and M. Surendra. Adaptive self-tuning memory in db2. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 1081–1092. VLDB Endowment, 2006.

[87] David G. Sullivan, Margo I. Seltzer, and Avi Pfeffer. Using probabilistic reasoning to automate software tuning. *SIGMETRICS Perform. Eval. Rev.*, 32(1):404–405, June 2004.

[88] Test optimal, 2014. http://testoptimal.com/.

[89] The Institute of Electrical and Eletronics Engineers. Ieee standard glossary of software engineering terminology. IEEE Standard, September 1990.

[90] Transaction Processing Performance Council TPCC. Tpc benchmark c - standard specification revision 5.11. Technical report, 2010. `http://www.tpc.org/tpcc/spec/tpcc_current.pdf` , Acessado em 01/2013.

[91] Transaction Processing Performance Council TPCE. Tpc benchmark e - standard specification version 1.12.0. Technical report, 2010. `http://www.tpc.org/tpce/spec/v1.12.0/TPCE-v1.12.0.pdf` , Acessado em 01/2013.

[92] Dinh Nguyen Tran, Phung Chinh Huynh, Y. C. Tay, and Anthony K. H. Tung. A new approach to dynamic self-tuning of database buffers. *Trans. Storage*, 4(1):3:1–3:25, May 2008.

[93] Carolyn Turbyfill, Cyril U. Orji, and Dina Bitton. As^{3}ap - an ansi sql standard scaleable and portable benchmark for relational database systems. In Gray [45].

[94] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos. Management of an academic hpc cluster: The ul experience. In *Proc. of the 2014 Intl. Conf. on High Performance Computing & Simulation (HPCS 2014)*, Bologna, Italy, July 2014. IEEE.

[95] Lucas Vespa, Mini Mathew, and Ning Weng. P3fsm: Portable predictive pattern matching finite state machine. In *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, pages 219–222. IEEE, 2009.

[96] Marco Vieira and Henrique Madeira. A dependability benchmark for oltp application environments. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 742–753. VLDB Endowment, 2003.

[97] How voltdb works, 2014. `http://docs.voltdb.com/UsingVoltDB/IntroHowVoltDBWorks.php`.

[98] Voltdb, 2014. `http://voltdb.com/`.

[99] Inc. VoltDB. Technical overview - high performance, scalable rdbms for big data and real-time analytics. Technical report, VoltDB, Inc., 2014.

[100] Ronghua Wang, Naijun Sha, Beiqing Gu, and Xiaoling Xu. Comparison analysis of efficiency for step-down and step-up stress accelerated life testing. *Reliability, IEEE Transactions on*, 61(2):590–603, 2012.

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

[102] David Willmor and Suzanne M. Embury. An intensional approach to the specification of test cases for database applications. In *Proceedings of the 28th International Conference on Software Engineering*, pages 102–111. ACM, 2006.

[103] Michael M Wu and Michael C Loui. Modeling robust asynchronous communication protocols with finite-state machines. *Communications, IEEE Transactions on*, 41(3):492–500, 1993.

[104] C. Xiong. Inferences on a simple step-stress model with type-ii censored exponential data. *IEEE Transactions on Reliability*, 47:142 – 146, June 1998.

[105] Guang Yang, Jianhui Jiang, and Jipeng Huang. System modules interaction based stress testing model. *Computer Engineering and Applications, International Conference on*, 2:138–141, 2010.

[106] Ji Zhu, James Mauro, and Ira Pramanick. R-cubed (r3): Rate, robustness, and recovery - an availability benchmark framework. Technical report, Mountain View, CA, USA, 2002.