ALESSANDRO GUSTAVO FARIAS FIOR

UNDER PRESSURE BENCHMARK: A LARGE-SCALE AVAILABILITY BENCHMARK FOR DISTRIBUTED DATABASES

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

CURITIBA

2013

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A disponibilidade de um sistema refere-se a probabilidade dele estar totalmente operacional (*i.e.,* sem falhas) em um determinado momento [32]. Considerando que todo ambiente computacional é susceptível a falhas, especialmente os distribuídos, é de fundamental importância a utilização de mecanismos que provêem disponibilidade. Enquanto a maioria das aplicações tem como requisito estarem disponíveis (*i.e.,* totalmente operacionais) em 99,99% do tempo, as aplicações críticas (*e.g.*, telefonia, controle do espaço aéreo) exigem mais de 99,999% de disponibilidade [19]. Os Sistemas Gerenciadores de Banco de Dados Distribuídos (SGBDDs) implementam dois mecanismos em particular para atingir estas altas taxas de disponibilidade: replicação dos dados e particionamento. Porém, o uso destes mecanismos pode impactar negativamente o desempenho global do sistema. Este impacto precisa ser mensurado e compreendido para um bom planejamento do ambiente computacional. Nesta dissertação é apresentado o *Under Pressure Benchmark* (UPB) para avaliar os impactos causados pelos mecanismos de disponibilidade dos SGBDDs. Este *benchmark* é composto por quatro etapas nas quais o SGBDD é avaliado com diferentes configurações e cenários (incrementando a complexidade da avaliação, iniciando com o sistema estável até um ambiente com múltiplas falhas). O UPB oferece várias métricas para avaliação de desempenho que podem ser utilizadas para comparar diferentes configurações e SGBDDs de diferentes fabricantes. O UPB foi validado através de experimentos que avaliaram a disponibilidade do VoltDB [10], que é um SGBDD relacional baseado em memória, que atende as propriedades ACID e utiliza uma arquitetura *shared-nothing*.

ABSTRACT

Availability refers to the probability that a system is operational at a given point in time [32]. Considering that every computing environment is susceptible to failures, especially the distributed ones, it is rather important to implement availability mechanisms. While most systems require to be available (*i.e.*, fully operational) at 99.99% of the time, mission critical systems (*e.g.*, telephone networks, airspace control) require more than 99.999% of availability [19]. The Distributed Database Management Systems (DDBMS) implements two particular mechanisms to achieve those high availability rates: data replication and partitioning. However, the use of these mechanisms may badly impact the global performance as a side effect that must be measured for further comparisons or even performance analysis. This master dissertation presents the Under Pressure Benchmark (UPB) for evaluating the performance impact of the availability mechanisms of DDBMS. The benchmark consists of four steps in which the DDBMS is evaluated through different configurations and scenarios (increasing the evaluation complexity from a stable system scenario up to a faulty system scenario). UPB offers a range of metrics for performance measurements that could be used to compare different setup configurations or even DDBMS from different vendors. We validate our benchmark through experimentation and evaluation of the in-memory VoltDB [10] database, an ACID relational DDBMS based on shared-nothing architecture.

CAPÍTULO 1

INTRODUÇÃO

Um Banco de Dados Distribuído (BDD) é um conjunto de bancos de dados logicamente inter-relacionados e distribuídos em uma rede de computadores. O Sistema Gerenciador de Banco de Dados Distribuído (SGBDD) é o software que gere os BDDs e permite que a distribuição dos dados seja transparente para o usuário através do processamento de transações distribuídas. Os SGBDDs tem como potencial vantagem um desempenho linear no processamento de transações concorrentes devido a distribuição no acesso aos dados, além de facilitar a expansão do sistema através da adição de computadores [32] (i.e., escalabilidade horizontal).

A utilização dos SGBDDs trouxe a tona para a comunidade de Banco de Dados questões que antes eram de maior relevância para Sistemas Distribuídos, incluindo a transparência no acesso aos recursos distribuídos, a replicação destes recursos, e a disponibilidade dos dados diante de falhas que possam ocorrer durante sua execução. Neste trabalho nos concentramos em tratar e avaliar a disponibilidade de um BDD através de um método de avaliação de desempenho. Por definição [19] disponibilidade refere-se a probabilidade de um sistema estar operacional em um determinado momento do tempo. Nessa definição deve-se considerar que qualquer sistema computacional sempre está sujeito a falhas de diferentes naturezas, sejam elas causadas por fatores internos (*e.g.*, erros de *software*, problemas de *hardware*) ou externos (*e.g.*, falhas humanas, falta de energia, catástrofes ambientais) [39].

No contexto dos SGBDDs a definição de disponibilidade pode ser extendida, sem perdas para a definição original, para a capacidade de um SGBDD gerenciar suas falhas [32] de forma que gere o menor impacto possível para a aplicação que o utiliza. Gerenciar falhas, nesse contexto, está relacionado a diversas capacidades, como:

- Manter o sistema operacional mesmo com a ocorrência de determinado nível de

falha;

- Recuperar o sistema de forma eficaz e eficiente após a ocorrência de falhas;
- Garantir que nenhuma informação seja perdida, ou seja, garantir a durabilidade dos dados.

Os SGBDD possuem dois principais mecanismos para garantir a disponibilidade de BDDs: replicação e particionamento dos dados. Porém a utilização desses mecanismos tem um custo no desempenho global relacionado ao aproveitamento da capacidade de *hardware*. Consequentemente, quanto mais estes mecanismos sejam utilizados, maior será o impacto no desempenho global do SGBDD [27].

Tipicamente a maioria das aplicações tem como requisito estarem disponíveis (*i.e.,* totalmente operacionais) em mais de 99,99% do tempo [19], o que corresponde a aproximadamente 50 minutos de indisponibilidade durante o ano. Já nas aplicações críticas, como por exemplo na área de telefonia, esse requisito aumenta para 99,999% do tempo, ou seja, menos de 5 minutos por ano de indisponibilidade. Para atingir estas taxas de disponibilidade é necessário, portanto, a replicação e o particionamento dos dados [19] considerando, como efeito colateral, a perda no desempenho global. Então é fundamental para um bom planejamento do ambiente computacional avaliar e compreender os impactos causados pelo aumento da disponibilidade.

A avaliação de desempenho é conhecida pelo termo inglês *benchmarking* [21] que pode ser traduzido como aferição ou análise comparativa. Em *benchmarking* são simuladas cargas de trabalho e gerados volumes de dados próximos do observado no mundo real. Em seguida, métricas são calculadas para indicar a eficiência de aspectos diversos, incluindo desempenho, disponibilidade e escalabilidade. Para que as métricas geradas possam ser comparáveis entre diferentes sistemas, é necessário que o *benchmarking* seja feito seguindo métodos de avaliação (do inglês *benchmark*) bem definidos [21]. Como os termos *benchmarking* e *benchmark* são amplamente utilizados e aceitos pela comunidade acadêmica eles serão utilizados nesse trabalho.

Há mais de 30 anos a área de *benchmarking* de SGBD tem atraído a atenção de

pesquisadores e indústria. Diversos *benchmarks* foram propostos com o objetivo de fazer comparações entre produtos de diferentes fabricantes e dar subsídios técnicos para a escolha do melhor SGBD dentro do contexto da sua aplicação. A maior parte dos *benchmarks* são voltados para avaliar exclusivamente o desempenho dos SGBD, não levando em consideração os mecanismos de disponibilidade de BDDs.

O crescimento na utilização de sistemas distribuídos, sejam voltados para armazenamento de dados como para processamento, fomentou o interesse da comunidade acadêmica e indústria por novos *benchmarks*. O *Yahoo! Cloud Serving Benchmark* (YCSB) [16] é uma das mais recentes plataformas para *benchmark* de SGBDD que leva em consideração as questões de escalabilidade e desempenho. Apesar desse trabalho citar a importância da disponibilidade, a metodologia para avaliá-la aparece como trabalho futuro e ainda ainda não foi desenvolvida. Alguns *benchmarks* de disponibilidade de SGBDs foram apresentados em [39, 26], porém eles não são voltados para SGBDDs. Pelo nosso conhecimento, nenhum *benchmark* para avaliação da disponibilidade específica de SGBDDs foi proposto até a data de conclusão deste trabalho.

Nesta dissertação apresentamos o método de avaliação de disponibilidade de SGBDDs *Under Pressure Benchmark* (UPB). O objetivo do UPB é medir a eficiência e os impactos dos mecanismos utilizados para prover e manter a disponibilidade de SGBDDs.

No UPB o SGBDD é avaliado em quatro situações distintas: (1) Sem utilização de mecanismo de tolerância a falhas, (2) com mecanismos de tolerância mas sem falhas, (3) com falhas dentro do nível tolerado, (4) em recuperação das falhas ocorridas. Para cada uma dessas situações são geradas métricas parciais que indicam a eficiência do SGBDD. Ao final as métricas parciais são reunidas para representar o desempenho da disponibilidade do SGBDD.

Para validar o UPB, avaliamos o SGBDD VoltDB que é baseado na arquitetura *shared-nothing* onde os nós computacionais não compartilham recursos entre si. Os experimentos foram realizados nas máquinas do projeto francês Grid'5000 [2]. Os resultados obtidos, além de demonstrar a viabilidade do UPB e avaliar a disponibilidade do SGBDD VoltDB, contém descobertas sobre a operação do VoltDB em diferentes situações não relatadas em

sua documentação oficial.

No Capítulo 2 deste trabalho serão apresentados conceitos gerais relevantes e relacionados ao UPB, como SGBDDs, disponibilidade e termos relacionados, além de uma visão geral sobre o VoltDB. Em seguida, no Capítulo 3 é feita uma revisão bibliográfica contendo os principais trabalhos relacionados a *benchmarking* e o histórico da área, começando pelos primeiros benchmarks de desempenho até os últimos estudos sobre *benchmarking* de disponibilidade. O UPB será apresentado em detalhes no Capítulo 4, onde todos os pontos do *benchmark* serão discutidos, incluindo a arquitetura do ambiente de avaliação, a carga de trabalho e as métricas. Por fim, os experimentos realizados que validam o UPB através da avaliação da disponibilidade do VoltDB são descritos no Capítulo 5.

CAPÍTULO 2

CONCEITUAÇÃO

Neste Capítulo são abordados conceitos relacionados ao *benchmark* de disponibilidade apresentado nesse trabalho. A Seção 2.1 introduz os conceitos relacionados à Sistemas Gerenciadores de Banco de Dados Distribuídos (SGBDDs), classifica seus tipos e descreve suas respectivas características. Questões gerais relacionadas à Disponibilidade e Dependabilidade são discutidas na Seção 2.2 e na Seção 2.3 estas questões são abordadas no contexto de SGBDD. Por fim, a Seção 2.4 encerra este Capítulo através do detalhamento do SGBDD VoltDB como um estudo de caso.

2.1 Sistemas Gerenciadores de Banco de Dados Distribuídos

Os Bancos de Dados Distribuídos (BDD) surgiram da junção de dois conceitos aparentemente distintos, Banco de Dados e Redes de Computadores [32]. Se, por um lado, os bancos de dados procuram a centralização dos dados, as redes de computadores tem como foco a distribuição deles. À primeira vista, considerando os objetivos de cada um, essa união pode aparentar antagônica. Porém, o que de fato a área de BDs busca é a gestão dos dados, que é realizada tradicionalmente através de processamento centralizado. Naturalmente, nos BDDs a abordagem empregada é baseada na distribuição do processamento.

A distribuição do processamento pode ocorrer em vários níveis, desde do paralelismo que acontece no *pipeline* do processador, das trilhas sobrepostas dos discos magnéticos, ou até do ponto de vista de rede de computadores, em que dados correlatos são armazenados em um mesmo nó da rede a fim de se evitar sobrecarga no tráfego. Sobre "Distribuição de processamento" pode-se definir como *uma quantidade de elementos processantes autônomos (não necessariamente homogêneos) que são interconectados por uma rede de computadores e que cooperam entre si pela execução de suas tarefas* [32].

Ao distribuir a informação em diversos agentes de processamento que sejam capazes de processar de forma independente e paralela aumenta-se o desempenho e a confiabilidade, evitando que o ambiente fique vulnerável a um único ponto de falha.

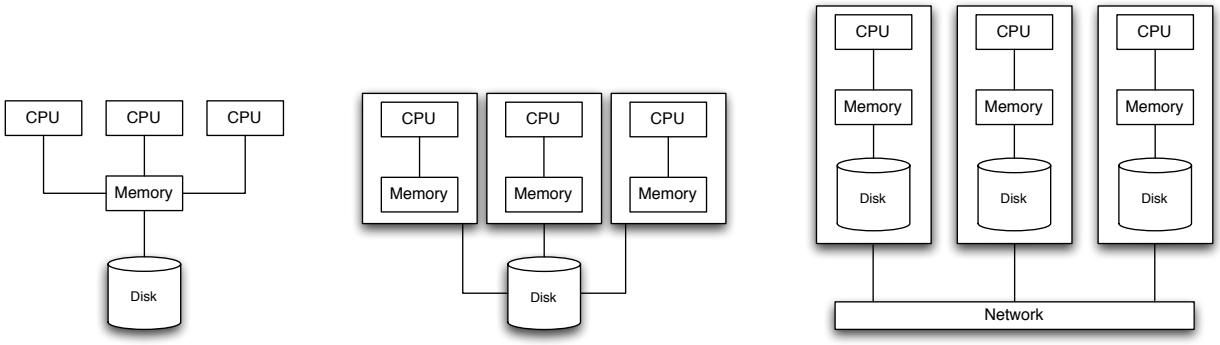
Aplicando distribuição de processamento em banco de dados, chega-se a **Banco de Dados Distribuídos (BDD)**, que pode ser definido como em [32] "*uma coleção de múltiplos banco de dados logicamente inter-relacionados sobre uma rede de computadores*". Já um **Sistema Gerenciador de Banco de Dados Distribuídos (SGBDD)** é definido como "*o sistema de software que auxilia no gerenciamento dos bancos dados distribuídos, fazendo que a distribuição seja empregada de forma transparente aos usuários*".

BDDs podem seguir 3 arquiteturas, ilustradas na Figura 2.1:

- *Shared memory (tightly coupled)*: Sistemas com memória compartilhada, geralmente empregado em sistemas OLTP. Exemplos de produtos incluem os SGBDs Oracle, DB2, PostgreSQL, MySQL e Sybase;
- *Shared disk (loosely coupled)*: Sistemas com compartilhamento de disco. Modelo mais simples, pois todos os nós podem acessar o dado. Exemplos de SGBDs: Oracle RAC e Sybase-IQ;
- *Shared nothing*: Sistemas que requerem um cluster de processamento cujos nós não compartilham nenhum recurso e se comunicam através de uma rede. Exemplos de SGBDs: VoltDB, Teradata, Greenplum e Tandem.

Outro aspecto importante para Sistemas Gerenciadores de Banco de Dados são as propriedades **ACID**, que podem ser resumidas da seguinte forma:

- **Atomicidade**: Garante que todo grupo de operações, denominado transação, seja executado por completo; caso uma operação falhe, todas as demais pertencentes ao mesmo grupo são canceladas;
- **Consistência**: Garante que o que o banco de dados saia de um estado válido para outro estado válido após a execução da transação;



(a) Shared memory (b) Shared disk (c) Shared nothing

Figura 2.1: Arquiteturas de BDDs.

- **Isolamento**: Garante a concorrência no uso do Banco de Dados. Através do isolamento o processamento de uma transação não interfere em outra, de tal forma que se as transações paralelas fossem executadas de forma sequencial, o resultado seria o mesmo (i.e., conceito da "serialidade");

- **Durabilidade**: Garante que uma vez que a transação tenha sido finalizada, os dados estarão armazenados corretamente e não irão se perder.

Tipicamente as propriedades ACID são requisitos obrigatórios em SGBDs ditos "transacionais". Contudo, alguns desses requisitos podem ser relaxados em troca de vantagens em outras propriedades, como desempenho e dependabilidade. Sob o contexto de SGBDDs, a distribuição do ambiente exige a garantia de diversas propriedades que vão além das ACID afim de que o sistema possa processar consultas de forma confiável e atendendo os requisitos diversos da aplicação para o qual é destinado.

2.2 Dependabilidade, Disponibilidade e Confiabilidade

Frequentemente os termos dependabilidade, disponibilidade e confiabilidade são utilizados na literatura com significados muito próximos, quase como sinônimos. Mesmo entre pesquisadores da área, as definições para esses termos variam. Nessa seção serão apresentadas as definições adotadas para esse trabalho.

Segundo [14], sistemas computacionais em geral podem ser caracterizados por quatro

propriedades fundamentais: Funcionalidade, desempenho, custo e dependabilidade. A **Dependabilidade** [14] é descrita como um agregado de propriedades adicionais que, quando considerados em conjunto, expressam que um determinado sistema é capaz de, justificadamente, oferecer um serviço confiável. Para isso, tal sistema deve promover as seguintes propriedades adicionais:

- **Disponibilidade**: Prontidão para correto funcionamento em um determinado momento.
- **Confiabilidade**: Continuidade do correto funcionamento durante um período.
- **Confidencialidade**: Impossibilidade de acessos não autorizados aos dados.
- **Integridade**: Impossibilidade de alteração do sistema para um estado inválido durante a sua utilização.
- **Manutenibilidade**: Habilidade de manutenção sobre o sistema, tanto reparos como modificações.
- **Segurança**: Impossibilidade de consequências catastróficas (sobre o usuário, dados e ambiente em geral) durante a operação do sistema. Pode ser vista como a existência conjunta da disponibilidade, confidencialidade e integridade.

A ênfase em cada um dessas propriedades deve ser maior ou menor conforme a aplicação do sistema em questão. Por exemplo, a disponibilidade é uma propriedade que é sempre requerida, ainda que em maior ou menor grau. Já outras, como confiabilidade, segurança, confidencialidade e manutenibilidade podem ser ou não requeridas conforme a aplicação.

De qualquer forma, estas propriedades devem sempre ser vistas de forma relativa e probabilística, mas não absoluta. Devido a inevitável existência das falhas um sistema nunca é totalmente disponível, confiável ou seguro [14].

Dentre as propriedades da dependabilidade, o foco deste trabalho está na confiabilidade e disponibilidade. Segundo [32], **confiabilidade** refere-se a probabilidade do sistema

em questão não apresentar nenhuma falha durante um determinado intervalo de tempo. Esta propriedade é tipicamente utilizada para descrever sistemas que não são capazes de ser reparados ou cuja operação é tão crítica que absolutamente nenhum período de tempo *offline* é tolerado.

Já a **disponibilidade** [32] refere-se a probabilidade de um sistema estar totalmente operacional, conforme a sua especificação, em um determinado ponto no tempo t. Mesmo que falhas tenham ocorrido anteriormente a esse momento t, se elas forem corretamente reparadas o sistema estará disponível em t.

A disponibilidade pode ser usada como uma medida da capacidade de um sistema (no sentido de "quanto bom ele é") em reparar suas falhas, permanecendo indisponível (*offline*) durante um curto período de tempo para recuperação [32]. Esta definição, adotada em nosso trabalho, pode ser extendida no contexto de sistemas distribuídos para a capacidade do sistema de gerenciar falhas, incluindo as consequências delas para outras propriedades do sistema, como o desempenho.

Considerando essas definições pode-se observar que os objetivos de confiabilidade e disponibilidade são, em parte, contraditórios. Enquanto confiabilidade refere-se a inexistência absoluta de falhas, a disponibilidade considera a recuperação de falhas para que o sistema esteja operacional em um determinado momento. Como em ambientes reais a ocorrência de falhas é inevitável, a disponibilidade é uma propriedade tipicamente de maior importância.

2.3 Disponibilidade em SGBDDs

Uma das potenciais vantagens dos SGBDDs quando comparados aos SGBDs centralizados é com relação a confiabilidade e disponibilidade [32]. Devido a distribuição dos bancos de dados, mesmo na ocorrência de falhas que impossibilitem a operação de alguns nós, os nós remanescentes não são afetados e seus dados permanecem consistentes. Porém, para desfrutar dessas vantagens, os SGBDDs devem ser dotados de mecanismos que garantam a consistência do BDD mesmo com a indisponibilidade de alguns dos seus nós e que sejam capazes de detectar, tolerar e se recuperar de falhas de maneira eficiente.

No contexto de SGBDDs, a confiabilidade pode ser traduzida como a sua capacidade de permanecer operacional, atendendo requisições dos usuários sem violar a consistência do banco de dados, mesmo quando componentes do seu ambiente apresentam falhas [32]. Essa capacidade, chamada de **tolerância a falha**, é uma propriedade importante da disponibilidade de SGBDDs.

Os principais conceitos [32] relacionados a tolerância a falhas em sistemas distribuídos são modularização da arquitetura, redundância dos módulos do sistema (i.e., nós) e replicação dos dados.

A modularização da arquitetura [19] prevê que em sistemas tolerantes a falhas todos os módulos devem ser implementados de forma que seu funcionamento seja independente. No contexto de SGBDDs, cada nó pode ser visto como um módulo. Além disso, os nós devem ser capazes de monitorar o seu funcionamento e na ocorrência de qualquer falha interna, o nó inteiro deve interromper sua execução imediatamente para evitar a propagação de erros.

A redundância e a replicação dos dados são recursos utilizados por sistemas distribuídos para evitar a existência de pontos únicos de falhas. Todos os módulos de sistemas distribuídos tolerantes a falhas devem ser redundantes, ou seja, deve existir pelo menos mais um módulo similar no sistema capaz de executar as mesmas funções e substituí-lo quando necessário. No contexto de SGBDDs, em geral todos os nós executam os mesmos processos computacionais, tendo os mesmos recursos e capacidades (*e.g.*, processamento de transações e gerenciamento de conexões com clientes).

A **replicação dos dados** em SGBDDs consiste em manter cópias dos dados em diferentes nós do cluster. Esse recurso é utilizado para que nenhuma informação seja perdida ou fique indisponível durante a ocorrência de falhas em alguns nós. Para tirar proveito da replicação dos dados, o SGBDD deve ser capaz de detectar a ocorrência de falhas e automaticamente redirecionar as transações para os nós disponíveis e que possuem cópias dos dados. O SGBDD também é responsável por manter a sincronia dos dados entre as diversas réplicas, de forma que o BDD esteja sempre consistente.

Apesar dos benefícios de disponibilidade garantidos pela replicação, garantir a con-

sistência de dados entre as réplicas requer métodos computacionalmente caros, que podem gerar perdas de desempenho do SGBDD. Uma estratégia utilizada para diminuir esse impacto é o relaxamento na propriedade de consistência do banco de dados. O critério de replicação adotado pode ser de dois tipos: (1) Forte (síncrono), que prevê que as réplicas sejam mantidas sempre sincronizadas, ou (2) fraco (assíncrono), que não requer a sincronia imediata entre as réplicas após a atualização dos dados e prevê apenas que eventualmente as réplicas se tornarão idênticas.

Tipicamente aplicações OLTP, como por exemplo aplicações bancárias, tem como requisito consistência total do banco de dados e requerem estratégias de replicação fortes. Já outros tipos de aplicações, como por exemplo redes sociais na internet, podem utilizar requisitos de consistência mais fracos com a vantagem de obter maior desempenho.

Alguns SGBDDs permitem configurar a rigidez na propriedade de consistência do banco de dados. De forma geral, quanto maior a rigidez maiores são os impactos no desempenho do sistema. Também é possível configurar o nível de redundância desejado, ou seja, a quantidade de nós que podem falhar sem que a operação do sistema seja interrompida.

Outro fator importante da replicação é o seu custo, uma vez que para a manutenção de réplicas é necessário a utilização de recursos extras de *hardware* e *software*. A manutenção de uma réplica, por exemplo, requer o dobro da quantidade de nós e espaço de armazenamento. Isso aumenta significativamente o custo do ambiente como um todo.

Os mecanismos de tolerância a falha oferecem proteção do sistema contra a perda de uma quantidade configurável de nós do cluster. Porém os sistemas estão sujeitos a falhas de maiores proporções, quando uma quantidade de nós maior do que a tolerada fique indisponível. Por esse tipo de situação, a disponibilidade de SGBDDs também está diretamente relacionada ao conceito de **durabilidade**. As estratégias tipicamente adotadas nessa situação consistem na utilização de memória permanente, como disco, para armazenamento dos dados. Porém o desempenho desse tipo de memória é significativamente menor do que da memória principal, então, por esse motivo, cada SGBDD tem recursos e estratégias específicas para minimizar o impacto no desempenho geral (*e.g.*, backups

assíncronos periódicos, logs de transações).

Outro aspecto da disponibilidade em SGBDDs é a **recuperação de falhas**, que pode ser de dois tipos:

- Com o sistema *online*, após a ocorrência de uma falha que foi tolerada pelo SGBDD. Nesse caso o nó que apresentou falha, após a correção dos seus problemas, precisa ser reinserido no cluster. Esse processo de reinserção requer a cópia de uma parte dos dados do banco para o novo nó, o que pode gerar sobrecarga da rede de comunicação e instabilidade no desempenho do SGBDD.

- Com o sistema *offline*, após a ocorrência de falhas de grandes proporções. Nesse caso a recuperação deve ser baseada nos mecanismos específicos utilizadas para garantia de durabilidade, como, por exemplo, recuperação de backups e execução de logs de transações.

Em ambos os tipos de recuperação, o tempo necessário para que um nó seja reinserido no cluster e esteja operacional, pronto para processar transações, é um fator importante para a disponibilidade.

2.4 Estudo de caso: VoltDB

O VoltDB [10] é um SGBD Distribuído baseado em clusters do tipo *shared-nothing* que tem como principal diferencial o seu alto desempenho quando comparado a SGBDs tradicionais. Entre as suas características pode-se destacar [10, 42]:

- É um SGBDD transacional que atende os requisitos ACID (Atomicidade, consistência, isolamento e durabilidade).

- Utiliza armazenamento de dados em memória principal

- Permite que os dados sejam particionados horizontalmente ou replicados

- Permite escalabilidade próxima da linear com a adição de nós no cluster.

- Oferece mecanismos para prover confiabilidade e alta disponibilidade.

O acesso aos dados no VoltDB é feito através de *Stored Procedures*, que são códigos Java com instruções SQL incorporadas. Apesar do VoltDB permitir a execução de SQL diretamente a recomendação é que, por questões de desempenho, todo o acesso ao banco de dados seja feito através de *Stored Procedures*. Essas procedures podem ser extremamente simples, como um único comando SQL, ou bastante complexas, com implementação de regras de negócio e vários comandos SQLs. No contexto do VoltDB, as *Stored Procedures* são tratadas como transações e seguem as propriedades de atomicidade e isolamento.

Os nós de um cluster VoltDB podem abrigar um ou mais *sites*, que são as unidades de processamento do SGBDD, responsáveis por processar transações e armazenar dados. Como os dados do VoltDB são particionados, e cada partição é atribuída para um site, os dois termos (site e partição) podem ser utilizados para se referir a mesma unidade de processamento.

2.4.1 Arquitetura

A alta performance do VoltDB [42, 41] se deve a uma arquitetura que tira proveito dos modernos ambientes computacionais. Em [20], os autores analisaram o funcionamento dos SGBDs convencionais e executando a transação *New Order* do TPC-C identificaram que apenas 12% dos ciclos de CPU foram utilizados para o processamento da transação propriamente dita, enquanto os 88% restantes foram gastos com outros processos internos do SGBD, como gestão de logs, *locks*, *buffers*, índices e *deadlocks*. A partir dessa constatação, Stonebraker *et al.* propuseram em [20] e [22] uma arquitetura que reduz significativamente esse custo computacional extra e, utilizando armazenamento em memória principal, é capaz de obter um desempenho muito superior ao dos SGBDs tradicionais.

Com o banco de dados sendo mantido inteiro em memória principal, ganha-se desempenho em dois aspectos: (1) O acesso aos dados é muito mais rápido já que evita-se as custosas operações de acesso ao disco e (2) não é necessário a existência de *buffer* e *tablespaces*, o que em outros SGBDs gera um custo computacional adicional.

Outro aspecto fundamental para o alto desempenho do VoltDB é que o processamento de transações dentro de cada partição é executado de forma serial por uma única linha de execução (*thread*) autônoma [41] . Cada partição tem sua própria fila de chamadas de transações, que são executadas uma a uma. Com isso elimina-se a necessidade de gestão de *locks* e *deadlocks*. Como cada partição é capaz de processar apenas uma transação de cada vez, a escalabilidade do desempenho do cluster pode ser feito de forma quase linear com a adição de mais partições.

Os bancos de dados do VoltDB são otimizados para a aplicação específica para a qual ele é destinado através da compilação do seu *schema* e *Stored Procedures*. Com essas informações o VoltDB é capaz de particionar os dados juntamente com todo o seu processamento associado e atribuir para nós individuais do cluster. Dessa forma, cada partição só realiza o processamento das transações cujos dados estão armazenados nela.

Quando uma transação requer dados de múltiplas partições, um nó atua como coordenador e é responsável por direcionar o trabalho para os outros nós, coletar os resultados e completar a transação. Esse processo faz com que transações multi-partições sejam mais lentas, porém a sua integridade é garantida e o desempenho geral do SGBDD não é afetado já que não existem *locks* e *deadlocks*.

O particionamento das tabelas do VoltDB é feito através da sua chave primária. E cabe ao desenvolvedor escolher boas chaves primárias para as suas tabelas de forma que evitem a ocorrência de transações que necessitem acessar múltiplas partições. Entretanto o VoltDB também permite a replicação da mesma tabela em diferentes partições. Esse recurso pode ser utilizado por tabelas muito acessadas para evitar a necessidade de transações multi-partições.

É interessante observar que o VoltDB prioriza a vazão do cluster (quantidade de transações por segundo processadas) em detrimento da latência individual das transações. Apesar disso, segundo o fabricante [42] a latência geral do VoltDB é semelhante a outros SGBDs, porém a sua vazão é significativamente maior.

2.4.2 Garantia das propriedades ACID

As propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade) são fundamentais para diversos tipos de aplicação. O VoltDB garante essas propriedades, ao mesmo tempo em que oferece um alto desempenho, da seguinte forma:

- **Atomicidade**: As transações do VoltDB são as *Stored Procedures*, e seguem os mesmos conceitos clássicos de transação em SGBD. O processamento das Stored Procedures deve ser concluído por inteiro com sucesso para que as alterações sejam transparentes para todo o banco de dados (*commit*) ou falham por inteiro e o banco de dados volta ao estado original (*rollback*).

- **Consistência**: O VoltDB requer a utilização de esquema e tipos de dados bem definidos.

- **Isolamento**: As transações são ordenadas globalmente no VoltDB e são executadas até sua total conclusão sem escalonamentos, intercalações e locks em todas as partições afetadas.

- **Durabilidade**: Diversos recursos estão disponíveis no VoltDB para aumentar a disponibilidade de forma geral, incluindo a durabilidade. Esses recursos são apresentados na Seção 2.4.4.

2.4.3 Uso de memória

A memória total utilizada pelo servidor VoltDB recebe o nome de RSS (*Resident set size*) que é dividida em três partes distintas:

- **Memória Persistente**: Armazena os dados e metadados do BD, como definições de tabelas, índices e visões. Quando maior for o banco de dados, maior a porção de memória persistente necessária.

- **Memória Semi-Persistente**: Utilizada como área de memória temporária para auxiliar diversas operações do SGBD. Nela são armazenadas as tabelas temporárias

resultantes de transações que estão em processamento e auxiliam a realização de operações como junção e ordenação, além de buffers de operações de administração do SGBD (*snapshot* e exportação de dados) e resultados de transações que ainda não foram confirmadas (i.e., *commit*) ou anuladas (i.e., *rollback*).

- **Memória Temporária**: É onde ficam armazenadas a fila de chamadas de transações pendentes para serem processadas e os resultados retornados pelas transações até que sejam recuperados pelos clientes. Essa memória é armazenada na *Heap* do Java.

2.4.4 Disponibilidade

A disponibilidade no VoltDB é garantida por três recursos principais que serão apresentados nessa seção: Replicação de partições (*K-Safety*), Snapshot e Logs de Comando.

2.4.4.1 *K-Safety*

O funcionamento do *K-Safety* consiste em duplicar partições do banco de dados de forma que, mesmo na ocorrência de falhas em alguns nós, o serviço possa ser mantido *online* e nenhum dado seja perdido. Essas partições replicadas são membros normais do cluster, totalmente funcionais e ativas na execução das transações [42].

O conceito de *K-Safety* está relacionado a replicação ao nível da partição dos nós, e não do cluster completo. Ou seja, o cluster não é totalmente replicado em uma localização diferente, como acontece em SGBDs de outros fabricantes. Com o *K-Safety* a replicação acontece dentro do próprio cluster, onde todos os dados são copiados uma ou mais vezes em diferentes partições pertencentes a diferentes nós. Dentro dessa arquitetura todas as partições replicadas são exatamente iguais, tendo a mesma importância e funcionalidade. Não existem partições principais e cópias secundárias, por exemplo.

A estratégia de manutenção das réplicas utilizada pelo *K-Safety* é apresentada na Figura 2.2. Quando o cluster está completo, sem falhas, as transações são enviadas para todas as cópias da partição. Como todas as réplicas operam simultaneamente a consistência entre elas é garantida. E quando um nó fica indisponível o SGBDD continua

enviando as transações para as cópias que permanecem *online*.

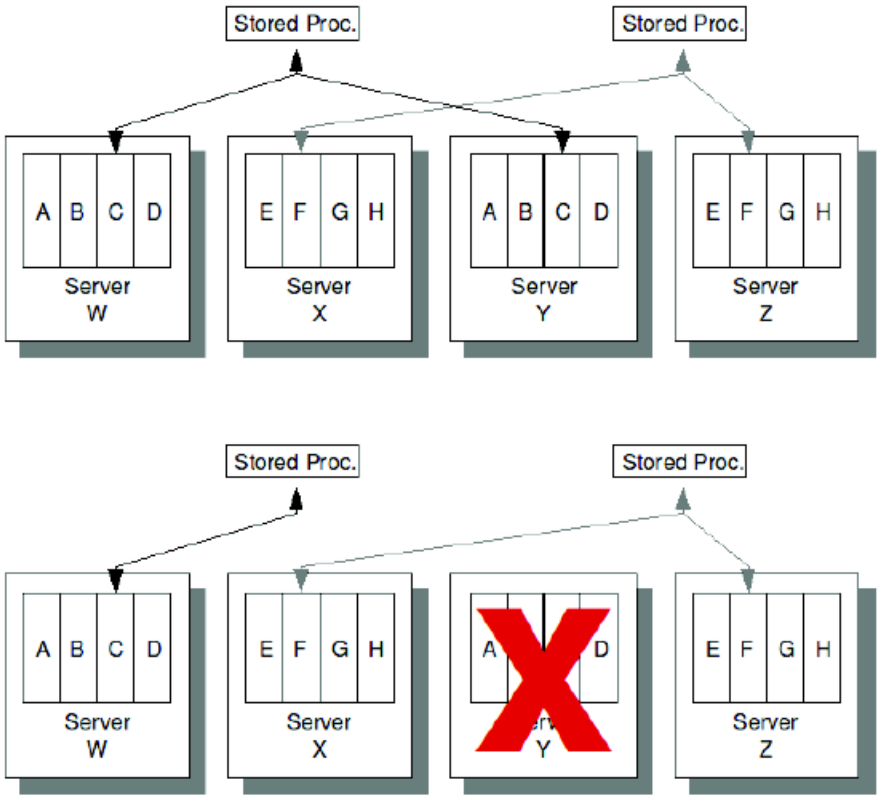


Figura 2.2: Funcionamento do *K-Safety* [42].

Para configurar o *K-Safety* no VoltDB deve-se definir o valor de K, que indica quantas vezes as partições serão duplicadas. Para $K = 1$ todas as partições serão duplicadas uma vez, o que significa que no caso de falha de um nó o SGBD ainda permanecerá *online*. Analogamente, para $K = 2$, serão necessárias duas cópias de cada partição e o cluster será capaz de tolerar dois nós indisponíveis.

Para garantir a tolerância a falha configurado pelo K, o VoltDB distribui as partições replicadas em nós diferentes. Para fazer isso ele calcula o número máximo de partições únicas e determina a melhor distribuição das partições entre os nós. A documentação do VoltDB recomenda que, para atingir a máxima utilização do hardware, o número total de partições seja um múltiplo de $K + 1$.

Um ponto importante sobre o *K-Safety* é relacionado ao seu impacto no desempenho e capacidade do cluster. Quando utilizamos a replicação, estamos dividindo as partições disponíveis (e consequentemente os seus recursos computacionais) entre as cópias. Ou

seja, o desempenho e capacidade de armazenamento do cluster diminuem conforme se aumenta a replicação.

O desempenho do cluster também pode ser impactado no momento em que um nó está sendo reinserido no cluster após uma falha. Nessa situação, o nó que está entrando na rede recebe uma cópia dos dados a partir de outros nós. A duração dessa operação depende da quantidade de dados que devem ser transferidos e da velocidade da rede. Entretanto esse procedimento é tratado pelo VoltDB como uma transação única que bloqueia as demais na partição que está fornecendo os dados. Durante esse procedimento, as transações recebidas pelo cluster são colocadas em uma fila, o que pode aumentar a latência e desempenho do cluster.

2.4.4.2 *Snapshot*

Um *snapshot* é um conjunto de arquivos, salvos em memória permanente, que contém um registro completo do banco de dados em um determinado momento. Como os dados do VoltDB ficam armazenados na memória principal durante a sua execução, os *snapshots* são utilizados como backups não voláteis. Eles são úteis tanto para paradas programadas no sistema para manutenção como para recuperação de falhas inesperadas.

O VoltDB permite configurar a criação automática de *snapshots* com intervalos de tempo configuráveis. No caso de falhas de grandes proporções, onde o cluster inteiro seja parado (e os dados perdidos, já que são armazenados em memória principal), os *snapshots* permitem a recuperação do banco de dados para um estado válido anterior à falha. Porém as transações executadas após o último *snapshot* não estarão inclusas e serão perdidas. Para recuperá-las existe outro recurso chamado Log de Comando.

2.4.4.3 Log de Comando

O Log de Comando (*Command Logging*) mantém um registro de todas as transações executadas no VoltDB desde o último *snapshot*. Ele é um mecanismo de recuperação de falhas que garante a propriedade de durabilidade das transações. Esse recurso só está disponível no *VoltDB Enterprise Edition*.

Um ponto importante do Log de Comando é que ele armazena as chamadas das transações e não os seus resultados. Dessa forma o tamanho do log é reduzido, diminuindo a quantidade de operações de E/S de disco. Entretanto toda operação em disco impacta no desempenho do sistema. Por esse motivo o log de comando possui diversas configurações que devem ser feitas conforme o cenário em que está sendo utilizado, calibrando suas vantagens e desvantagens.

Quando habilitado, o Log de Comando do VoltDB passa a armazenar todas as chamadas de transações executadas pelo cluster. Para manter bom desempenho, a forma e a frequência com que o log é escrito no disco é configurável. No disco esse log será incrementado continuamente até que atinja determinado tamanho, quando o VoltDB executará automaticamente um snapshot do banco. Nesse momento entradas do arquivo de log podem ser apagadas, mantendo-se apenas as transações executadas após o snapshot. Esse comportamento é apresentado na Figura 2.3.

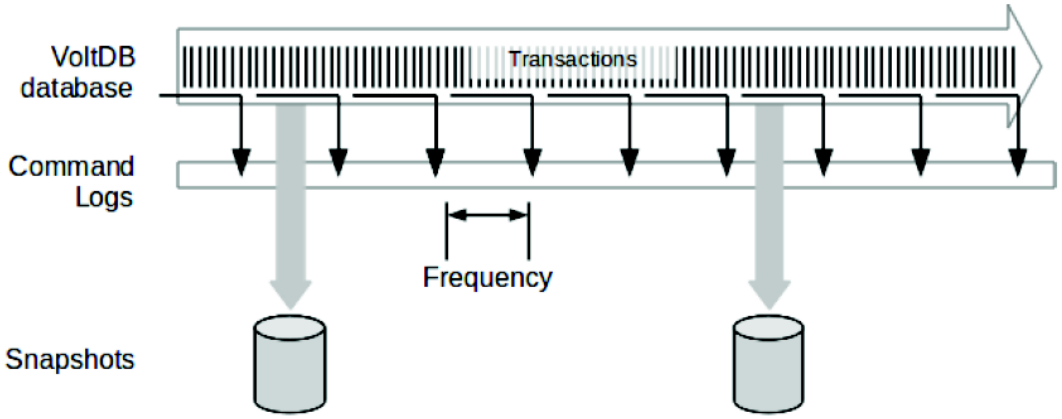


Figura 2.3: Funcionamento do Log de Comando do VoltDB [42].

Existem três configurações principais do Log de Comando que devem ser feitas buscando o melhor equilíbrio entre desempenho e durabilidade:

- **Log Síncrono ou Assíncrono:** Quando configurado de forma assíncrono o impacto do log de comando no desempenho do cluster é mínimo. O cluster pode retornar a transação para a aplicação antes do log ser gravado em disco. O ganho de desempenho tem a desvantagem de diminuir a confiabilidade, uma vez que no caso de falhas as transações executadas depois da última operação de escrita serão

perdidas. Nos casos em que nenhuma perda de dados pode ser tolerada, então é necessário utilizar logs síncronos. O impacto direto disso é na latência das transações já que elas só serão retornadas após a operação de escrita em disco.

- **Frequência:** No caso de log Assíncrono, esse parâmetro indica a frequência com que as transações devem ser escritas em disco.

- **Tamanho:** Especifica o quanto de espaço em disco deve ser utilizado para o log de comando. Esse espaço é dividido em três segmentos e sempre que um deles estiver cheio, o *snapshot* é executado. O tamanho do log deve ser configurado conforme o tipo de carga de trabalho para que a frequência dos *snapshots* seja adequada.

Após a ocorrência de falhas, a recuperação do cluster inicia-se com a recuperação do *snapshot* mais recente. Depois todas as transações do log são executadas na mesma sequência com que foram armazenadas, esse procedimento é chamado de *replay* do log de comando.

CAPÍTULO 3

REVISÃO BIBLIOGRÁFICA

Há aproximadamente 30 anos a área de *benchmarking* de SGBDs tem atraído a atenção de pesquisadores e do mercado. Com o objetivo de fazer comparações entre sistemas e dar subsídios técnicos para a escolha do melhor SGBD dentro do contexto da sua aplicação, diferentes *benchmarks* foram propostos.

Nessa seção serão apresentados inicialmente os *benchmarks* de desempenho de SGBDs, que são amplamente utilizadas e servem de embasamento desse trabalho. Em seguida serão apresentados alguns *benchmarks* voltados para SGBDs Distribuídos e disponibilidade. Para finalizar será apresentado um comparativo entre os trabalhos mais relevantes.

3.1 Benchmarks de Desempenho

Os *benchmarks* voltados para SGBDs, de forma geral, tem como objetivo simular a carga de trabalho de uma determinada aplicação e gerar índices para comparações posteriores entre diferentes sistemas. Esses *benchmarks* criam estruturas de dados relativas à aplicação que está sendo reproduzida, simulam os clientes que vão acessá-las e geram métricas de desempenho para avaliar o SGBD. Existem *benchmarks* que simulam, por exemplo, um sistema bancário enquanto outros uma loja de comércio eletrônico. Cabe às organizações escolher o *benchmark* mais aderente ao seu contexto de negócio.

3.1.1 DebitCredit

O primeiro *benchmark* encontrado na literatura é o DebitCredit [12], publicado em 1985. Ele foi motivado por uma solicitação de uma grande instituição bancária por um sistema que simulasse o seu negócio e oferecesse medidas que auxiliassem na decisão de compra de equipamentos e sistemas. A arquitetura do DebitCredit reproduz as agências, os ATMs e os clientes do banco realizando operações em um sistema *online*.

O DebitCredit previa apenas três relações: agência, ATMs e clientes. E ele possui uma única transação que contém operações de leitura, atualização e inserção de registro. A métrica utilizada é uma relação entre custo e desempenho (*$/tps*)[21]. O desempenho é a quantidade de transações executadas por segundo (*tps*), desde que 95% delas tenham sido executadas em no máximo 1 segundo. O custo inclui todos os gastos em um período de 5 anos em compra, instalação e manutenção em hardware e software.

A especificação do DebitCredit conta ainda com regras de escala que relacionam o tamanho do banco de dados com o desempenho esperado. Cada *tps* corresponde a 10 tuplas na relação agência, 100 na de ATMs e 100.000 registros na de clientes. Então, por exemplo, se o desempenho esperado fosse de 100 *tps*, o sistema deveria contar com 1.000 agências, 10.000 ATMs e 10.000.000 clientes.

No artigo do DebitCredit não foram publicados os nomes dos SGBDs testados, porém cada fornecedor sabia o seu resultado. A partir dessa publicação iniciou-se uma "guerra de *benchmarks*" entre os grandes fabricantes de SGBDs [17]. Os fornecedores passaram a não autorizar que seus SGBDs passassem por *benchmarks* e apenas o próprio fabricante podia executar e publicar seus resultados.

3.1.2 TPC-A e TPC-B

O desejo da indústria por *benchmarks* padronizados e independentes fez com que os próprios fabricantes de software e hardware fundassem em 1988 o *Transaction Performance Council (TPC)* [9]. Nesse conselho os fabricantes entraram em concenso sobre as regras para execução dos *benchmarks* e divulgação de resultados. Até hoje a TPC é a principal entidade de definição de *benchmarks* e seus resultados são utilizados como referência pela indústria.

O primeiro *benchmark* proposto pelo conselho foi o TPC-A [7], que é uma evolução natural do DebitCredit. Ele simula um ambiente típico de uma aplicação OLTP, onde existe um grande volume de atualizações de registros. As principais modificações com relação ao DebitCredit dizem respeito a um maior controle de integridade das transações e aplicação das propriedades ACID (Atomicidade, Consistência, Isolamento e Durabili-

dade), ser direcionado para arquitetura menores (*e.g.*, cliente/servidor) e relaxamento do tempo de resposta (90% das transações poderiam ser executadas em até 2 segundos).

No ano seguinte ao lançamento do TPC-A foi lançado o TPC-B [8]. Ao contrário do TPC-A, ele foi definido como sendo um teste de estresse e não como um *benchmark* de aplicação OLTP. Apesar do TPC-A e TPC-B utilizarem o mesmo esquema de banco de dados e tipo de transação, seus resultados não podem ser comparados. O TPC-B foi feito para avaliar o desempenho de aplicações em lote e, portanto, não considera questões de interface com o usuário.

O TPC-B é considerado um teste de estresse porque as transações são enviadas na maior frequência que o SGBD é capaz de responder. Em *benchmarks* que simulam aplicações OLTP, é considerado um tempo (*think time*) entre a resposta de uma transação e o envio da próxima, que seria o tempo necessário para o usuário receber o retorno da aplicação e executar a próxima ação. No TPC-B, como o objetivo é simular aplicações em lote, não existe esse tempo, as transações são enviadas de forma consecutiva.

Em 1992 foi lançado o TPC-C, que passou a ser o *benchmark* padrão para sistemas OLTP, e três anos depois tanto o TPC-A como o TPC-B foram descontinuados.

3.1.3 TPC-C

O TPC-C [36] foi o primeiro *benchmark* que não possui relação com o DebitCredit. Comparativamente mais complexo que os anteriores, o TPC-C envolve cinco tipos diferentes de transações concorrentes, um esquema de banco de dados maior e uma estrutura de execução mais complexa que as metodologias anteriores. Esse *benchmark* simula uma aplicação que gerencia os pedidos e estoque de um fornecedor atacadista.

A organização simulada pelo TPC-C é composta por vários armazéns regionais (*Warehouses*), sendo que cada um atende dez distritos de vendas (*District*). Cada distrito tem capacidade para atender 3.000 clientes. Essa hierarquia é apresentada na Figura 3.1. Conforme a organização cresce, novos armazéns e distritos são criados. Cada armazém mantém estoque de 100.000 itens vendidos pela companhia.

Para representar essa organização, o TPC-C define um conjunto de nove tabelas,

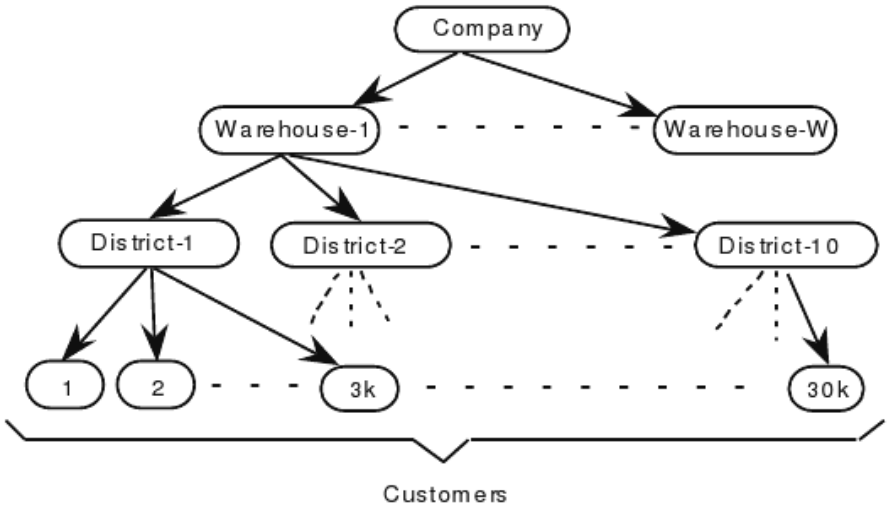


Figura 3.1: Hierarquia das estruturas do TPC-C [36].

apresentadas na Figura 3.2. A cardinalidade de cada tabela segue uma razão com relação a tabela WAREHOUSE. A exceção é a tabela ITEM que sempre possui 100.000 registros. A especificação do TPC-C define também como os valores de cada atributo devem ser carregados e o conjunto mínimo de dados necessários antes da execução do *benchmark*.

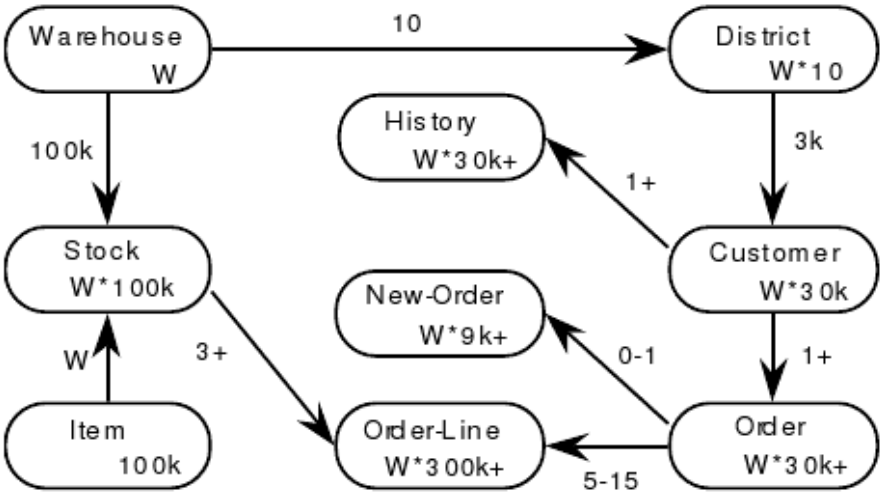


Figura 3.2: Diagrama entidade-relacionamento do TPC-C [36].

No TPC-C são definidos cinco tipos de transações: *New-Order* representa a entrada de novos pedidos no sistema, *Payment* atualiza o crédito do cliente, *Order-Status* verifica o estado do último pedido feito por um cliente, *Delivery* faz o processamento em lote de dez novos pedidos e a *Stock-Level* que fornece os itens que possuem quantidade abaixo do limite de estoque. Cada transação envolve diversas operações de seleção, atualização

e inserção.

Para simular os usuários da aplicação, o TPC-C utiliza o conceito de Terminais Remotos Emulados (RTEs). Cada RTE está conectado a um Armazém e é através dele que as transações são enviadas para o SGBD. Cada armazém pode aceitar conexões de no máximo 10 RTEs.

As operações executadas pelos RTEs incluem seleção e envio de transação, espera de resposta, exibição do resultado e espera pelo tempo de ação do usuário (*think time*). A cada ciclo de execução, o RTE deve escolher de forma equilibrada e aleatório um dos cinco tipos de transações do TPC-C. A especificação define um percentual mínimo do quanto cada tipo de transação deve ser executada.

A principal métrica definida pelo TPC-C é a tpmC. Ela é calculada a partir do total de transações *New-Order* executadas com sucesso dividido pelo tempo de execução do *benchmark*.

3.2 Benchmarks de Sistema Distribuídos

A grande maioria dos *benchmarks* de desempenho, como os TPCs, podem ser utilizados com poucas ou nenhuma modificação para avaliar SGBDs Distribuídos. Isso acontece porque do ponto de vista da aplicação o SGBD pode ser abstraído como uma única unidade lógica, ou seja, como uma caixa preta. Seja o SGBD distribuído ou não, a sua funcionalidade final é a mesma, processar transações. Porém os SGBDDs apresentam algumas características, relacionadas a sistemas distribuídos, que devem ser levadas em consideração, como escalabilidade e disponibilidade. Para avaliar essas características surgiram na literatura alguns *benchmarks* voltados especificamente para SGBDs Distribuídos.

Orji [31] em seu pioneiro trabalho apresenta uma metodologia de *benchmarking* para SGBDs Distribuídos. Algumas características levadas em consideração na avaliação de bancos de dados distribuídos são: (1) configuração da rede, (2) número de nós, e (3) distribuição dos dados entre os nós. Com base nestas três variáveis, o autor define um conjunto de oito consultas para avaliar os SGBDs distribuídos. De acordo com o tempo de resposta do sistema para cada uma das consultas, um índice de desempenho é gerado,

e posteriormente usado para avaliar diferentes sistemas. Neste trabalho [31], o autor não leva em consideração a disponibilidade do sistema como um fator crítico.

Nos últimos anos surgiram no mercado uma grande quantidade de sistemas distribuídos para armazenamento de dados. Com isso tem ficado cada vez mais difícil para os desenvolvedores escolherem qual é o sistema que melhor se adapta a uma determinada aplicação. Com essa motivação, em 2010 os pesquisadores do *Yahoo!* Cooper *et al.* [16] apresentaram o *Yahoo! Cloud Serving Benchmark* (YCSB) para facilitar a avaliação de SGBDs Distribuídos.

O YCSB é uma plataforma totalmente modular e extensível que conta com um gerador de dados, um gerador de carga de trabalho que também monitora o desempenho e conectores para diversos SGBDDs populares. Ele foi desenvolvido para ser utilizado tanto com SGBDs relacionais como de armazenamento chave-valor. Além disso ele foi implementado em código aberto [11] e com interfaces bem definidas, sendo relativamente simples a sua extensão (e.g., criação de conectores para outros SGBDs, desenvolvimento de outros tipos de cargas de trabalho).

Esse *benchmark* foi proposto com quatro camadas: (1) Desempenho, (2) Escalabilidade, (3) Disponibilidade, e (4) Replicação. As duas primeiras camadas foram implementadas pelos autores do trabalho, enquanto as duas últimas são descritas apenas como trabalho futuro.

A camada de Desempenho tem como objetivo caracterizar a relação entre desempenho em transações por segundo e latência, ou seja, o comportamento ao aumentar a carga de trabalho mantendo fixo o hardware utilizado. Já a camada de Escalabilidade utiliza duas métricas. A primeira, chamada de *scaleup*, avalia o desempenho em transações por segundo quando aumenta-se proporcionalmente o hardware e a carga de trabalho do sistema. A segunda métrica, chamada de *Elastic speedup*, avalia o comportamento do sistema quando o número de máquinas no cluster aumenta com o sistema em execução.

O YCSB possui um gerador de carga de trabalho sintética baseado nas operações *CRUD* (*create, read, update e delete*). O usuário pode definir a distribuição dessas operações assim como a quantidade de registros, tamanho dos dados e outras carac-

terísticas para o seu *benchmark*. Os resultados experimentais demonstraram que a plataforma pode ser utilizada para avaliações de desempenho, possibilitado a comparação de diferentes SGBDs distribuídos. Esse *benchmark* tem sido amplamente utilizado pela comunidade acadêmica e indústria [35, 6, 5].

3.3 Benchmarks de Disponibilidade

3.3.1 Evolução da Área

Ao longo dos anos 80 e 90 as técnicas de *benchmarking* deram maior ênfase para avaliação de desempenho. A própria definição de *benchmark* utilizada por diversos autores já estava diretamente relacionada a desempenho. Em [21], por exemplo, *benchmarking* é definido como o processo de comparação de desempenho de dois ou mais sistemas através de métricas. Entretanto ao longo da década de 90, com o avanço dos sistemas computacionais, a disponibilidade passou a ter maior relevância [19].

Em 1980 um bom sistema computacional tipicamente oferecia 99% de disponibilidade[38]. Apesar de parecer uma boa taxa, isso significava que a aplicação ficava em média 100 minutos fora do ar por semana, que é um tempo muito grande para aplicações críticas. Já nos anos 90, a disponibilidade típica alcançava 99,99%[18], que já representa menos de 50 minutos de falha por ano. Com isso, somado à outros fatores inerentes ao avanço da tecnologia como aumento da capacidade computacional e queda dos preços, tornou-se viável o desenvolvimento de sistemas computacionais de alta disponibilidade para serem utilizados em aplicações críticas e *online*[19].

Nos anos 80, apesar de ainda não se falar em *benchmark* de disponibilidade, os conceitos relacionados a tolerância a falha e dependabilidade já estavam sendo discutidos, o que levou a formação de dois grupos acadêmicos que contribuiram muito para o avanço da terminologia e entendimento do tema, o IFIP Working Group 10.4[4] e IEEE Technical Comittee on Fault-Tolerant Computing [3]. Os resultados iniciais das discussões desses grupos são apresentados em [25] onde são apresentados os atributos que constituem a dependabilidade e a confiabilidade.

Nos início dos anos 90 a comunidade acadêmica passou a publicar mais trabalhos relacionados a avaliação da disponibilidade. De forma geral, a maioria desses trabalhos discutiam a crescente importância da disponibilidade e definiam conceitos relacionados à ela[19], além de apresentar técnicas para avaliar tipos específicos de aplicações. Muitos desses trabalhos ainda tinham seu foco na avaliação de sistemas operacionais, levando em consideração falhas em diferentes camadas de hardware [30]. Nesse época ainda se discutia pouco sobre *benchmark* de disponibilidade e menos ainda sobre a disponibilidade específica de SGBDs.

Um dos primeiros trabalhos em que é feito a injeção de falhas para avaliação da confiabilidade de um sistema é o [30]. Os autores desse trabalho testaram aproximadamente 90 utilitários do sistema operacional UNIX com uma técnica simples, gerar sequências de caracteres aleatórias para serem usadas como dados de entrada dos utilitários. O objetivo era verificar se a aplicação apresentava erro ou travamento, e isso aconteceu em mais de 24% das aplicações.

Em [19], Gray e Siewiorek apresentaram conceitos chaves e técnicas para desenvolver sistemas computacionais de alta disponibilidade. Nesse trabalho eles também discutiram a importância da disponibilidade e sua tendência nos sitemas computacionais. Segundo eles, um sistema de alta disponibilidade está relacionado aos seguintes conceitos e técnicas: (1) Modularidade, (2) Módulos *failt-fast* (deve operar corretamente ou falhar imediatamente), (3) Modos independentes de falhas (as falhas de um módulo não devem afetar os outros), (4) Redundância e (5) Reparo. Nesse trabalho, utilizando conceitos anteriores apresentados em [25], os autores quantificam a disponibilidade como:

$$Disponibilidade = \frac{MTTF}{MTTF + MTTR}$$

Onde $MTTF$ é a média de tempo entre falhas (do inglês, *mean-time-to-failure*) e $MTTR$ é a media de tempo para reparo (do inglês, *mean-time-to-repair*).

A partir dos anos 2000 a área de disponibilidade atraiu a atenção de mais pesquisadores e diversos trabalhos sobre disponibilidade ou assuntos relacionados (dependabilidade, tolerância a falha, robustez, entre outros) foram apresentados [23]. Nas próximas seções

serão detalhadas as publicações de maior repercussão e relevância para este trabalho.

3.3.2 Frameworks para análise de disponibilidade

Pesquisadores da empresa Sun Microsystem apresentaram uma plataforma de alto nível para teste de disponibilidade [43] voltada para a indústria. Nesse trabalho é proposto um *benchmark* hierárquico, chamado *R-cubed*, para avaliar a disponibilidade de sistemas computacionais em geral. O foco desse trabalho não é especificamente SGBDs.

A plataforma proposta [43] tem como objetivo final responder a pergunta "Qual o percentual de tempo que o sistema está funcional durante um determinado período?". Para isso ela leva em consideração os três fatores principais que afetam a disponibilidade e apresenta métodos de avaliá-los, são eles:

1. Taxa de falhas/manutenções: É o número de falhas e manutenções que ocorrem no sistema em um determinado período de tempo.

2. Robustez: Capacidade do sistema em detectar e corrigir falhas sem impactar na funcionalidade do sistema. Além de falhas, a robustez também diz respeito a manutenção e eventos externos, como erros humanos e fatores ambientais.

3. Recuperação de falhas: É medida através da velocidade na qual o sistema volta a ser operacional após a ocorrência de alguma falha.

A abordagem hierárquica da plataforma *R-cubed* é apresentada na Figura 3.3. No primeiro nível a disponibilidade é medida através do tempo em que o sistema não está disponível durante um intervalo (por exemplo, horas de indisponibilidade durante o ano). Nos próximos níveis a disponibilidade é segmentada em fatores que a afetam e em métricas para avaliá-los.

A taxa de falhas/manutenções visa medir a frequência com que ocorrem eventos que podem gerar uma parada no sistema. A métrica definida para esse fator é a simples soma da taxa de falhas com a taxa de manutenção.

Para mensurar a Robustez, a plataforma classifica as falhas e manutenções em diferentes categorias conforme o seu impacto no sistema. Os eventos que geram maior impacto,

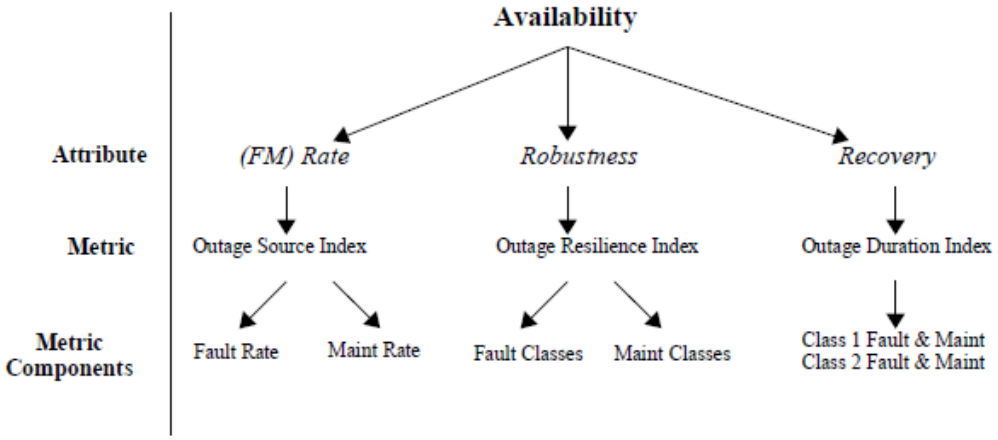


Figura 3.3: Plataforma *R-cubed* para *benchmarking* de disponibilidade [43].

por exemplo, fazendo com o que sistema fique indisponível, recebem um peso maior do que as transparentes, aquelas em que o usuário final não é afetado. A partir da taxa de falhas/manutenções para cada uma dessas categorias é gerada uma métrica que indica a robustez geral do sistema.

A recuperação de falhas é medida através da média do tempo em que o sistema fica indisponível em cada evento de falha. Ela é calculada dividindo o tempo total de indisponibilidade pela quantidade de eventos durante certo período.

Um dos pontos interessantes da plataforma *R-cubed* é o fato dela ser genérica. Seus fatores podem ser mensurados e combinados de diferentes formas para atender os mais diversos tipos de sistemas.

Partindo dos conceitos apresentados em [43], os mesmos pesquisadores da Sun Microsystems [29] propuseram uma suite de *benchmarks* de recuperação de sistema, a *System Recovery Benchmark* (SRB). O foco dos *benchmarks* que compõem essa suite é avaliar o tempo de recuperação do sistema após uma situação de falha. O SRB define alguns requisitos que devem ser atendidos, como, por exemplo, que a recuperação deve ser feita sem a intervenção humana e que o sistema só é considerado recuperado quando está em um estado funcional e usável[28, 23].

Cada *benchmark* que compõe essa suite avalia uma camada diferente de um sistema computacional típico. As camadas e seus respectivos *benchmarks* são apresentados na tabela 3.1. Desses apenas o SRB-A e o SRB-X foram efetivamente desenvolvidos.

Tabela 3.1: *Camadas do System Recovery Benchmark (SRB) [43]*

Camada	**Benchmark**
Cluster	SRB-X
Aplicação	SRB-D
Banco de Dados	SRB-C
Controlador de Disco (*RAID*)	SRB-B
Sistema Operacional	SRB-A
Hardware	SRB-A

O SRB-X[34] é um *benchmark* para avaliar o tempo de recuperação de sistemas baseados em cluster. A métrica utilizada no SRB-X considera o tempo de recuperação do sistema após uma falha ou reinicialização intencial para reconfiguração. O tempo de recuperação é obtido em duas situações diferentes, quando o sistema está sem nenhuma carga ($t_{no-load}$) e quando está submetido a uma carga sintética ($t_{syn-load}$). A métrica é calculada através da seguinte equação:

$$SRB - X = \frac{3600}{t_{no-load} + tsun - load}$$

Além do SRB-A e SRB-X, os pesquisadores da Sun Microsystems [29] também desenvolveram o *Maintenance Robustness Benchmark-A* (MRB-A) [44] para avaliar a robustez de sistemas operacionais e hardware. Esse *benchmark* considera aspectos específicos de eventos de manutenção como, por exemplo, a troca de componentes de hardware e a instação de atualizações.

3.3.3 Benchmarks de disponibilidade para SGBD

Em [39] foi proposto o DBench-OLTP, um *benchmark* de dependabilidade pra sistemas OLTP que utiliza como carga de trabalho o TPC-C. Nele são especificadas métricas e passos para avaliar os principais aspectos da dependabilidade, com ênfase na disponibilidade que é considerada pelos autores como uma das características mais importantes de banco de dados e sistemas OLTP.

A proposta do DBench-OLTP é baseada em métricas e seu principal diferencial das metodologias anteriores são as métricas relacionadas a dependabilidade e a utilização do

conceito de "carga de falha" (do inglês *faultload*), que é um conjunto de falhas inseridas no sistema para simular problemas reais. Assim como existe a carga de trabalho, o DBench-OLTP utiliza uma carga de falhas.

Uma visão geral da execução do *benchmark* é apresentada na Figura 3.4. Inicialmente ele é dividido em duas fases. A Fase 1 é a primeira execução completa do workload sem nenhuma falha para gerar uma base de comparação de desempenho. Na fase 2 a carga de trabalho é executada com a ocorrência de falhas. A fase 2 é dividida em intervalos que recebem o nome de *injection slots*. Cada *injection slot* representa um intervalo de tempo onde a carga de trabalho é executada e uma falha é inserida. As falhas são inseridas apenas quando o sistema já está no seu estado estável. Depois que a falha é ativada, faz parte do *injection slot* o tempo para ela ser detectada (de forma automática ou manual) e recuperada. No final a carga de trabalho ainda é executada por um tempo para validar o desempenho após a recuperação.

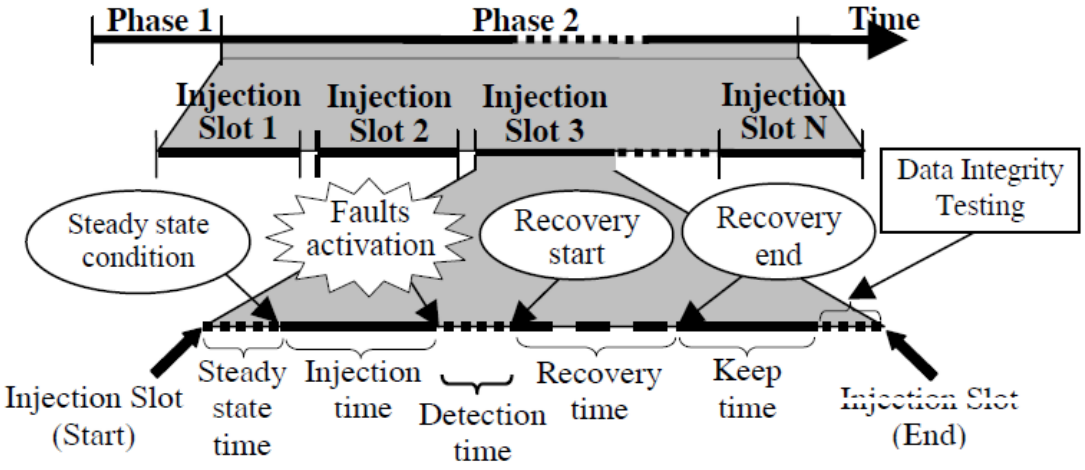


Figura 3.4: Método de execução do DBench-OLTP [39].

Além das métricas utilizadas pelo TPC-C [36], o DBench-OLTP apresenta métricas de desempenho na presença da carga de falhas e métricas de disponibilidade. As métricas de desempenho são análogas às do TPC-C: Número de transações executadas por minuto e o preço por transação durante a fase 2. As métricas de disponibilidade são três: (1) O número de erros detectados por testes de consistência e metadados para avaliar o impacto das falhas na integridade dos dados, (2) disponibilidade para o SGBD, que considera o tempo em que o sistema é capaz de responder pelo menos um terminal de emulação de

transações e (3) disponibilidade para o usuário final, que mensura o tempo em que o sistema é capaz de responder as transações submetidas por cada terminal.

Em [40], publicado em 2009, os autores discutem o cenário do *benchmark* de dependabilidade. A maioria dos sistemas atualmente tem como requisitos fundamentais a garantia de alta disponibilidade e confiabilidade. Com isso é mandatório que os *benchmarks* mudem seu foco e considerem em suas métricas os atributos de dependabilidade. Apesar de diversos trabalhos sobre *benchmark* de dependabilidade terem sido conduzidos no passado, nenhum se tornou padrão na indústria até o momento, endossado pelos grandes fabricantes e comunidade acadêmica.

Uma forma de criar um *benchmark* de dependabilidade padrão na indústria seria acoplá-lo a um *benchmark* de desempenho de ampla aceitação. E essa foi a proposta apresentada em [13]. Nessa trabalho os autores defendem que métricas de dependabilidade deveriam fazer parte dos *benchmarks* do TPC[9]. As métricas e conceitos citados são baseados no DBench-OLTP [39].

Duas formas de adicionar métricas de dependabilidade nos *benchmarks* TPC são possíveis [13]. Uma delas é extender a especificação de cada *benchmark* TPC, especificamente a parte dos testes ACID, acoplando as métricas de dependabilidade e a carga de falhas de forma semelhante as do DBench-OLTP [39]. A outra forma é criar uma abordagem única com uma série de critérios e testes que poderiam ser acoplado em todos os *benchmarks* TPCs. Essa segunda abordagem apresenta mais desafios devido a sua generalização entretanto apresenta vantagens do ponto de vista de custo já que uma única implementação poderia ser utilizada múltiplas vezes, como foi feito anteriormente pelo *TPC-Energy* [33].

Entre os *benchmarks* TPCs, apenas o TPC-E [37] apresenta algum tipo de medição de tempo de recuperação. Entretanto essa métrica, chamada de "tempo de recuperação de negócio" (do inglês *business recovery time*) tem um escopo muito limitado. Uma abordagem para extender o TPC-E para avaliar a disponibilidade de sistemas de banco de dados é apresentada em [26]. Essa metodologia foi desenvolvida e utilizada no desenvolvimento do SGBD *Microsoft SQL Server*. O foco desse trabalho é avaliar os mecanismos que ga-

rantem a alta disponibilidade (HA, do inglês, *High Availability*) em SGBDs. O cenário de alta disponibilidade considerado é baseado em dois servidores com instâncias do banco de dados. Um é o servidor principal, que processa todas as requisições dos clientes, e o outro é o servidor de *Standby*, que é utilizado para prover o serviço quando o servidor principal falha.

A metodologia para execução do *benchmark* consiste em executar por determinado período o TPC-E para gerar as métricas de desempenho para linha de base e depois executar a mesma carga de trabalho inserindo falha. Para avaliar a disponibilidade, duas métricas principais foram propostas: (1) Perda de desempenho na utilização de sistemas HA contra sistemas não-HA e (2) quão rápido o SGBD pode recuperar o serviço após a ocorrência de uma falha.

3.4 Comparativo

Nas seções anteriores foram apresentados diversos estudos sobre disponibilidade e *benchmarks*. Através desse levantamento podemos observar que a área de *benchmark* de disponibilidade para SGBDs ainda é relativamente nova. Enquanto os *benchmarks* de desempenho já existem desde 1985 [12] e são muito bem aceitos e suportados pela indústria [9], o primeiro *benchmark* de disponibilidade de maior repercussão surgiu em 2001 [39, 40]. E até o momento não existe um *benchmark* padrão na indústria para caracterização dos aspectos da disponibilidade [26].

A disponibilidade de sistemas computacionais está relacionada diretamente à probabilidade da ocorrência de falhas, sejam elas causadas por fatores internos do sistema (*e.g.software* e *hardware*) ou externos (*e.g.*ambiente e falhas humanas) [39].

O cálculo da disponibilidade de um sistema [25] pode ser feito levando em consideração o tempo médio entre falhas e o tempo médio de reparo. Porém essa métrica expressa a disponibilidade considerando dados do passado, e não a capacidade de disponibilidade dele para o futuro. Além disso a simplicidade dessa abordagem tem como efeito colateral diversas limitações: (1) não é possível avaliar o ambiente sem ter um histórico de utilização, (2) os diversos fatores envolvidos fazem com que não seja confiável prever

o futuro baseando-se no passado, (3) a métrica não considera aspectos importantes de sistemas específicos, como os SGBDDs. Devido a essas limitações faz-se necessário a utilização de métodos mais completos para avaliar a disponibilidade. E diversos estudos apresentados nessa revisão abordaram o problema.

Os pesquisadores da Sun Microsystems contribuiram muito para a área de *benchmark* de disponibilidade. Em [43] foram discutidos os fatores que afetam a disponibilidade e métricas possíveis. Porém essa é uma plataforma genérica, com idéias relevantes para sistemas computacionais em geral, não considera os aspectos específicos de um SGBD ou sistema distribuído.

Uma suite de *benchmarks* específicos, a *System Recovery Benchmark* (SRB), foi proposta pela Sun Microsystems. Essa suite inclui um *benchmark* voltado para SGBDs (SRB-C), porém ele não chegou a ser implementado.

O DBench-OLTP é um *benchmark* de dependabilidade voltado para SGBDs que considera os aspectos de disponibilidade. Através de um método que utiliza a carga do trabalho do TPC-C e uma carga de falhas, ele avalia os impactos de diversos tipos de falhas no SGBD. Esse foi o primeiro *benchmark* de dependabilidade de SGBDs que teve grande repercussão no meio acadêmico.

O DBench-OLPT descreve vários tipos de falhas e avalia o impacto delas em um SGBD centralizado. O método considera, por exemplo, o tempo que cada falha leva pra ser detectada e o tempo de recuperação necessário.

Outro trabalho recente de grande importância acadêmica foi o [26]. Utilizando como carga de trabalho o TPC-E, foi proposto um *benchmark* para avaliar os mecanismos de disponibilidade de SGBDs. Essa abordagem é voltada para SGBDs centralizados que utilizam mecanismos de alta disponibilidade.

Portanto, concluímos que nenhum dos trabalhos anteriores são voltados para avaliar a disponibilidade de SGBDs Distribuídos, levando em consideração seus mecanismos para prover e manter a disponibilidade.

CAPÍTULO 4

UNDER PRESSURE *BENCHMARK*

O Under Pressure *benchmark* (UPB) é um método baseado em métricas para avaliar a disponibilidade de SGBDDs. Dentre as características relacionadas a disponibilidade, discutidas na Seção 2.3, o foco do UPB é voltado para a tolerância e recuperação de falhas.

Os SGBDDs possuem diversos mecanismos utilizados para prover a sua disponibilidade. Através deles é possível configurar um SGBDD para tolerar um determinado nível de falhas mantendo o BDD totalmente operacional e acessível. Porém o uso desses mecanismos tem um custo em termos de desempenho e aproveitamento da capacidade de hardware. A tendência é que quanto maior o nível de proteção oferecido, maior será o impacto no desempenho. Avaliar a eficiência e os impactos da utilização de recursos que provem a disponibilidade dos SGBDDs são os objetivos do UPB.

O UPB avalia o comportamento do SGBDD com diferentes configurações e submetido a diversas situações, incluindo aquelas críticas para sua operação onde os mecanismos de disponibilidade são exercitados, como durante a ocorrência de falhas e na recuperação delas. A partir dessas avaliações são geradas métricas parciais e finais. As métricas parciais avaliam o SGBDD com uma configuração ou condição específica. Já as métricas finais, que sumarizam as parciais, representam a disponibilidade do SGBDD de forma geral. As métricas finais de um SGBDD podem ser classificadas e comparadas com outros SGBDDs e ambientes.

Dentre as diversas características do UPB é possível destacar as seguintes:

- Portabilidade: O UPB é independente de tecnologias específicas, podendo ser utilizado com SGBDDs de diferentes fabricantes.

- Independente de carga de trabalho: O método proposto pode ser utilizado com diferentes cargas de trabalho, o que permite maior abrangência e utilização do *ben-*

chmark.

- Métricas parciais e finais: As diferentes métricas geradas pelo UPB podem ser utilizadas para diferentes propósitos. As métricas parciais podem ser utilizadas para comparar diferentes configurações do SGBDD ou para avaliar o comportamento do SGBDD em uma aplicação específica. As métricas finais podem ser utilizadas para comparar SGBDDs de diferentes fabricantes ou sendo executados em diferentes ambientes computacionais.

- Expansível: O método proposto pode ser facilmente expandido ou adaptado para considerar outros fatores dos SGBDDs ou incorporar outras métricas.

Os diversos aspectos do UPB serão detalhados nas seções seguintes. A arquitetura de um ambiente típico para a execução do *benchmark* é descrita na Seção 4.1. Em seguida a carga de trabalho e características relacionadas serão apresentadas na Seção 4.2. Na seção 4.3 é descrito o método de execução do UPB com a explicação de cada um dos seus passos e métricas parciais. A partir das métricas parciais são geradas as métricas finais, apresentadas na Seção 4.4. Por fim é feito uma discussão sobre o benchmark na Seção 4.5.

4.1 Ambiente de Execução do UPB

A arquitetura do ambiente utilizado para a execução do UPB é semelhante a um ambiente típico de uma aplicação que utiliza um SGBDD. A arquitetura é composta por máquinas clientes que acessam o BDD submetendo transações e por um conjunto de máquinas onde o SGBDD é executado, denominado *cluster*. Cada máquina desse *cluster* recebe o nome de nó. Todas as máquinas estão inter-conectadas através de uma rede de computadores.

A quantidade de máquinas que compõem o *cluster* e a topologia de rede utilizada varia conforme o ambiente e o SGBDD que está sendo avaliado. Tipicamente a maioria dos *clusters* seguem a topologia física de estrela, tendo seus nós conectados através de comutadores de rede (do inglês, *switches*).

4.1.1 Clientes

A forma como as máquinas clientes se conectam ao *cluster* depende do SGBDD. Em geral, os clientes podem se conectar e submeter transações para qualquer um dos nós do cluster ou até mesmo a todos os nós ao mesmo tempo. Nesse caso o conector do cliente com o SGBDD (também chamado de *driver*) é responsável por balancear o número de transações enviadas para cada nó.

Independente da capacidade de processamento de transações do *cluster*, cada cliente individualmente tem um limite de carga (em transações por segundo) que é capaz de submeter para o SGBDD. São vários os fatores que podem gerar essas limitações, sejam contenções de hardware (*e.g.*, capacidade de rede e processamento) ou contenções de software (*e.g.*, limitações do número de *threads* e do número de transações por conexão). Essa limitação do cliente implica que podem ser necessárias várias máquinas clientes para atingir o limite de carga suportado pelo *cluster*.

Do ponto de vista de uma aplicação OLTP, normalmente a limitação do cliente não é um fator importante já que são esperados centenas ou milhares de clientes distribuídos se conectando ao *cluster*. Já em um *benchmark*, para viabilizar a avaliação do SGBDD em um ambiente menor e controlado, temos que simular esse mesmo volume de carga com o menor número de clientes possível.

Em cada etapa do *benchmark* busca-se atingir o desempenho máximo do SGBDD. Para isso a quantidade de clientes necessários varia conforme a etapa. Mas sempre é importante que o limite de carga dos clientes seja definido e respeitado para evitar falsas medições. Quando um cliente é configurado para submeter mais transações do que seu limite máximo ocorrem variações no desempenho que podem ser confundidas com a saturação do *cluster* (*e.g.*, aumento da latência das transações, SGBDD recusar novas conexões).

4.1.2 Índice de Tolerância a Falhas

Os SGBDDs possuem diversos recursos para aumentar a sua disponibilidade (conforme Seção 2.3). Um dos objetivos do UPB é avaliar os mecanismos que garantem a continuidade do *cluster* operacional mesmo com a ocorrência de falhas em alguns nós. Esse tipo

de proteção é tipicamente implementado nos SGBDDs através da replicação dos dados entre seus nós.

Cada fabricante de SGBDD utiliza diferentes estratégias para a replicação, com diferentes nomenclaturas e configurações. Mas, independente de nomenclaturas, em todos os SGBDDs é possível configurar um índice de tolerância à falhas que indica quantos nós do *cluster* podem ficar indisponíveis sem que nenhum dado seja perdido ou o serviço seja interrompido. Nesse trabalho esse índice será chamado de K. Essa nomenclatura foi adotada por ser a utilizada pelo VoltDB [10], cujo mecanismo de replicação é baseado no conceito de *k-safety*.

Quando K for igual a zero ($K = 0$) significa que nenhum dado do BDD é replicado, ou seja, todos os registros existem em apenas um nó do *cluster*. Nesse caso, se um nó for perdido o serviço será interrompido e os dados armazenados nele serão perdidos. Já configurando K como um ($K = 1$), todos os dados terão uma réplica, ou seja, deverão ser armazenados em pelo menos dois nós. Isso significa que quando um dos nós é perdido, nenhum registro é perdido e o serviço não é interrompido. A mesma lógica se aplica para valores maiores de K.

O índice de tolerância a falhas é configurado conforme o ambiente e os requisitos de disponibilidade da aplicação que está sendo utilizada. Conforme conhecimento de especialistas, na maioria dos ambientes de produção de aplicações OLTP esse índice varia entre 1 ($K = 1$) e o valor que representa a metade do número de nós do *cluster* (*e.g.*se o ambiente possuir 6 nós, então $K = 3$). Portanto, a UPB considera a faixa de valores de K variando de 1 à 50% do número total de nós do *cluster*.

4.1.3 Falhas Utilizadas

Em [19], Gray e Siewiorek apresentaram conceitos que devem ser seguidos por sistemas tolerantes a falhas. Entre eles está o de falha rápida (*fail-fast*) que prevê que cada módulo do sistema deve operar corretamente ou parar imediatamente.

Considerando que o conceito de falha rápida seja seguido por SGBDDs que fornecem algum nível de tolerância a falhas, qualquer tipo de falha em um nó do *cluster*, seja de

software ou hardware, faz com que o processo do SGBDD seja interrompido imediatamente e o nó fique indisponível para o *cluster*.

Partindo do princípio de falha rápida [19], o objetivo do UPB não é avaliar o impacto de diferentes tipos de falhas. Assume-se que qualquer que seja a falha em um nó o impacto causado por ela é o mesmo: o nó fica imediatamente indisponível. Portanto, as falhas utilizadas no UPB consistem na interrupção abrupta do processo.

4.2 Carga de Trabalho

O UPB não define uma carga de trabalho específica. O *benchmark* pode ser utilizado com diferentes cargas de trabalho conforme o interesse do que deve ser avaliado. O único requisito é que a carga de trabalho seja baseada no conceito de transação. E essas transações podem ser de qualquer tipo, desde operações simples de banco de dados (*e.g.*, escrita, leitura, atualização, exclusão) até transações de negócios mais complexas compostas de várias operações, como as utilizadas no TPC-C [36] e TPC-E [37]. As medições de desempenho que compõem as métricas do UPB são baseadas em transações por segundo (tps), podendo ser aplicadas para qualquer tipo de transação.

Apesar da independência do UPB com a carga de trabalho, nesse trabalho é proposto a utilização do *YCSB*[16, 11] como gerador de carga. O YCSB, que já foi apresentado na Seção 3.2, é uma plataforma para execução de *benchmarks* em SGBDs Distribuídos. Ele contém um gerador de dados para fazer a carga inicial do BD, um gerador de carga de trabalho configurável e conectores para diversos SGBDDs populares do mercado.

Junto com o YCSB é distribuído o *YCSB Core Package*[16] que é um pacote com uma série de tipos de cargas de trabalho criadas a partir da observação do comportamento de aplicações de *internet*. O objetivo desse pacote não é simular uma aplicação específica completa, como acontece nos TPCs, mas sim executar operações básicas de banco de dados, como leitura e inserção.

4.2.1 Nível da Carga de Trabalho

O aumento da carga de trabalho pode gerar diferentes efeitos no BD que variam conforme o ambiente e a configuração do SGBDD. Em alguns casos o desempenho pode ser simplesmente limitado e o SGBDD pode se recusar a atender mais transações. Em outras, o *cluster* pode ser capaz de atender um volume maior de transações porém comprometendo sua latência, ou seja, aumentando o tempo que o *cluster* leva para processar cada transação individualmente.

A carga do *benchmark* deve ser a máxima possível que simule a utilização do sistema por aplicações reais. Para isso o seguinte requisito, representado por ϕ, deve ser atendido pela carga de trabalho:

- ϕ: A latência de 90% das transações deve ser inferior à 1 segundo.

Esse tipo de requisito é amplamente utilizado em *benchmarks* [36, 37] para garantir que o tempo de resposta do SGBDD seja compatível com grande parte das aplicações reais. A latência das transações do UPB deve ser medida como o tempo entre um cliente submeter uma transação e receber seu resultado ou retorno (*i.e.*, sucesso ou erro) do SGBDD.

4.3 Metodologia

O UPB realiza a avaliação do SGBDD em quatro situações distintas onde os mecanismos que provem a disponibilidade do banco de dados são exercitados de diferentes formas. Cada uma dessas situações são chamadas de cenários.

Os cenários do UPB foram definidos a partir da combinação e avaliação das principais variáveis relevantes para a disponibilidade de SGBDDs. O método utilizado para a definição dos cenários e as variáveis consideradas são descritos a seguir.

4.3.1 Variáveis e Definição de Cenários

Para desenvolver o UPB foi necessário definir todos os cenários importantes para um *benchmark* de disponibilidade. Esses cenários devem considerar todos os mecanismos e

variáveis relevantes em um SGBDD para a gestão da disponibilidade. Para isso foram utilizados alguns conceitos da técnica de *pairwise testing* [24].

Essa técnica consiste em um método combinatório para criação de casos de teste. Em um teste exaustivo o objetivo é que a partir de um conjunto de variáveis envolvidas no sistema todas as possíveis combinações discretas delas sejam testadas. Porém em ambientes complexos, com muitas variáveis, testar todas as combinações de variáveis pode resultar em um número muito grande de cenários a serem testados, inviabilizando o teste. Para evitar isso o *pairwise testing* propõem uma forma de diminuir o número de cenários de testes, paralelizando o teste de múltiplas variáveis.

A lógica por trás do *pairwise testing* é que as situações mais relevantes em um teste, por exemplo a ocorrência de falhas dependem de uma única variável ou da combinação de duas delas. Essas situações são testadas pelo *pairwise testing*. Já as situações que dependem da combinação de três ou mais variáveis são menos frequentes e precisam de técnicas mais caras para serem completamente testadas.

O *pairwise testing* foi desenvolvido inicialmente para a criação de testes de software (*i.e.*, busca por falhas). Porém o seu conceito pode ser adaptado para o desenvolvimento de um *benchmark*. Nesse caso ao invés de procurar por falhas procuramos por situações que gerem perdas significativas de desempenho do SGBDD.

Alguns conceitos do *pairwise testing* foram utilizados no desenvolvimento do UPB. As variáveis consideradas foram as seguintes:

1. Índice de tolerância a falhas (K): Indica a quantidade de nós com falhas que o *cluster* é capaz de tolerar mantendo o banco de dados *online* e respondendo as requisições dos clientes.

   Os valores possíveis para essa variável variam conforme o ambiente que está sendo testado, sendo os seguintes:

   - $K = 0$ (sem tolerância): O serviço é interrompido com a ocorrência de uma falha.
   - $K = 1, 2, ..., \frac{N}{2}$, onde N representa o número de nós do *cluster*. Portanto,

o *cluster* é capaz de tolerar a ocorrência de falhas em K nós. Os valores avaliados variam conforme o ambiente, variando de 1 até 50% do número de nós do *cluster*.

2. Número de nós com falhas (F): Representa a quantidade de nós que apresentam falhas e ficam indisponíveis para o *cluster*.

   Os valores possíveis e relevantes para serem testados são:

   - $F = 0$: O *cluster* está completo, sem falhas.
   - $F = 1, 2, ..., K$: O *cluster* está incompleto, sendo que F nós estão indisponíveis. Devem ser testados todas as quantidades de falhas possíveis até a K-ésima.

3. Reinserção de nós no *cluster*: Após a ocorrência de falhas, os nós que sairam podem ser reestabelecidos para que sejam reinseridos no *cluster*. O processo de reinserção pode levar certo tempo já que é necessário copiar dados para ele e atualizar os controles do *cluster*. Na UPB são considerados apenas dois valores para essa variável:

   - Sim: Significa que um nó está sendo reinserido.
   - Não: Nenhum nó está sendo reinserido.

   Em um ambiente real o processo de reinserção de nós consiste normalmente na solução dos problemas individual deles que o fizeram sair do *cluster*. Por esse motivo o processo é feito normalmente de forma manual ou com o acompanhamento de técnicos competentes. Por esse motivo na UPB essa variável assume apenas dois valores e considera a reinserção de um nó de cada vez.

Das três variáveis consideradas, o domínio de duas delas dependem do tamanho do ambiente que está sendo avaliado. É fácil observar que a simples combinação dos valores possíveis para estas três variáveis geraria um número grande de cenários. Porém muitos desses cenários são irrelevantes, não factíveis ou as variáveis envolvidas podem ser avaliadas individualmente, conforme o *pairwise testing*.

Na Tabela 4.1 é apresentada a combinação de todas essas variáveis, juntamente com um indicador se o cenário é relevante para o *benchmark* e considerações adicionais. Como o domínio de algumas variáveis depende do ambiente, foi necessário o uso de algumas generalizações. Nessa tabela o X representa a configuração do índice de tolerância a falhas do ambiente (K) tal que $1 <= X <= \frac{N}{2}$ e Y representa a quantidade de falhas, tal que $1 <= Y <= K$.

Tabela 4.1: *Combinação das variáveis avaliadas pelo Under Pressure benchmark*

K	**F**	**Reinserção**	**Relevância**	**Consideração**
0	0	Não	Sim	Base de comparação para os outros cenários
0	Y	Não	Não factível	Número de falhas maior que o tolerado ($F > K$)
X	0	Não	Sim	Permite avaliar o impacto da utilização dos mecanismos de tolerância a falhas
X	Y	Não	Sim	Permite avaliar o impacto das falhas
0	0	Sim	Não factível	Nenhum nó falhou para ser reinserido ($F = 0$)
0	Y	Sim	Não factível	Número de falhas maior que o tolerado ($F > K$)
X	0	Sim	Não factível	Nenhum nó falhou para ser reinserido ($F = 0$)
X	Y	Sim	Sim	Permite avaliar a reinserção dos nós no *cluster*

Após a avaliação da combinação das variáveis, os quatro cenários propostos para o UPB são os seguintes:

1. *Cluster* sem tolerância à falhas ($K = 0$) e sem falhas ($F = 0$).

2. *Cluster* com tolerância à falhas ($K > 0$) e sem falhas ($F = 0$).

3. *Cluster* com tolerância à falhas ($K > 0$) e com falhas ($0 < F <= K$).

4. *Cluster* em recuperação com um nó sendo reinserido ($K > 0$, $F > 0$)

Vale ressaltar que, como os números de K e F variam conforme o ambiente, a avaliação de cada um desses cenários requer a execução de vários passos.

4.3.2 Medição do Desempenho Sustentável

Durante a execução do UPB em diversos momentos é necessário mensurar o desempenho do *cluster*, ou seja, definir quantas transações por segundo o *cluster* está processando em

determinada situação. Para determinar esse desempenho algumas regras, descritas nessa Seção, devem ser seguidas.

Quando um cliente submete uma carga de trabalho contínua para um SGBD, seja ele distribuído ou não, comumente em um primeiro momento o desempenho máximo não é atingido. Durante esse período inicial, chamado de **aquecimento**, o desempenho do *cluster* apresenta instabilidades. Após certo ponto o desempenho estabiliza, e o cluster pode ser considerado em **estado estável**. Esse comportamento é ilustrado na figura 4.1.

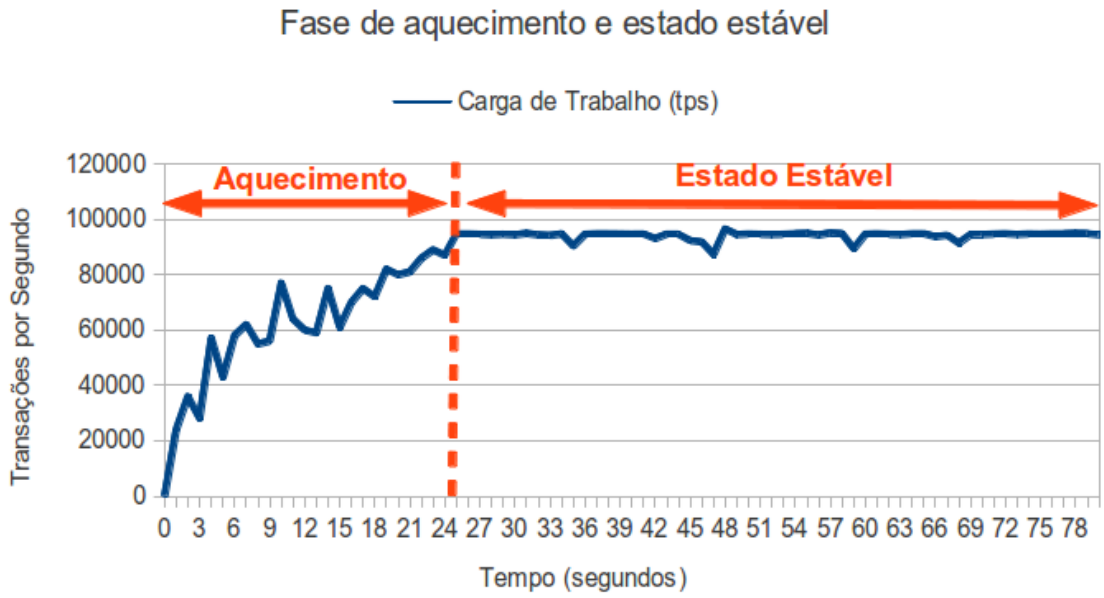


Figura 4.1: Fase de aquecimento e estado estável.

Esse comportamento pode ser ocasionado por diversos motivos (*e.g.*, movimentação dos dados entre diferentes níveis de memória, espaços de memória sendo preenchidos, roteamento da rede). Após o estado estável ser atingido, tipicamente a oscilação do desempenho ao longo do tempo é pequena, de até 2% [36]. Porém isso depende dos diversos componentes envolvidos no ambiente (*e.g.*, *hardware*, SGBDD, configurações, forma de conexão entre cliente e *cluster*).

O **desempenho sustentável** é aquele obtido com o *cluster* em estado estável. Todas as medições de desempenho utilizadas no UPB devem ser feitas com o *cluster* em estado estável. O desempenho do *cluster* durante o período de aquecimento deve ser desconsiderado no cálculo das métricas.

Apesar do estado estável ser fácil de ser definido e observado, ele é difícil de ser provado [36]. Além disso as variações de desempenho no estado estável variam conforme peculiaridades de implementação cada ambiente. Então, assim como acontece no TPC-C, o UPB não prevê um método ou regras rígidas para definir o estado estável. Cabe ao responsável pela execução do *benchmark* verificar o tempo necessário para o aquecimento e garantir que o *cluster* se encontre no estado estável.

Quando o *cluster* está em estado estável a variação do seu desempenho é relativamente pequena. Isso significa que nessa situação poucos segundos são suficientes para ter uma boa caracterização do desempenho do *cluster*. Para o UPB foi definido que a medição do desempenho deve resultar da média de transações por segundo processadas durante um período de 2 minutos em estado estável. Esse período de 2 minutos foi definido empiricamente como um intervalo suficiente para capturar o comportamento do sistema e suficiente para compensar eventuais instabilidades pontuais.

Para que as medições sejam precisas, em cada etapa da *benchmark* os clientes devem ser configurados para submeter um limite máximo de carga de trabalho (*i.e.*, transações por segundo), representado por Q. Dessa forma evita-se que todos os clientes utilizados submetam o máximo possível de transações simultaneamente, o que poderia colocar o SGBDD em um estado de estresse e gerar instabilidades no seu desempenho.

Resumindo, três regras básicas devem ser seguidas ao medir o desempenho do *cluster*:

- A medição deve ser feita com o *cluster* em estado estável.
- O resultado da medição deve ser a média de transações por segundos processadas durante 2 minutos.
- Deve ser configurado a quantidade máxima de carga de trabalho (*i.e.*, transações por segundo) submetidas por cada cliente (Q).

4.3.3 Desempenho Máximo Sustentável

Várias métricas do UPB são baseadas no desempenho máximo do SGBDD, ou seja, em quantas transações por segundo o *cluster* é capaz de processar de forma sustentável em

determinada situação. O método descrito nesta seção deve ser seguido para fazer esta medição sempre que necessário durante a execução dos passos do *benchmark* que serão apresentados nas Seções seguintes.

É importante observar que sempre que se fala em desempenho máximo na UPB, considera-se que é a carga máxima aplicada, medida em transações por segundo, que atenda o requisito de latência ϕ definido na Seção 4.2.1.

Durante a execução do UPB, na primeira vez que for necessário realizar uma medição de desempenho deve-se determinar antes qual o limite de carga que os clientes individualmente são capazes de submeter para o *cluster*, representado por L_c. Esse limite é definido para evitar que sejam geradas contenções de desempenho no lado do cliente devido a carga de trabalho excessiva gerada por ele, o que poderia ocasionar aumento de latência e instabilidade de desempenho.

Uma vez definido L_c, pode-se então mensurar o desempenho máximo sustentável do cluster.

4.3.3.1 Medição do Limite de Carga dos Clientes

Para definir L_c, um único cliente deve ser utilizado em um processo iterativo de busca da carga máxima. O processo é descrito a seguir:

1. A busca deve iniciar realizando a medição de desempenho (conforme apresentado na Seção 4.3.2) com o cliente configurado com um nível baixo de carga (Q), cuja latência atenda o requisito de latência ϕ. Esse nível de carga inicial Q pode ser qualquer, porém quanto mais próximo for do L_c, menos iterações serão necessárias.

2. Incrementar o nível de carga aplicado pelo cliente e realizar nova medição de desempenho.

   Observação: O incremento pode variar conforme a precisão que se deseja obter (quanto maior o incremento, menos preciso será o valor final encontrado). Recomenda-se incrementos de até 10%.

3. Deve-se verificar se a execução realizada atende as seguintes condições:

(a) Se o requisito de latência ϕ foi atingido.

(b) Se o desempenho obtido está compatível com a carga Q configurada nos clientes, com uma tolerância de até 5%. Ou seja, se a seguinte condição for satisfeita:

$$Q - 5\% <= Desempenho <= Q + 5\%$$

, onde Q representa a carga de trabalho máxima aplicada pelo cliente e *Desempenho* é o resultado da medição de desempenho do *cluster*.

Se as duas condições forem satisfeitas, então retorna para o passo 2.

Se não, L_c é a carga aplicada na iteração anterior.

Como o L_c corresponde ao limite de desempenho dos clientes individualmente, que é independente do limite de desempenho do *cluster*, ele pode ser definido uma única vez durante o *benchmark* e reutilizado ao longo de todas as suas etapas.

4.3.3.2 Medição do Desempenho Máximo Sustentável

Para mensurar o desempenho máximo sustentável em determinado cenário os seguintes passos devem ser seguidos:

1. Iniciar com um cliente submetendo L_c transações por segundo ($Q = L_c$).

2. Adicionar um novo cliente com carga inicial Q mínima. Esse nível de carga inicial pode ser qualquer desde atenda o requisito de latência ϕ.

3. Incrementar o nível de carga Q aplicado pelo último cliente adicionado e realizar nova medição de desempenho.

   Observação: O incremento pode variar conforme a precisão que se deseja obter (quanto maior o incremento, menos preciso será o valor final encontrado).

   Se a carga desse último cliente já for máxima ($Q = L_c$), então voltar para Passo 2.

4. Deve-se verificar se a execução realizada atende as seguintes condições:

   (a) Se o requisito de latência ϕ foi atingido.

(b) Se o desempenho do cluster obtido está compatível com a carga Q total configurada nos clientes, com uma tolerância de até 5%. Ou seja, se a seguinte condição for satisfeita:

$$Q_{total} - 5\% <= Desempenho <= Q_{total} + 5\%$$

, onde Q_{total} representa a soma da carga de trabalho Q de todos os clientes e *Desempenho* é o resultado da medição de desempenho do *cluster*.

Se as duas condições forem satisfeitas, então retorna para o Passo 3.
Se não, o *Desempenho* da iteração anterior (em que as condições foram satisfeitas) é o desempenho máximo do cluster.

4.3.4 Passos para a Execução da UPB

Para que todos os cenários sejam avaliados o UPB é baseado em um método iterativo de *benchmark*. Esse método contém 4 passos principais e um fluxo de execução iterativo, onde os passos são executados mais de uma vez conforme o valor máximo de K que depende do ambiente que está sendo avaliado.

As métricas parciais de desempenho seguem o padrão de nomenclatura de $T_{K,F}$[1], onde K é o índice de tolerância a falha parametrizado para o passo em questão e F representa a quantidade de nós com falhas no *cluster*.

A seguir são detalhados os passos e suas respectivas métricas parciais.

4.3.4.1 Passo 1 - *cluster* completo e sem tolerância à falhas

Nesse cenário os mecanismos configuráveis que aumentam a disponibilidade e provêm tolerância a falha são desabilitados ($K = 0$ e $F = 0$). Essa configuração não é recomendada para um ambiente de produção, porém no UPB esse passo é importante para ser utilizado como base de comparação.

[1] A letra T foi escolhida para representar essa métrica devido ao termo em inglês *throughput* que é usado para indicar a quantidade de transações por segundo de um sistema.

Nessa etapa espera-se obter o desempenho máximo que o *cluster* é capaz de oferecer já que não existe o custo computacional adicional para a manutenção das réplicas e, como o *cluster* está completo, toda a capacidade computacional está disponível para o processamento de transações.

O desempenho máximo sustentável nesse cenário é representado por $T_{0,0}$.

A execução desse passo deve ser feita em duas etapas:

1. **Definições iniciais:** Como este é o primeiro passo da execução do UPB, é necessário definir dois parâmetros utilizados ao longo de todo o *benchmark*: Tempo de aquecimento e quantidade máxima de transações por cliente (L_c).

   O tempo de aquecimento deve ser definido através da análise do comportamento do desempenho do cluster, como discutido na Seção 4.3.2. Já para definir L_c é necessário aplicar o método apresentado na Seção 4.3.3.1.

2. **Desempenho do cluster com K=0 e F=0:** Uma vez gerados os parâmetros inicias, o desempenho máximo sustentável do cluster $T_{0,0}$ pode ser obtido seguindo o método da Seção 4.3.3.2.

4.3.4.2 Passo 2 - *cluster* completo e com tolerância a falhas

A utilização de mecanismos de tolerância à falhas pode impactar no desempenho geral do *cluster* mesmo quando ele está completo, sem falhas. Isso acontece porque a manutenção das réplicas pelo SGBDD consome recursos computacionais (*i.e.*, processamento e memória) antes disponíveis para o processamento de transações. Esse impacto varia conforme as estratégias específicas implementadas por cada fabricante de SGBDD. O objetivo do Passo 2 é verificar se realmente existe esse impacto e, se existir, mensurá-lo.

Durante a aplicação do *benchmark* este cenário será executado com diferentes configurações de índices de tolerância a falha ($K > 0$ e $F = 0$). Para cada execução deve-se obter $T_{K,0}$, que representa o desempenho máximo sustentável do *cluster* configurado com índice de tolerância K.

A partir de $T_{K,0}$, deve-se gerar a taxa de degradação de desempenho ($D_{K,0}$) que representa o percentual de perda da desempenho quando comparado a um ambiente sem tolerância a falha ($T_{0,0}$ do passo anterior). Essa métrica pode ser escrita da seguinte forma:

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

4.3.4.3 Passo 3 - *cluster* com falhas toleradas

Quando configurado para tolerar falhas ($K > 0$), o *cluster* deve ser capaz de permanecer *online*, recebendo e processando transações, mesmo quando um ou mais nós sairem da rede por qualquer motivo. Porém, apesar dos mecanismos de tolerância, esta situação pode apresentar diferentes efeitos colaterais como perda de desempenho ou instabilidades.

O desempenho do cluster nesse cenário está relacionado a diversos fatores:

- **Carga de Trabalho - Operações de leitura:** Como esse tipo de transação não modifica os dados elas podem ser distribuídas entre os nós do cluster. Uma estratégia de otimização adotada é que cada transação de leitura seja enviada para uma das réplicas, dessa forma a vazão do cluster (*i.e.*, transações por segundo) aumenta conforme a quantidade de nós. Nesse caso, a perda de nós significa menor capacidade de processamento de transação, o que impacta diretamente no desempenho.

- **Carga de Trabalho - Operações de escrita:** Para esse tipo de operação o SGBDD deve manter a sincronia dos dados escritos entre os nós replicados. Cada fabicante adota uma estratégia diferente para isso. Mas, de forma geral, pode-se considerar que com menos réplicas o custo computacional de manter a sincronia entre elas é menor, o que pode fazer o desempenho até aumentar com a ocorrência de falhas.

- **Redução da capacidade computacional:** Com menos nós no cluster, os recursos computacionais disponíveis (*i.e.*, processamento e memória) diminuem. Esses recursos podem ser relacionados tanto a processamento de transações como para

atividades de gestão do cluster. Com menos recursos disponíveis, a tendência é diminuição do desempenho.

O objetivo desta etada é verificar a variação de desempenho causada pela saída de nós da rede. Para isso deve-se verificar o impacto causado pela falha de diferentes quantidades de nós, começando com $F = 1$ até o máximo tolerado, $F = K$ $(0 < F <= K)$. A sequência de passos para executar essa validação é apresentada a seguir:

1. Iniciar com o *cluster* completo, sem falhas $(F = 0)$ e configurado com índice de tolerância $K > 0$.

2. Remover um nó do *cluster* simulando uma falha.

3. Obter $T_{K,F}$, que é o desempenho máximo do *cluster* com índice de tolerância K e F falhas.

4. Comparar o desempenho obtido com o desempenho do *cluster* sem falhas ($T_{K,0}$ obtido no passo anterior) e gerar a seguinte taxa de degração de desempenho:

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

5. Enquanto o *cluster* tolerar mais falhas $(K > F)$ voltar para passo 2.

Após a execução desses passos temos a degradação de desempenho $D_{K,F}$ do sistema quando sujeito a diferentes quantidades de falhas.

Com base nos valores de degradação de desempenho $D_{K,F}$ deve ser gerada a métrica parcial *Degradação do cluster com falhas*, DF_K. O objetivo é sumarizar em uma única métrica a degradação do desempenho do cluster, configurado com índice K, quando sujeito a diferentes quantidades falhas. Essa métrica é calculada através da seguinte média ponderada:

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

Um ponto importante considerado nessa métrica é que as situações de maior probabilidade tem um peso maior do que outras menos prováveis. Como os nós do *cluster* apresentam *hardware* e *software* iguais, podemos assumir que a probabilidade de falhas de todos eles é a mesma. Com isso a probabilidade da ocorrência de falhas simultâneas em dois nós é menor do que a ocorrência de uma única falha. Esse fato é considerado na métrica através dos pesos das médias ponderadas que são inversamente proporcionais ao número de falhas do *cluster*. Ou seja, quanto maior a quantidade de falhas simultâneas, menor o peso do seu impacto na métrica final.

Por fim, a equação é dividida pela soma dos seus pesos como uma forma de normalização. Isso é feito para que essa métrica possa ser comparada entre diferentes configurações e sistemas.

Nessa métrica quanto menor for o valor obtido, menor é a perda de desempenho e melhor é o SGBDD/Ambiente. Essa métrica parcial será utilizada posteriormente no cálculo de uma métrica final.

4.3.4.4 Passo 4 - *cluster* em recuperação com a reinserção de nós

Quando um ou mais nós são reinseridos no *cluster* após a ocorrência de falhas ocorre um processo de recuperação com várias etapas. Cada fabricante de SGBDD adota suas próprias estratégias, mas podemos citar como exemplo o rebalanceamento da carga e dos dados do *cluster*, a cópia dos dados para o novo nó e atualização do catálogo.

Esse processo de recuperação do *cluster* com a entrada de um nó leva determinado tempo e durante esse intervalo pode gerar impactos no desempenho. Um impacto direto que pode ser citado é o tráfego adicional na rede durante a cópia dos dados para o novo nó.

O passo 4 visa medir a duração do processo de reinserção de todos os nós, um a um, quando o *cluster* tem o número máximo de falhas ($F = K$). Essa duração do processo, mensurada em segundos, indica quanto tempo leva entre o comando para um nó ser reinserido e a completa estabilização do *cluster*, quando o novo nó já recebeu todos os

dados e está disponível para processar transações.

Seja um *cluster* com índice de degradação $K > 0$ e com o número máximo de falhas toleradas ($F = K$). $E_{K,I}$ é o tempo para estabilização do i-ésimo nó a ser reinserido no *cluster*, considerando que nenhum outro nó apresente falhas entre o processo de reinserção do primeiro e do último nó.

Com isso, a métrica parcial E_K, que avalia o tempo de estabilização para o *cluster* com índice K, pode ser representada como uma média ponderada, de forma semelhante a utilizada no passo 3:

$$E_K = \frac{\sum_{I=1}^{K} (\frac{1}{I} * E_{K,I})}{\sum_{I=1}^{K} (\frac{1}{I})}$$

Assim como no passo 3, os pesos dessa média representam a probabilidade maior da ocorrência de falhas em poucos nós do que em muitos de forma concorrente.

4.4 Métricas Finais

Com a execução dos passos do UPB diversas métricas parciais são geradas. Essas métricas representam a disponibilidade do SGBDD em uma situação específica, com um determinado índice de tolerância a falha (K). Para sumarizar esses números de forma que representem o sistema como um todo, independente de cenários específicos, o UPB propõem três métricas finais. Essas métricas finais podem ser utilizadas para a classificação e comparação de diferentes SGBDDs.

O cálculo das métricas finais é realizado através da média aritmética de determinadas métricas parciais, como apresentado a seguir:

Métrica	**Descrição**	**Métrica parcial utilizada**
D_T	Degradação de desempenho da tolerância a falhas	$D_{K,0}$, gerada no passo 2
D_F	Degradação de desempenho durante a ocorrência de falhas	DF_K, gerada no passo 3
E_F	Tempo de estabilização para a reinserção de nós	E_K, gerada no passo 4

A seguir será apresentado o detalhamento das três métricas finais. Os valores obtidos representam a degradação de algum aspecto do ambiente, ou seja, quanto menor forem os valores das métricas finais melhor é o SGBDD.

4.4.1 Degradação de desempenho da tolerância a falhas

Essa métrica, representada por D_T, indica a degradação de desempenho do *cluster* quando são utilizados mecanismos de tolerância a falha, como a replicação. Ela é expressa pela seguinte fórmula:

$$D_T = \frac{\sum_{i=1}^{m} D_{i,0}}{m}$$

Onde m representa o maior valor do índice K avaliado nesse ambiente e $D_{K,0}$ é a degradação de desempenho obtida no passo 2 do *benchmark* quando é utilizado um índice de tolerância a falhas K.

4.4.2 Degradação de desempenho durante a ocorrência de falhas

No passo 3 foi gerado a métrica parcial DF_K que representa a degradação de desempenho do SGBDD na ocorrência de falhas quando está configurado com índice de tolerância K.

A métrica final que sumariza a degradação de desempenho do SGBDD durante a ocorrência de falhas independente de configurações específicas, representada por D_F, é calculada da seguinte forma:

$$D_F = \frac{\sum_{i=1}^{m} DF_i}{m}$$

Onde m representa o maior valor do índice K avaliado nesse ambiente e DF_K é a degradação de desempenho obtida no passo 3 do *benchmark* quando é utilizado um índice de tolerância a falhas K.

4.4.3 Tempo de estabilização para a reinserção de nós

No passo 4, para expressar o tempo de estabilização de um nó quando está sendo reinserido no *cluster* configurado com índice de tolerância de K foi gerada a métrica E_K. A partir dela podemos gerar a métrica final E_F através da fórmula a seguir:

$$E_F = \frac{\sum_{i=1}^{m} E_i}{m}$$

De forma análoga as outras métricas, m representa o maior valor do índice K avaliado nesse ambiente e E_K é o tempo de estabilização obtido no passo 4 quando utilizado um índice de tolerância a falhas K.

4.5 Considerações Finais

A UPB avalia três aspectos de grande relevância para a disponibilidade de um SGBDD: (1) A degradação de desempenho gerada pela utilização dos mecanismos de tolerância a falha, (2) degradação de desempenho do *cluster* durante a ocorrência de falhas toleradas e (3) o tempo de recuperação do *cluster* quando um nó é reinserido após a ocorrência de falhas.

Para a avaliação da disponibilidade foram apresentadas métricas parciais e finais apresentadas na Tabela 4.2. Essas métricas podem ser utilizadas tanto para a comparação entre SGBDDs de diferentes fabricantes como para a comparação de diferentes configurações possíveis para um ambiente.

Tabela 4.2: *Métricas geradas pelo UPB*

Métrica	**Tipo**	**Descrição**
$T_{K,F}$	Parcial	Desempenho do *cluster* (em tps) com índice de tolerância K e F nós indisponíveis
$D_{K,0}$	Parcial	Degradação de desempenho do *cluster* com índice K e sem falhas
$D_{K,F}$	Parcial	Degradação de desempenho do *cluster* com índice K e com F nós com falhas
DF_K	Parcial	Índice para representar a degradação de desempenho do *cluster* com falhas e índice K
$E_{K,I}$	Parcial	Tempo de reinserção do I-ésimo nó do *cluster* com índice K
E_K	Parcial	Índice para representar o tempo de reinserção de nós do *cluster* índice K
D_T	Final	Degradação de desempenho da tolerância a falhas
D_F	Final	Degradação de desempenho durante a ocorrência de falhas
E_F	Final	Tempo de estabilização para a reinserção de nós

CAPÍTULO 5

RESULTADOS EXPERIMENTAIS

Para validar o *Under Pressure Benchmark* avaliamos como estudo de caso o SGBD distribuído VoltDB [10] que possui arquitetura *shared-nothing*, onde os nós do cluster não compartilham recursos entre eles. Cada nó do VoltDB constitui uma unidade independente de processamento, capaz de processar ou rotear transações dentro do cluster e armazenar uma partição do banco de dados. Adicionalmente, a arquitetura do VoltDB requer o armazenamento de todos os dados em memória principal. Mais detalhes do VoltDB foram apresentados na Seção 2.4.

Os experimentos desse trabalho foram executados utilizando máquinas do projeto Grid'5000 [2, 15] localizado na França. O objetivo do projeto é prover uma plataforma computacional altamente configurável e controlável para pesquisas de sistemas distribuídos e de larga escala. Ele é composto por diversos cluster que diferem entre si pela configuração das máquinas e rede.

Nas próximas seções serão apresentados detalhes do experimento, começando pela descrição do ambiente utilizado, na Seção 5.0.1 e do Banco de Dados e da Carga de Trabalho utilizada, nas Seções 5.1 e 5.2, respectivamente. Em seguida os resultados gerados em cada um dos passos do UPB e as métricas calculadas são apresentadas na Seção 5.3. Para finalizar, algumas considerações finais sobre o experimento são feitas na Seção 5.4.

5.0.1 Ambiente de Hardware e Software

A escolha do cluster para o experimento foi baseada na especificação da rede de comunicação entre as máquinas, pois o SGBD VoltDB requer grande tráfego de mensagens distribuídas devido sua grande vazão no processamento de transações. Portanto, a rede de comunicação entre as maquinas do cluster é um fator de desempenho de grande im-

portância.

Nos experimentos foi utilizado o cluster *ChinqChint*, localizado no site da cidade de Lille [1], onde todas as máquinas utilizadas possuem a mesma configuração e adaptadores de rede 10 Gigabit, como apresentado na Tabela 5.1:

Tabela 5.1: *Configuração das máquinas utilizadas [1]*

Recurso	**Descrição**
Processador	Intel Xeon E5440 QC (2.83 GHz / 4 MB)
Quantidade de processadores	2
Cores por processador	4
Total de cores	8
Memória	8 GB
Rede	Myri-10G (10G-PCIE-8A-C)
Sistema Operacional	Debian GNU/Linux Lenny x64
SGBD	VoltDB v2.8.4.1 *Community Edition*

A versão do VoltDB utilizada (v2.8.4.1) é a mais atual na data de execução dos experimentos. E a edição *VoltDB Community Edition* foi utilizada por ser de livre distribuição.

Nos experimentos o cluster VoltDB é composto por 6 máquinas e outras 3 máquinas serão utilizadas como clientes para submeter transações para o SGBDD. Os nomes das máquinas utilizadas como cliente e cluster são listados na Tabela 5.2.

Tabela 5.2: *Máquinas do cluster ChinqChint [1], do Grid'5000, utilizadas nos experimentos*

Cluster VoltDB		**Clientes**
chinqchint-12	chinqchint-13	chinqchint-25
chinqchint-14	chinqchint-15	chinqchint-26
chinqchint-2	chinqchint-22	chinqchint-40

O cluster VoltDB utilizado é composto por 6 nós e cada um deles possui 6 sites, totalizando 36 partições. A quantidade de sites por nó foi definida seguindo a recomendação da documentação [42] de utilizar $\frac{3}{4}$ da quantidade de cores existentes do servidor. O arquivo de configuração *deployment.xml*, onde são especificados as configurações gerais do cluster, é apresentado abaixo. Nessas configuração, o índice de replicação (*k-factor*) varia conforme o passo da metodologia.

```
<?xml version="1.0"?>
<deployment>
   <cluster hostcount="6"
            sitesperhost="6"
            kfactor="0"
   />
</deployment>
```

Para a execução do processo do VoltDB, a área de memória *heap* do Java foi definida com tamanho variando entre 1.500MB e 3.500MB (parâmetros do java *-Xms1500m -Xmx3550m*). A *heap* é utilizada pelo VoltDB para a área de memória temporária responsável por armazenar a fila de chamadas de procedures. Foi definido um tamanho relativamente grande para essa área de memória, e em conformidade com as recomendações da documentação do VoltDB [42, 41], para que o cluster não apresentasse contenção de desempenho por causa de tamanho da memória.

Os clientes foram implementados em linguagem de programação Java versão 1.7 e cada cliente é executado em uma máquina distinta e com a sua própria máquina virtual.

Para simplificar a nomenclatura, na sequência desse trabalho o termo *cluster* será utilizado para se referir ao cluster VoltDB completo, e nó para referenciar qualquer uma das máquinas que compõem este cluster.

5.1 Banco de Dados

O esquema do banco de dados utilizado é baseado em uma única tabela, de nome *USERTABLE*, com 11 colunas capazes de armazenar até 1000 caracteres cada. Uma das colunas, de nome *YCSB_KEY*, é a chave da tabela. As outras são campos de dados que recebem o nome *FIELD1*, *FIELD2* e assim por diante até *FIELD10*.

O SQL utilizado pelo VoltDB como schema é o seguinte:

```
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)
);
```

Essa tabela é populada através do gerador de dados do YCSB [11]. Os dados utilizados são sequências aleatórias de caracteres geradas automaticamente. Nesse experimento a tabela foi populada com 1 milhão de tuplas.

5.2 Carga de Trabalho

A carga de trabalho desse experimento foi gerada através do YCSB [11], que é uma plataforma para execução de *benchmarks* em SGBDs Distribuídos que foi apresentado na Seção 3.2. Foi utilizado o *YCSB Core Workload*, que é uma carga de trabalho fornecida pelo YCSB, configurada com 100% de operações de leitura, que correspondem a operações de projeção e seleção no banco de dados que podem ser traduzidas em transações SQL com o seguinte formato:

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

O arquivo de configuração da carga de trabalho utilizado pelo YCSB é o seguinte:

```
recordcount=1000000
workload=com.yahoo.ycsb.workloads.CoreWorkload
readallfields=true
```

```
readproportion=1
updateproportion=0
scanproportion=0
insertproportion=0
requestdistribution=zipfian
```

Junto com o YCSB são fornecidos conectores para alguns SGBDDs, porém o do VoltDB não é incluído nativamente. Além disso até a data em que esses experimentos foram executados, não existia nenhum conector público na Internet do YCSB com o VoltDB. Por esse motivo tivemos que implementar o nosso próprio conector.

Um fator importante no desenvolvimento do conector é com relação a sincronia das transações. A maioria dos SGBDs e aplicações utilizam transações síncronas, onde cada linha de execução da aplicação cliente (*thread*) aguarda o resultado das transações submetidas para continuar o seu processamento. Dessa forma cada linha de execução só é capaz de submeter uma transação de cada vez. Já o VoltDB permite e recomenda para melhor desempenho a utilização de transações assíncronas. Nesse caso, quando a aplicação cliente submete uma transação para o SGBDD ela é colocada em uma fila de processamento e o cliente cria um objeto de *callback* que é notificado quando a transação tiver um retorno. O *callback* passa a ser responsável por processar os resultados da transação e a linha de execução principal, que submeteu a transação, pode continuar seu processamento independente da resposta da transação, podendo inclusive submeter outras transações antes da primeira ser respondida.

Para obter melhor desempenho do VoltDB o conector desenvolvido é baseado em transações assíncronas. Porém isso traz outra situação. O comportamento do gerador de carga de trabalho do YCSB é baseado em transações síncronas. A geração de estatísticas sobre a carga de trabalho aplicada considera o comportamento síncrono, onde cada linha de execução do gerador de carga submete a transação e aguarda seu retorno para continuar. Então, para gerar estatísticas de transações assíncronas modificando o mínimo possível do YCSB, a estratégia adotada foi desenvolver um monitor de desempenho es-

pecífico para o VoltDB dentro do conector.

O conector foi desenvolvido em Java seguindo uma interface bem definida do YCSB com os métodos que devem ser implementados, incluindo um para cada uma das seguintes operações: Leitura, leitura em lote, atualização, inserção e remoção. As operações leitura, inserção e remoção foram implementadas utilizando as *stored procedures* fornecidas nativamente pelo VoltDB. Para leitura em lote e atualização foi necessário desenvolver *stored procedures* específicas.

5.3 Resultados

Nesta seção detalhamos a execução e resultados de cada um dos passos da UPB. Ao final discutimos as métricas alcançadas pelo SGBD VoltDB.

5.3.1 Passo 1

Durante a execução do UPB, dois parâmetros determinam como os experimentos devem ser executados: Tempo de aquecimento e quantidade máxima de transações por cliente (L_c). Esses parâmetros dependem do ambiente e devem ser definidos durante a primeira execução da carga de trabalho. Após essas definições iniciais, deve ser definido o desempenho máximo do cluster sem falhas ($F = 0$) e configurado com $K = 0$.

5.3.1.1 Definições iniciais

As Figuras 5.1 e 5.2 apresentam o desempenho do cluster, em transações por segundo, e latência, em milisegundos (ms), respectivamente, durante a aplicação da carga de trabalho. Nesses gráficos estão representadas cinco execuções diferentes, sendo que para cada uma o cliente estava com diferentes configurações de Q (máximo de transações por segundos submetidas pelo cliente).

O gráfico da Figura 5.1 indica que em todas as execuções a quantidade de tps respondidas pelo cluster apresentou um comportamento estável após 2 segundos, com uma variação inferior a 10%. Já a latência, Figura 5.2, levou 18 segundos para apresentar um

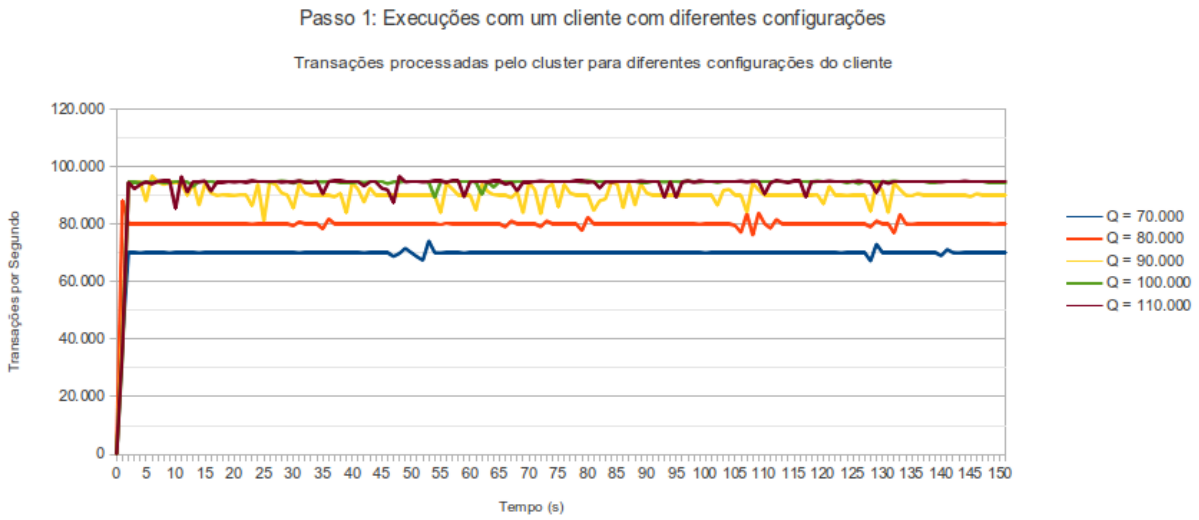


Figura 5.1: Desempenho do cluster durante execuções com um cliente e diferentes configurações de Q.

comportamento estável em uma das execução (cliente com $Q = 90.000$). Durante essas execuções a latência de todas as transações foi inferior a 1 segundo, atendendo o requisito de latência ϕ.

Este passo mostra que o SGBD passa por uma fase transitória de aquecimento durante a qual ele se comporta de maneira diferente do que quando aquecido (*i.e.*, fase de operação contínua), uma vez que a área de *buffer* ainda não está preenchida com dados.

Com isso podemos definir que o tempo de aquecimento para o *benchmark* deve ser superior a 18 segundos. Nesse trabalho será utilizado um período de 30 segundos.

Na Figura 5.3 é apresentado o desempenho sustentável do cluster, que representa a média de tps durante 2 minutos. Também é apresentado nesse gráfico a variação entre o desempenho atingido e o valor de Q dos clientes, que é a configuração do máximo de tps que o cliente pode submeter.

Com o índice Q de até 90.000 tps, o cliente foi capaz de submeter a carga máxima definida pela sua configuração (a variação entre Q e o desempenho foi de apenas 0,04%). Nas execuções com Q superior a 90 mil, os clientes foram capazes de submeter aproximadamente 94 mil tps. Nessas execuções a variação entre o desempenho e o Q ultrapassa 5%. Por esse motivo, seguindo as regras definidas pela metodologia, a carga máxima submetida pelos clientes será de 90.000 tps.

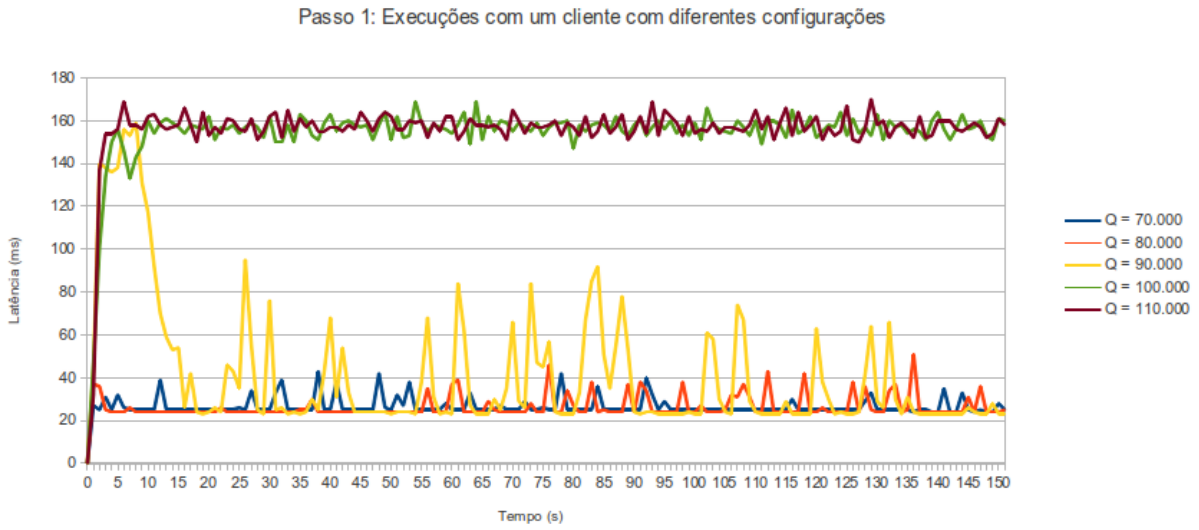


Figura 5.2: Latência média das transações durante execuções com um cliente e diferentes configurações de Q.

Através do gráfico 5.2, pode-se observar também que a latência aumenta significativamente nas execuções onde o Q não é atingido ($Q > 90.000$), chegando a ser quatro vezes maior do que nas outras execuções. Esse comportamento indica que o ambiente está em um estado de estresse e confirma que o valor adotado, 90.000 tps, é a melhor opção entre as avaliadas.

Então, os parâmetros definidos são os seguintes:

- Tempo de aquecimento: 30 segundos
- L_c: 90.000 tps

5.3.1.2 Desempenho do cluster com K=0 e F=0

Para definir o desempenho máximo do cluster sem falhas e com $K = 0$ foram feitas diversas execuções incrementando a quantidade de carga de trabalho total. Em cada execução foram utilizados diferentes quantidades de clientes (variando entre 1 e 3) com diferentes configurações de Q.

Na Figura 5.4 são apresentadas as execuções realizadas. Nesse gráfico as diferentes configurações, apresentadas no eixo x, são descritas através da soma dos valores de Q dos clientes utilizados. Então, por exemplo, a execução com $Q = 250.000$ utiliza dois

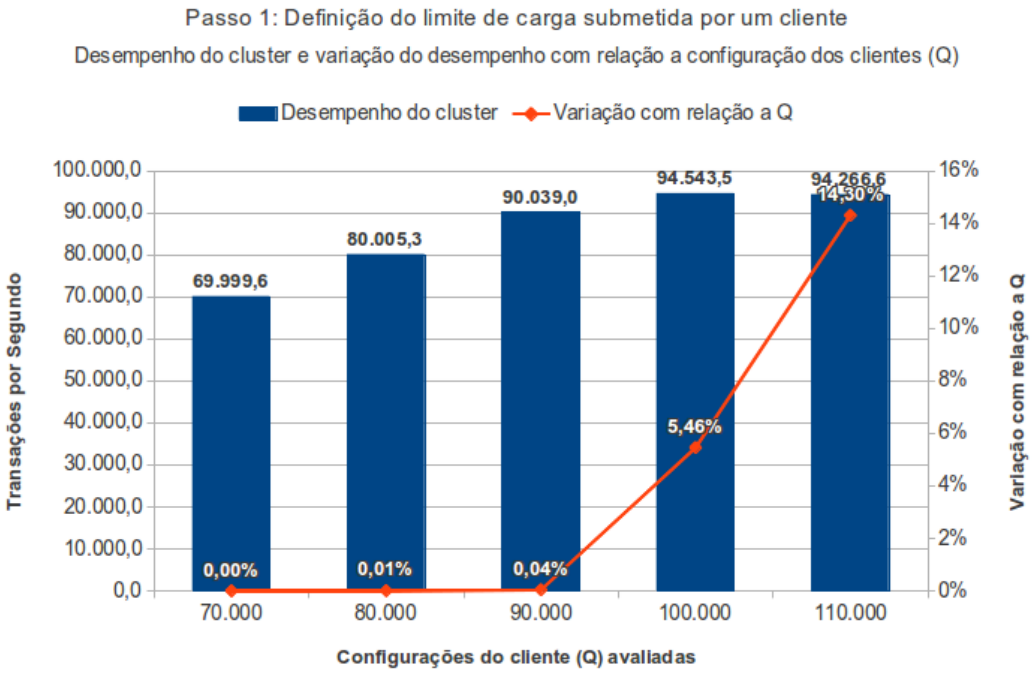


Figura 5.3: Desempenho de um cliente para diferentes configurações de Q.

clientes configurados com $Q = 90.000$ (máximo por cliente) e um cliente com $Q = 70.000$, totalizando uma carga total submetida pelos clientes de 250.000 tps.

Nesse gráfico é apresentado o desempenho atingido para cada uma das configurações e a variação entre o desempenho atingido e a configuração Q total dos clientes.

A latência não foi o fator limitador nesse ambiente já que todas as configurações avaliadas atendem o requisito de latência ϕ. Pode-se verificar que o desempenho do cluster é incremental e acompanha o valor de Q até aproximadamente 238 mil tps. Nas configurações com Q acima desse valor a variação entre desempenho e Q passa a aumentar e o desempenho permanece aproximadamente estável.

Esse comportamento de estabilização no desempenho, mesmo com o aumento da capacidade dos clientes de submeter transações, acontece devido a uma condição chamada pelo VoltDB de *backpressure* [42, 41]. Para garantir maior desempenho, os clientes do UPB submetem transações para o VoltDB de forma assíncrona, como explicado na Seção 5.2. Dessa forma, as transações submetidas, que no VoltDB são no formato de procedures, são colocadas em uma fila de chamada de procedimentos. Quando essa fila está cheia o VoltDB bloqueia a submissão de novas procedures até que espaço na fila seja liberado

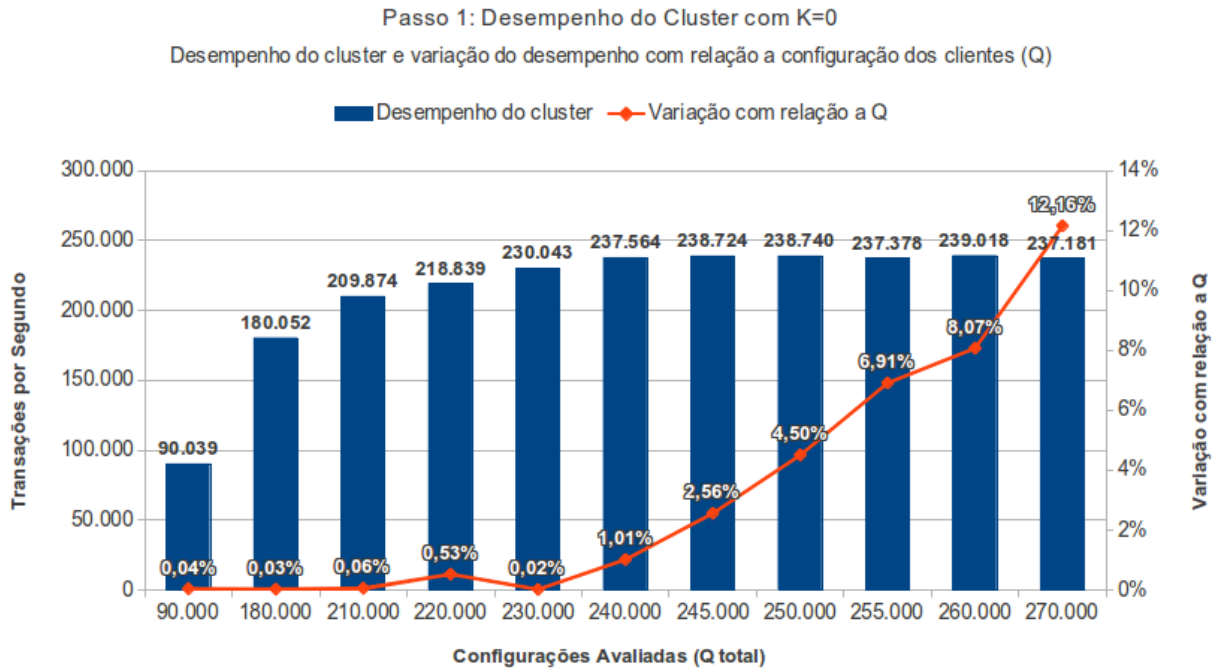


Figura 5.4: Desempenho do cluster ($K = 0$ e $F = 0$) para diferentes configurações de clientes.

para armazenar novas chamadas.

A fila de chamada de procedimentos é alocada na área de memória temporária do VoltDB e controlada através do tamanho da área de memória *heap* do Java, cuja configuração está descrita na Seção 5.0.1. Essa condição de *backpressure* gerada durante o experimento demonstra que o desempenho máximo do cluster foi atingido. Nessa situação o cluster está com a fila de chamadas de procedimentos cheia, processando transações em sua capacidade máxima.

Conforme a metodologia, $T_{0,0}$ representa o desempenho máximo que atende o requisito ϕ e tem variação inferior a 5% com relação a Q. Então, nos resultados apresentados, $T_{0,0} = 238.740$.

5.3.2 Passo 2

No passo 2 deve ser obtido o desempenho máximo do cluster sem falhas com diferentes configurações de K. O objetivo é calcular o percentual de perda de desempenho para manter a restrição K.

Para cada valor de K, variando entre 1 e 3, foram testadas diferentes configurações de

clientes (Q). As configurações testadas e seus resultados são representadas nos gráficos das Figuras 5.5, 5.6 e 5.7, cada uma para um valor de K. Esses gráficos seguem o formato do anterior, apresentando o desempenho do cluster e a variação do desempenho com relação ao Q total dos clientes.

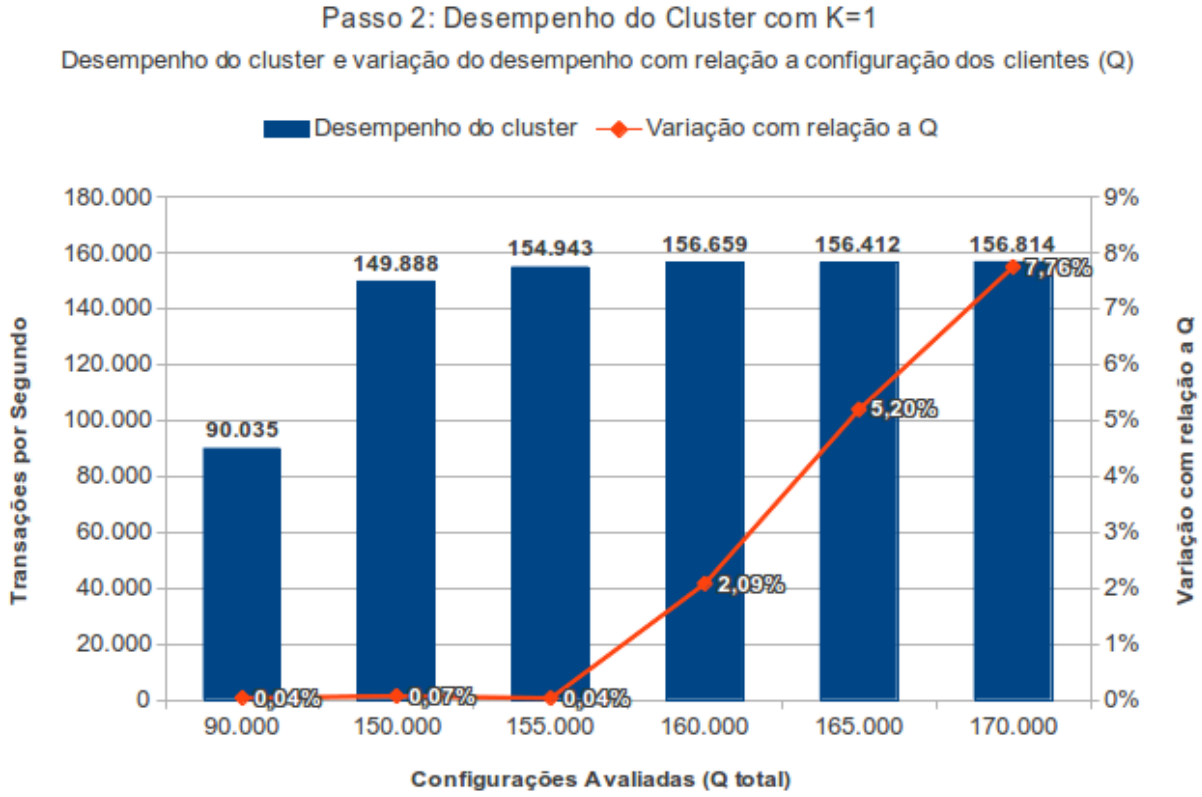


Figura 5.5: Desempenho do cluster com $K = 1$ para diferentes configurações de clientes.

A partir dos valores apresentados, as seguintes métricas parciais que representam o desempenho máximo do cluster com diferentes configurações de K são obtidas:

$$T_{1,0} = 156.659$$

$$T_{2,0} = 112.388$$

$$T_{3,0} = 101.899$$

A partir de $T_{K,0}$ é possível calcular a degradação de desempenho, $D_{K,0}$, que representa o percentual de perda de desempenho quando comparado a um ambiente sem tolerância a falha. O resultado dessa métrica é apresentado abaixo:

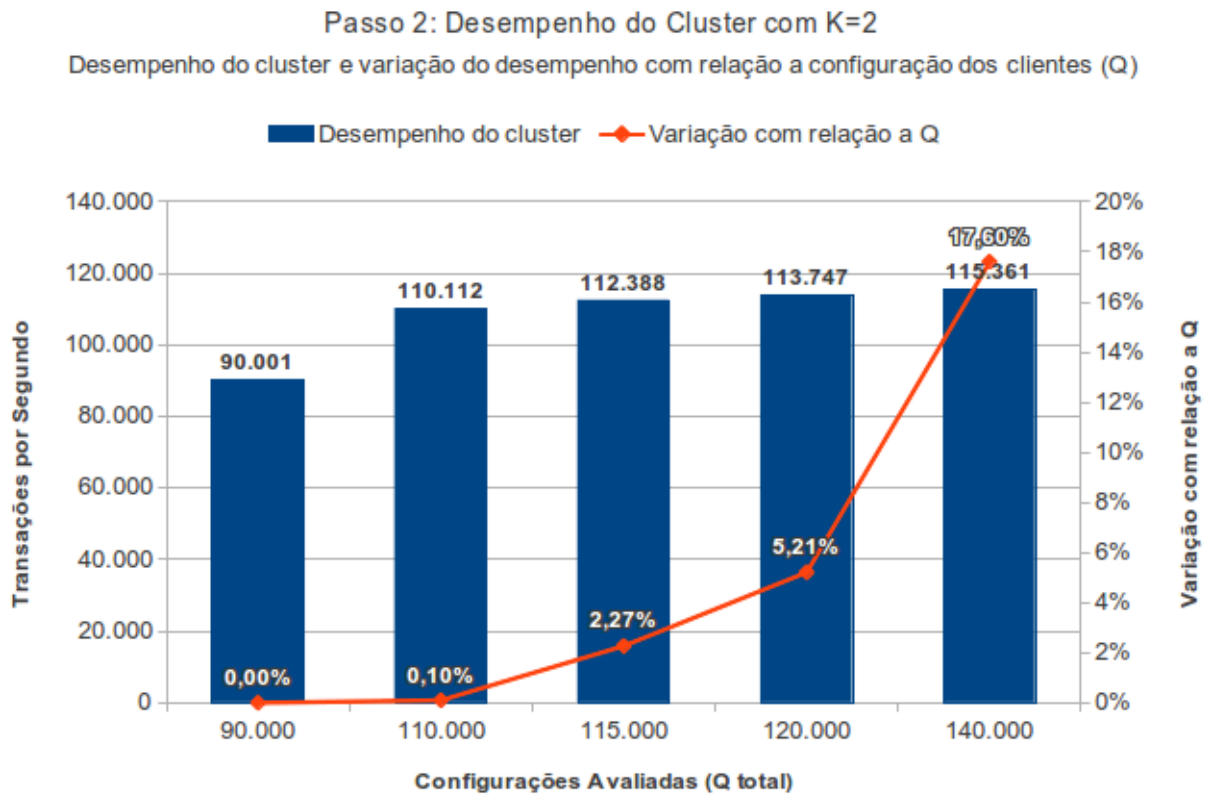


Figura 5.6: Desempenho do cluster com $K = 2$ para diferentes configurações de clientes.

$$D_{1,0} = 34,38\%$$

$$D_{2,0} = 52,92\%$$

$$D_{3,0} = 57,32\%$$

Assim como no passo anterior, nas três configurações avaliadas (K igual a 1, 2 e 3) o comportamento das execuções demonstram que o estado de *backpressure* do VoltDB foi atingido. Isso indica que o desempenho máximo do cluster em todas as configuração avaliadas foi obtido, confirmando o objetivo da metodologia.

A degradação de desempenho observada ocorre porque ao aumentar o nível de replicação, dividimos a quantidade de partições disponíveis para processar as transações. No caso do ambiente avaliado existem 36 partições, conforme Seção 5.0.1. Quando o cluster está configurado com $K = 0$, os dados estão distribuídos em 36 partições e todas elas são utilizadas para o processamento de transações de forma concorrente. Já com

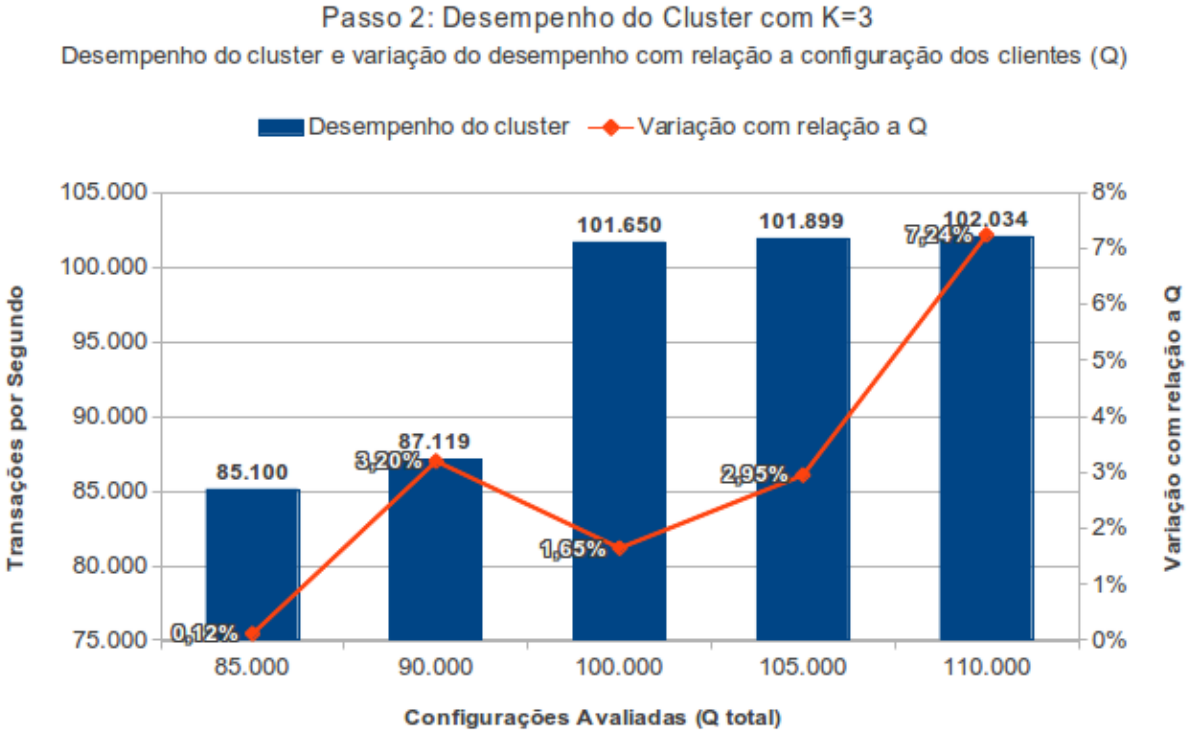


Figura 5.7: Desempenho do cluster com $K = 3$ para diferentes configurações de clientes.

$K = 1$, metade dessas partições são utilizadas para armazenar cópias dos dados e apenas 18 partições estarão disponível. Ou seja, utilizando o mesmo *hardware*, quanto maior for o nível de replicação, menor será o desempenho obtido.

Entretanto os resultados obtidos mostram que a queda de desempenho não é exatamente proporcional ao nível de replicação, como poderia se esperar ao considerar a arquitetura do (*k-safety*), descrita na Seção 2.4. Isso acontece devido a otimizações específicas da implementação dessa arquitetura pelo VoltDB. Um exemplo de otimização possível é que o processamento de transações de leitura seja realizado em apenas uma das réplicas enquanto transações de escrita sejam sempre executadas em todas as cópias.

5.3.3 Passo 3

No passo 3 deve-se obter o desempenho máximo do cluster com diferentes configurações de K e quantidades de falhas (F). O K deve variar entre 1 e 3 e a quantidade de falhas entre 1 e K. A combinação dessas variáveis resulta em 6 cenários que devem ser avaliados. Para cada cenário desses deve-se obter o desempenho máximo sustentável TK, F. Para

isso são necessárias diversas execuções, com diferentes configurações dos clientes (Q), até se alcançar o desempenho máximo desejado.

Como o número de execuções realizadas nesse experimento é grande, para simplificar a apresentação dos resultados eles são apresentados juntos na Tabela 5.3.

Tabela 5.3: *Resultados das execuções com diferentes configurações de ambiente*

Configuração	**Q**	**Desempenho (tps)**	**Variação com relação a Q**
K=1 e F=1	155.000	154.387	0,396%
K=1 e F=1	160.000	152.072	4,955%
K=1 e F=1	165.000	152.848	7,365%
K=2 e F=1	110.000	107.879	1,928%
K=2 e F=1	115.000	110.491	3,921%
K=2 e F=1	120.000	113.856	5,120%
K=2 e F=2	110.000	109.145	0,777%
K=2 e F=2	115.000	109.781	4,539%
K=2 e F=2	120.000	110.075	8,271%
K=3 e F=1	90.000	90.129	0,144%
K=3 e F=1	95.000	92.819	2,296%
K=3 e F=1	100.000	93.117	6,883%
K=3 e F=2	85.000	83.915	1,277%
K=3 e F=2	90.000	86.837	3,515%
K=3 e F=2	95.000	84.664	10,880%
K=3 e F=3	75.000	74.984	0,021%
K=3 e F=3	80.000	76.003	4,997%
K=3 e F=3	90.000	78.719	12,535%

As execuções apresentadas nessa tabela foram escolhidas para mostrar exatamente o intervalo de quando o cluster está atingindo sua capacidade máxima para cada cenário. A segunda e terceira execução de cada cenário não apresenta aumento de desempenho proporcional ao incremento do Q. Isso demonstra que o estado de *backpressure* do VoltDB foi atingido, indicando que o desempenho máximo do cluster em todas as configuração avaliadas foi obtido.

Com base nos resultados dessas execuções é possível obter $T_{K,F}$, que é o desempenho máximo obtido onde a variação entre Q e o desempenho do cluster é inferior a 5%. Também é possível obter $D_{K,F}$ que representa a degradação de desempenho entre o cluster com e sem falhas. Essas métricas parciais são apresentadas na tabela 5.4.

Tabela 5.4: *Métricas Parciais do Passo 3*

Configuração	**Desempenho máximo**	**Degradação com falhas**
K=1 e F=1	$T_{1,1} = 152.072$	$D_{1,1} = 2,928\%$
K=2 e F=1	$T_{2,1} = 110.491$	$D_{2,1} = 1,687\%$
K=2 e F=2	$T_{2,2} = 109.781$	$D_{2,2} = 2,319\%$
K=3 e F=1	$T_{3,1} = 92.8194$	$D_{3,1} = 8,910\%$
K=3 e F=2	$T_{3,2} = 86.837$	$D_{3,2} = 14,781\%$
K=3 e F=3	$T_{3,3} = 76.003$	$D_{3,3} = 25,413\%$

Os resultados obtidos demonstram uma degradação de desempenho relativamente pequena, inferior a 3%, quando o K está configurado como 1 e 2. Porém com $K = 3$ a degradação de desempenho foi maior, variando de 8,9% a 25,4%.

A degradação de desempenho ocorrida acontece porque com a existência de falhas menos nós estão disponíveis para conexão pelos clientes e também para o processamento de transações. A variação da degradação conforme a configuração do cenário e quantidade de falhas é inerente à questões internas de implementação do VoltDB. A avaliação desses aspectos é justamente o objetivo atingido com essas métricas.

Para finalizar o passo 3 ainda deve ser gerada a métrica DF_K que sumariza a degradação do cluster configurado com o índice K com diferentes quantidades de falhas. Os valores obtidos para essa métrica parcial, juntamente com uma breve memória de cálculo para demonstração, são os seguintes:

$$DF_1 = \frac{\frac{1}{1} * 2,928}{(\frac{1}{1})} = 2,928$$

$$DF_2 = \frac{(\frac{1}{1} * 1,687) + (\frac{1}{2} * 2,319)}{(\frac{1}{1}) + (\frac{1}{2})} = 1,897$$

$$DF_3 = \frac{(\frac{1}{1} * 8,910) + (\frac{1}{2} * 14,781) + (\frac{1}{3} * 25,413)}{(\frac{1}{1}) + (\frac{1}{2}) + (\frac{1}{3})} = 13,511$$

Essa métrica DF_K é a média ponderada das degradações onde o peso das métricas, que é inversamente proporcional a quantidade de falhas simultâneas, tem como objetivo dar maior ênfase para falhas de menores proporções, que são mais frequentes. Além disso,

essas métricas podem ser comparadas, tanto entre sistemas diferentes como entre si.

Os resultados desse experimento mostram que a degradação de desempenho na ocorrência de falhas é menor quando o cluster está configurado com um índice de tolerância a falhas $K = 2$, ainda que o resultado seja relativamente próximo a configuração $K = 1$. Já com o índice de tolerância $K = 3$, a degradação de desempenho com falhas tem um aumento significativo.

5.3.4 Passo 4

No passo 4 é avaliado o processo de reinserção de nós no cluster após a ocorrência de falhas. Para isso é considerado o tempo entre o comando para entrada de um nó no cluster e a sua completa estabilização, quando o nó está pronto para processar transações. A métrica parcial $E_{K,I}$ indica o tempo de estabilização do i-ésimo nó com o cluster configurado com índice K.

No VoltDB o principal fator que determina o tempo de estabilização de um nó durante sua reinserção é a cópia dos dados entre os nós já existentes para o que está sendo reinserido. Com isso, em um cluster do VoltDB configurado com um índice de tolerância a falhas K, o tempo de estabilização de todos os nós é aproximadamente o mesmo. O que afeta o tempo de estabilização não é a quantidade de falhas F, mas sim o índice K e o tráfego da rede no momento da reinserção.

Foram executados experimentos para obter as métricas parciais dessa etapa do UPB e os resultados são apresentados na tabela 5.5.

Tabela 5.5: *Tempo de reinserção de nós para diferentes configurações do cluster*

Configuração	**Tempo de Estabilização**	**Dados copiados**
$K = 1$	$E_{1,1} = 5,701$ segundos	339 MB
$K = 2$	$E_{2,1} = 6,678$ segundos	510 MB
$K = 2$	$E_{2,2} = 6,746$ segundos	510 MB
$K = 3$	$E_{3,1} = 8,317$ segundos	680 MB
$K = 3$	$E_{3,2} = 8,085$ segundos	680 MB
$K = 3$	$E_{3,3} = 8,160$ segundos	680 MB

Como pode ser observado, no VoltDB, quanto maior o valor do índice de tolerância a

falhas K, maior é a quantidade de dados que devem ser copiados durante a resinserção de nós. Isso acontece porque, como discutido na Seção 2.4, a quantidade total de partições do VoltDB é fixada na configuração do cluster e independe do índice K. Ao aumentarmos o valor de K, mais partições são utilizadas para armazenar cópias dos dados. Consequentemente, uma quantidade maior de dados precisa ser armazenados por sites.

Utilizando os resultados anteriores, as métricas parciais E_K podem ser obtidas da seguinte forma:

$$E_1 = \frac{\frac{1}{1} * 5,701}{(\frac{1}{1})} = 5,701$$

$$E_2 = \frac{(\frac{1}{1} * 6,678) + (\frac{1}{2} * 6,746)}{(\frac{1}{1}) + (\frac{1}{2})} = 6,700$$

$$E_3 = \frac{(\frac{1}{1} * 8,317) + (\frac{1}{2} * 8,085) + (\frac{1}{3} * 8,160)}{(\frac{1}{1}) + (\frac{1}{2}) + (\frac{1}{3})} = 8,225$$

De forma semelhante a métrica DF_K explicada anteriormente, E_K é a média ponderada dos tempos de estabilização onde o peso das métricas, que é inversamente proporcional a quantidade de falhas simultâneas, tem como objetivo dar maior ênfase para reinserções quando o cluster tem menor quantidade de falhas. Além disso, essas métricas também podem ser comparadas, tanto entre sistemas diferentes como entre si.

Os resultados dessa métrica parcial E_K estão de acordo com o esperado pela arquitetura do VoltDB. O tempo de estabilização aumenta conforme o índice de tolerância K.

5.3.5 Métricas Finais

Três métricas finais são propostas pela UPB e foram calculadas com base nos resultados obtidos durante os experimentos:

1. Degradação de desempenho da tolerância a falhas

$$D_T = \frac{34,38 + 52,92 + 57,32}{3} = 48,206$$

2. Degradação de desempenho durante a ocorrência de falhas

$$D_F = \frac{2,928 + 1,897 + 13,511}{3} = 6,112$$

3. Tempo de estabilização para a reinserção de nós

$$E_F = \frac{5,701 + 6,700 + 8,225}{3} = 6,875$$

Os valores gerados por essas métricas finais podem ser utilizadas para a comparação com diferentes SGBDDs.

5.4 Discussão dos Resultados

Nos experimentos realizados extraímos algumas descobertas interessantes além de avaliar a disponibilidade do VoltDB e a viabilidade da UPB.

Os resultados obtidos mostram que os cenários de execução da UPB são relevantes para a avaliação de um SGBDD, pois geram a degradação do desempenho. Esta degradação de desempenho é detectada incrementalmente pelas métricas da UPB precisando quais variáveis tiveram influência no resultado através de um percentual de impacto.

Especificamente, os resultados mostram que o VoltDB apresenta importante perda de desempenho quando se aumenta o nível de tolerância a falha do ambiente, representado pelo índice K. Com K=1 a degradação de desempenho foi de 34%, chegando a aproximadamente 53% e 57% com K igual a 2 e 3, respectivamente. Por outro lado, a ocorrência de falhas apresentou uma degradação relativamente pequena, inferior a 3%, com o K configurado como 1 ou 2. Já com K igual a 3 a perda de desempenho foi mais significativa, variando entre 8 e 25%.

Outro aspecto interessante é que durante o desenvolvimento desse trabalho diversas

versões do VoltDB foram lançadas. Da primeira versão utilizada, v2.2.1, até a mais atual na data em que os resultados desse trabalho foram obtidos, v2.8.4.1, muitas melhorias foram feitas e percebidas. Nas primeiras versões o *K-Safety* apresentava uma grande instabilidade e muitas vezes o cluster inteiro ficava indisponível durante a recuperação de um nó. Nesses versões também foram percebidas instabilidades no desempenho quando o cluster entrava no estado de *backpressure*. Todas essas questões foram corrigidas pelo VoltDB ao longo do tempo, estando resolvidas na versão utilizada nesses experimentos. Isso demonstra que o VoltDB, apesar de ser um produto com versões já consideradas estáveis, sua tecnologia ainda passa por uma fase de amadurecimento.

CAPÍTULO 6

CONCLUSÃO

Nesse trabalho foi apresentado o método de avaliação de disponibilidade de SGBDDs *Under Pressure Benchmark* (UPB). O objetivo do UPB é avaliar os mecanismos de tolerância e recuperação de falhas para a manutenção da disponibilidade de BDDs, quantificando os impactos da sua utilização. Para isso são levados em consideração três aspectos principais: (1) a degradação de desempenho ocorrida quando é utilizado replicação, (2) a degradação de desempenho do cluster durante a ocorrência de falhas e (3) o tempo de recuperação do cluster quando um nó é reinserido após a ocorrência de falhas.

A execução do UPB consiste em quatro passos que simulam diferentes situações (chamadas de cenários) onde os mecanismos de disponibilidade devem ser utilizados. Em cada passo métricas parciais quantificam o desempenho do SGBDD. Os resultados destas métricas são reunídos em métricas finais, que são índices para a classificação de SGBDDs independentes de configurações específicas e que podem ser utilizados para a comparação entre diferentes ambientes e entre SGBDDs de diferentes fabricantes.

Como estudo de caso utilizamos o SGBDD VoltDB [10] que apresenta uma nova arquitetura de processamento de transações. No VoltDB os dados são persistidos em memória principal para permitir o processamento serial das transações. Por ser um SGBDD relativamente novo no mercado, cujos princípios surgiram em 2009 [22] e só foi lançado como produto em 2010, ainda existem relativamente poucas análises sobre o seu desempenho. Até a data de publicação desse trabalho, nenhum outro estudo independente sobre a disponibilidade do VoltDB foi publicado.

Os experimentos, realizados em um cluster do projeto francês Grid5000 [2], demostraram a viabilidade do UPB em avaliar a disponibilidade de SGBDD. Os resultados gerados apontaram a degradação do desempenho do VoltDB em diversas situações onde o mecanismo de replicação e recuperação de falhas foram utilizados.

Além dos mecanismos de recuperação de falhas e replicação de dados, os SGBDD oferecem outros recursos para garantir alta disponibilidade dos seus serviços. Esses recursos variam conforme o fabricante, mas podemos citar como exemplo cópias de segurança dos dados em memória permanente (*e.g.snapshots*, *checkpoints*, *backups*), logs de transações e *redo*, replicação entre diferentes clusters, espelhamento de servidores (*e.g.database mirroring*) e detecção de particionamento de rede. Incluir a avaliação desses recursos no UPB faz parte do trabalho futuro.

O UPB pode ser visto como um esforço inicial na definição de um *benchmark* de disponibilidade padrão para a indústria. Para isso, o trabalho futuro também pode incluir a adaptação do UPB para ser utilizada como uma extensão de *benchmarks* amplamente utilizados, como os TPCs [9].

BIBLIOGRAFIA

[1] Grid'5000 - site lille, https://www.grid5000.fr/mediawiki/index.php/Lille:Hardware. Acessado em 12/2012.

[2] Grid'5000, http://www.grid5000.fr/. Acessado em 12/2012.

[3] Ieee technical committee on dependable computing and fault tolerance, http://www.computer.org/portal/web/tandc/tcft. Acessado em 01/2013.

[4] Ifip working group 10.4 on dependable computing and fault tolerance, http://www.dependability.org/wg10.4/. Acessado em 01/2013.

[5] Measuring the scalability of sql and nosql systems, http://http://www.odbms.org/blog/2011/05/measuring-the-scalability-of-sql-and-nosql-systems/. Acessado em 12/2012.

[6] Oracle nosql database exceeds 1 million mixed ycsb ops/sec, http://blogs.oracle.com/charlesLamb/entry/oracle_nosql_database_exceeds_1/. Acessado em 12/2012.

[7] TPC-A. Transaction Processing Performance Council, http://www.tpc.org/tpca/. Acessado em 06/2012.

[8] TPC-B. Transaction Processing Performance Council, http://www.tpc.org/tpcb/. Acessado em 06/2012.

[9] Transaction Processing Performance Council, http://www.tpc.org/. Acessado em 01/2013.

[10] Voltdb, http://www.voltdb.com/. Acessado em 12/2012.

[11] Yahoo! cloud serving benchmark, http://research.yahoo.com/Web_Information_Management/YCSB. Acessado em 01/2013.

[12] A measure of transaction processing power. *Datamation*, 31:112–118, April de 1985.

[13] Raquel Almeida, Meikel Poess, Raghunath Nambiar, Indira Patil, e Marco Vieira. How to advance tpc benchmarks with dependability aspects. *Proceedings of the Second TPC technology conference on Performance evaluation, measurement and characterization of complex systems*, TPCTC'10, páginas 57–72, Berlin, Heidelberg, 2011. Springer-Verlag.

[14] A. Avizienis, J.C. Laprie, B. Randell, et al. Fundamental concepts of dependability. *Technical Report Series-University Of Newcastle Upon Tyne Computing Science*, 2001.

[15] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, e Iréa Touche. Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, novembro de 2006.

[16] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, e Russell Sears. Benchmarking cloud serving systems with YCSB. *Proc. ACM Symp. on Cloud Computing*, junho de 2010.

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

[18] J. Gray. A census of tandem system availability between 1985 and 1990. *Reliability, IEEE Transactions on*, 39(4):409 –418, oct de 1990.

[19] Jim Gray e Daniel P. Siewiorek. High-availability computer systems. *Computer*, 24(9):39–48, setembro de 1991.

[20] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, e Michael Stonebraker. Oltp through the looking glass, and what we found there. *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, páginas 981–992, New York, NY, USA, 2008. ACM.

[21] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1991.

[22] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, e Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, agosto de 2008.

[23] Karama Kanoun e Lisa Spainhower. *Dependability Benchmarking for Computer Systems*. Wiley-IEEE Computer Society Press, 2008.

[24] Rick Kuhn, Yu Lei, e Raghu Kacker. Practical combinatorial testing: Beyond pairwise. *IT Professional*, 10(3):19–23, maio de 2008.

[25] J. C. Laprie. Dependable computing and fault tolerance: concepts and terminology. *Proceedings of 15th International Symposium on Fault-Tolerant Computing (FTSC-15)*, páginas 2–11, 1985.

[26] Yantao Li e Charles Levine. Extending tpc-e to measure availability in database systems. *Proceedings of the Third TPC Technology conference on Topics in Performance Evaluation, Measurement and Characterization*, TPCTC'11, páginas 111–122, Berlin, Heidelberg, 2012. Springer-Verlag.

[27] Xiangning Liu, Evaggelia Pitoura, e Bharat Bhargava. Adapting distributed database systems for high availability. Relatório técnico, 1995.

[28] James Mauro, Ji Zhu, e Ira Pramanick. The system recovery benchmark. *Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing*, PRDC '04, páginas 271–280, Washington, DC, USA, 2004. IEEE Computer Society.

[29] Sun Microsystems. Sun cluster software - quality by design for advanced availability. Relatório técnico, Santa Clara, CA, USA, 2004.

[30] Barton P. Miller, Louis Fredriksen, e Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, dezembro de 1990.

[31] Cyril U. Orji. A methodology for benchmarking distributed database management systems. *Proceedings of the Seventh International Conference on Data Engineering*, páginas 612–619, Washington, DC, USA, 1991. IEEE Computer Society.

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

[33] Meikel Poess, Raghunath Othayoth Nambiar, Kushagra Vaid, John M. Stephens, Jr., Karl Huppler, e Evan Haines. Energy benchmarks: a detailed analysis. *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, e-Energy '10, páginas 131–140, New York, NY, USA, 2010. ACM.

[34] Ira Pramanick, James Mauro, e Ji Zhu. A system recovery benchmark for clusters. *Cluster Computing, IEEE International Conference on*, 0:387, 2003.

[35] Tilmann Rabl, Sergio Gómez-Villamor, Mohammad Sadoghi, Victor Muntés-Mulero, Hans-Arno Jacobsen, e Serge Mankovskii. Solving big data challenges for enterprise application performance management. *Proceedings of the VLDB Endowment*, 5(12):1724–1735, agosto de 2012.

[36] Transaction Processing Performance Council (TPC). Tpc benchmark c - standard specification revision 5.11. Relatório técnico, janeiro. `http://www.tpc.org/tpcc/spec/tpcc_current.pdf` , Acessado em 01/2013.

[37] Transaction Processing Performance Council (TPC). Tpc benchmark e - standard specification version 1.12.0. Relatório técnico, junho. `http://www.tpc.org/tpce/spec/v1.12.0/TPCE-v1.12.0.pdf` , Acessado em 01/2013.

[38] E. Watanabe (tradutor). Survey on computer security. Relatório técnico, Tokyo, Japan, 1986.

[39] Marco Vieira e Henrique Madeira. A dependability benchmark for oltp application environments. *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '03, páginas 742–753. VLDB Endowment, 2003.

[40] Marco Vieira e Henrique Madeira. From performance to dependability benchmarking: A mandatory path. Raghunath Nambiar e Meikel Poess, editors, *Performance Evaluation and Benchmarking*, volume 5895 of *Lecture Notes in Computer Science*, páginas 67–83. Springer Berlin Heidelberg, 2009.

[41] VoltDB. Performance guide, v2.2.2. Relatório técnico, 2012.

[42] VoltDB. Using voltdb, v2.2.1. Relatório técnico, 2012.

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

[44] Ji Zhu, James Mauro, e Ira Pramanick. Robustness benchmarking for hardware maintenance events. *Proceedings of International Conference on Dependable Systems and Networks*, DSN 2003, páginas 115–122, San Francisco, CA, USA, 2003. IEEE Computer Society Press.