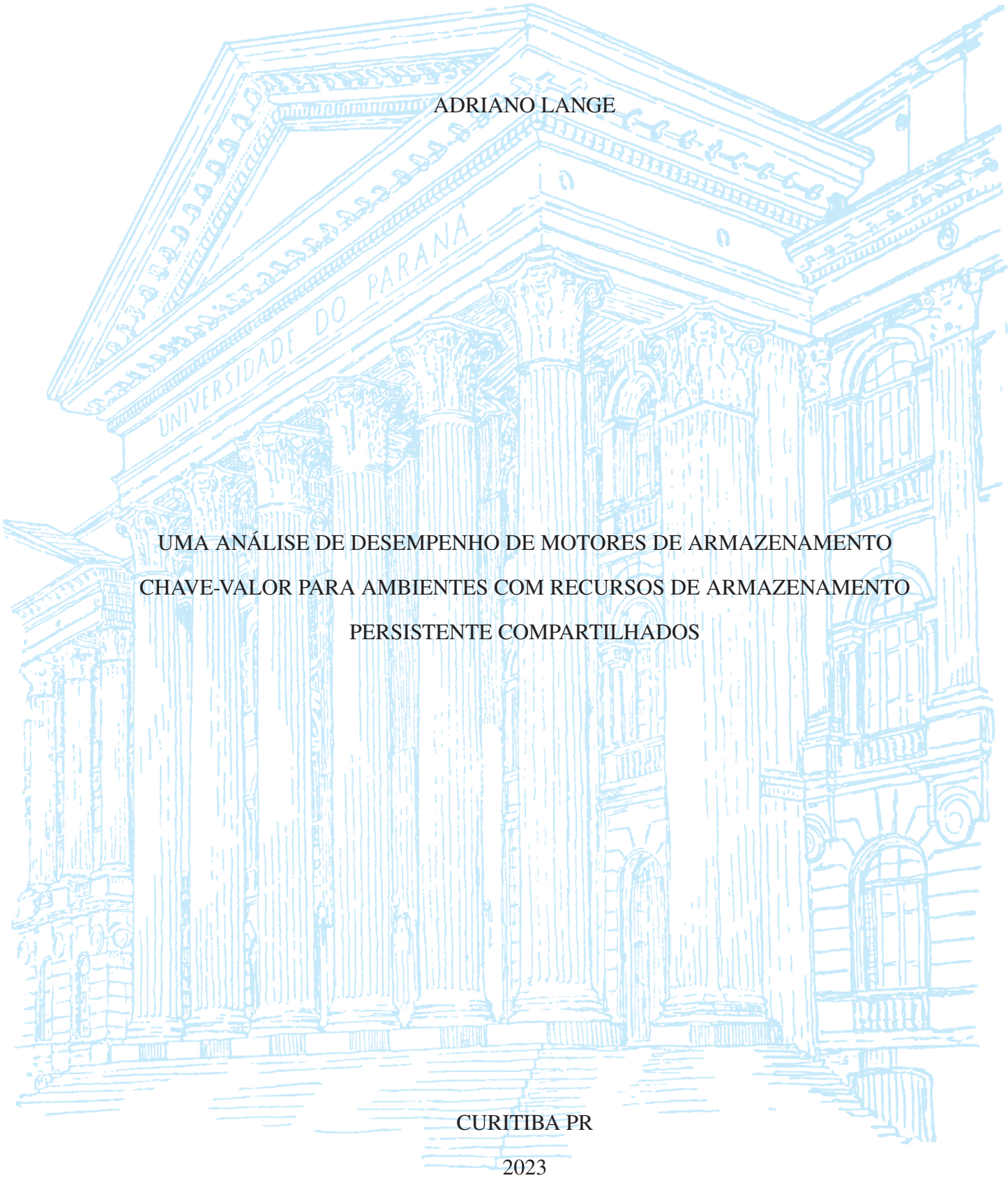
UNIVERSIDADE FEDERAL DO PARANÁ

ADRIANO LANGE



UMA ANÁLISE DE DESEMPENHO DE MOTORES DE ARMAZENAMENTO CHAVE-VALOR PARA AMBIENTES COM RECURSOS DE ARMAZENAMENTO PERSISTENTE COMPARTILHADOS

CURITIBA PR

2023

ADRIANO LANGE

UMA ANÁLISE DE DESEMPENHO DE MOTORES DE ARMAZENAMENTO CHAVE-VALOR PARA AMBIENTES COM RECURSOS DE ARMAZENAMENTO PERSISTENTE COMPARTILHADOS

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Ciência da Computação no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Marcos Sfair Sunye.

CURITIBA PR

2023

DADOS INTERNACIONAIS DE CATALOGAÇÃO NA PUBLICAÇÃO (CIP)
UNIVERSIDADE FEDERAL DO PARANÁ
SISTEMA DE BIBLIOTECAS – BIBLIOTECA DE CIÊNCIA E TECNOLOGIA

Lange, Adriano
Uma análise de desempenho de motores de armazenamento chave-valor para ambientes com recursos de armazenamento persistente compartilhados / Adriano Lange. – Curitiba, 2023.
1 recurso on-line : PDF.

Tese (Doutorado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática.

Orientador: Marcos Sfair Sunye

1. Armazenamento de dados. 2. Memória virtual (Computação). 3. Sistemas de memória de computadores. 4. Memória flash. 5. Motores de armazenamento chave-valor. I. Universidade Federal do Paraná. II. Programa de Pós-Graduação em Informática. III. Sunye, Marcos Sfair. IV. Título.

Bibliotecário: Elias Barbosa da Silva CRB-9/1894

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da tese de Doutorado de **ADRIANO LANGE** intitulada: **UMA ANÁLISE DE DESEMPENHO DE MOTORES DE ARMAZENAMENTO CHAVE-VALOR PARA AMBIENTES COM RECURSOS DE ARMAZENAMENTO PERSISTENTE COMPARTILHADOS**, sob orientação do Prof. Dr. MARCOS SFAIR SUNYE, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de doutor está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 31 de Maio de 2023.

Assinatura Eletrônica
31/05/2023 15:12:31.0
MARCOS SFAIR SUNYE
Presidente da Banca Examinadora

Assinatura Eletrônica
31/05/2023 15:14:48.0
TIAGO RODRIGO KEPE
Avaliador Externo (INSTITUTO FEDERAL DO PARANÁ)

Assinatura Eletrônica
31/05/2023 16:45:01.0
EDUARDO CUNHA DE ALMEIDA
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
02/06/2023 12:31:51.0
TARCIZIO ALEXANDRE BINI
Avaliador Externo (UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ)

To my family.

AGRADECIMENTOS

Agradeço aos meus pais, Arno Lange (*in memoriam*) e Lenir Haeberlin Lange, e à minha esposa, Lídia de Sousa Silva Lange, pelo imenso apoio, compreensão e sacrifício durante esses anos de estudo.

Agradeço enormemente ao meu orientador, Marcos Sfair Sunye, pelo incessante apoio durante o desenvolvimento deste trabalho.

Aos membros da banca examinadora, Tiago Rodrigo Kepe, Tarcízio Alexandre Bini e Eduardo Cunha de Almeida, agradeço pelo tempo dedicado a avaliar esta tese e pelas valiosas sugestões para o enriquecimento da mesma.

Aos professores do C3SL, Luis Carlos Erpen de Bona, Eduardo Cunha de Almeida, Marcos Didonet Del Fabro, Fabiano Silva e Marcos Alexandre Castilho, agradeço pelas preciosas e incontáveis trocas de ideias, dicas e exemplos, seja pessoalmente ou por email.

A Deus, agradeço por todos vocês que me auxiliaram e me apoiaram durante esses anos de estudo.

RESUMO

Os motores de armazenamento chave-valor possuem atualmente um *status* importante em diversos segmentos de tecnologia da informação, tendo sua aplicabilidade comprovada desde sistemas minimalistas de Internet das Coisas e dispositivos móveis até grandes e complexas aplicações científicas e de *Big Data*. Essenciais para a persistência de dados, as tecnologias de armazenamento também obtiveram avanços substanciais ao longo dos últimos anos com a substituição progressiva dos tradicionais discos rígidos por dispositivos baseados em memória flash em grande parte das infraestruturas de computação em nuvem públicas e privadas. Apesar de sua adoção crescente e alto desempenho, os dispositivos baseados em memória flash são muito mais complexos que seus predecessores, empregando técnicas avançadas de mapeamento e organização interna dos dados, cujos detalhes de implementação não são divulgados pela maioria dos fabricantes neste segmento. Tal complexidade e ausência de informações importantes sobre o funcionamento interno desses dispositivos não somente prejudicam a avaliação de desempenho de motores de armazenamento chave-valor, mas também tornam esta tarefa ainda mais complexa em cenários onde este tipo de recurso de armazenamento é compartilhado por outras cargas de trabalho. Uma vez que tais recursos são cada vez mais compartilhados em ambientes de computação em nuvem, a avaliação de motores de armazenamento chave-valor em tais condições é tão desafiadora quanto necessária. Este estudo propõe o Storiks, um framework desenvolvido para avaliar como o desempe-nho de um motor de armazenamento chave-valor é afetado por cargas de trabalho concorrentes compartilhando um mesmo dispositivo baseado em memória flash. A partir deste framework, este estudo avaliou experimentalmente 840 combinações de cargas de trabalho, dispositivos de armazenamento e versões de sistema operacional. Os resultados obtidos por esses experimentos demonstram que esta interferência de desempenho pode assumir padrões distintos de acordo com cada um desses fatores, variando desde patamares próximos a nenhuma interferência até condições altamente degradantes, onde o desempenho do motor de armazenamento chave-valor é reduzido em mais de 90%.

Palavras-chave: interferência de desempenho, motores de armazenamento chave-valor, dispositivos de armazenamento em memória flash.

ABSTRACT

Key-value stores have today an important status in numerous information technology segments, ranging from the minimalist world of mobile devices and Internet of Things to highly complex scientific applications and Big D ata. Essential to persist data, storage technology has also shown substantial improvements over the recent years with the progressive replacement of old hard disk drives by flash-based storage devices in most private and public cloud computing infrastructures. Despite the increased adoption and superior performance, flash-based storage devices usually comprise complex hardware and firmware logic with many details not revealed by their respective manufacturers. This complexity and lack of more information about their internal behavior not only hinder a proper performance evaluation of key-value stores but also turn this task even more complex in scenarios where such resources are shared with other co-located workloads. Once sharing resources is a concept increasingly used in cloud environments, evaluating key-value stores in such a context is challenging but necessary. This study proposes Storiks, a framework designed to assess how the key-value store's performance is affected by concurrent workloads when sharing the same flash-based storage device. Using this framework, we experimentally evaluated 840 combinations of different workloads, storage devices, and operating system versions. We demonstrate from these results that such interference may assume distinct patterns according to each of these factors, ranging from approximately no interference to highly degradation conditions, where the key-value store's performance is reduced by more than 90%.

Keywords: performance interference, key-value stores, flash devices.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

DINF	Departamento de Informática
PPGINF	Programa de Pós-Graduação em Informática
UFPR	Universidade Federal do Paraná
ABI	*Application Binary Interface*
API	*Application Programming Interface*
APST	*Autonomous Power State Transition*
CDF	*Cumulative Distribution Function*
CPU	*Central Processing Unit*
DRAM	*Dynamic Random Access Memory*
E/S	Entrada/Saída
ext4	*Fourth Extended Filesystem*
FTL	*Flash Translation Layer*
GB	*Gigabyte* (10^9 *bytes*)
GiB	*Gibibyte* (2^{30} *bytes*)
HDD	*Hard Disk Drive*
IaaS	*Infrastructure as a Service*
IoT	*Internet of Things*
KB	*Kilobyte* (10^3 *bytes*)
KiB	*Kibibyte* (2^{10} *bytes*)
LBA	*Logical Block Addresses*
LSM-Tree	*Log-Structured Merge Tree*
LVM	*Logical Volume Management*
MB	*Megabyte* (10^6 *bytes*)
MiB	*Mebibyte* (2^{20} *bytes*)
MV	Máquina Virtual
NOIOB	*Namespace Optimal I/O Boundary*
NoSQL	*Not Only SQL*
NOWS	*Namespace Optimal Write Size*
NPWA	*Namespace Preferred Write Alignment*
NPWG	*Namespace Preferred Write Granularity*
NVM	*Non-Volatile Memory*
NVMe	*Non-Volatile Memory Express*
OP	*Over-Provisioning*
PCIe	*PCI-Express*
PCI	*Peripheral Component Interconnect*

POSIX	*Portable Operating System Interface*
QoS	*Quality of Service*
RAID	*Redundant Array of Independent Disks*
RAM	*Random Access Memory*
SATA	*Serial AT Attachment*
SGBD	Sistema de Gerenciamento de Banco de Dados
SO	Sistema Operacional
SPEC	*Standard Performance Evaluation Corporation*
SQL	*Structured Query Language*
SSD	*Solid-State Drive*
SST	*Sorted Sequence Table*
TLC	*Triple-Level Cell*
URL	*Uniform Resource Locator*
VFS	*Virtual File System*
WAL	*Write-Ahead Log*
YCSB	*Yahoo! Cloud Serving Benchmark*

LISTA DE SÍMBOLOS

ρ	função de pressão
$\overline{\rho}$	função de pressão normalizada
C	conjunto de valores de desempenho
D	conjunto de cargas de trabalho concorrentes
$\preceq_C$	relação de ordem de desempenho
$\succeq_D$	relação de ordem de pressão
tx/s	transações por segundo

SUMÁRIO

1 INTRODUÇÃO

Ao longo dos últimos anos, os bancos de dados chave-valor têm sido largamente empregados em uma grande variedade de aplicações, abrangendo desde sistemas minimalistas e embarcados de Internet das Coisas (*Internet of Things*, ou IoT), mecanismos de *cache*, jogos *online* e mídias sociais, até grandes e complexas aplicações científicas e de *Big Data*. Compostos apenas por pares de dados <chave, valor>, tais bancos de dados admitem uma interface de consultas muito simples, sendo principalmente:

- *Get*, para a leitura ou busca do valor de uma chave;
- *Put*, para a inserção ou substituição do valor de uma chave; e
- *Delete*, para a exclusão de uma chave do banco de dados.

Devido à sua simplicidade de representação de dados e de consulta, o modelo de bancos de dados chave-valor é uma alternativa muito mais flexível e escalável que o modelo relacional tradicional. Por esta razão, diversos Sistemas Gerenciadores de Bancos de Dados (SGBDs) NoSQL adotam este modelo como interface de consulta ao usuário. Mesmo no caso em que a interface de consulta fornecida por esses sistemas seja mais complexa, o modelo chave-valor é empregado tanto em mecanismos internos de armazenamento persistente dos dados como em mecanismos auxiliares, como *cache* e indexação. São exemplos relevantes de SGBDs NoSQL que utilizam internamente este tipo de representação como forma de armazenamento de dados o BigTable [20], HBase [1], MongoDB [68], Cassandra [2], ZippyDB [42] e AsterixDB [10]. Além destes, SGBDs relacionais, como MyRocks [35] e CockroachDB [22] também utilizam o modelo chave-valor como forma de armazenamento interno.

Do ponto de vista arquitetural, os motores de armazenamento chave-valor são componentes de *software* especializados em armazenar e gerenciar bancos de dados do tipo chave-valor. Devido à sua alta aplicabilidade e reusabilidade, muitos desses motores vêm sendo desenvolvidos como projetos separados, na forma de bibliotecas ou serviços, e posteriormente incorporados a sistemas maiores. São exemplos relevantes deste tipo de motor o RocksDB [43], desenvolvido pela Meta/Facebook e utilizado pelo ZippyDB e MyRocks, o WiredTiger [69], utilizado pelo MongoDB, o LevelDB [44], desenvolvido pela Google e utilizado como base de implementação para o RocksDB, e o Pebble [23], o qual é inspirado no LevelDB e RocksDB e atualmente substitui o RocksDB como parte do projeto do CockroachDB [85].

Em sua maioria, os motores de armazenamento chave-valor são orientados a disco (*disk-oriented*). Ou seja, utilizam dispositivos de armazenamento persistente como local principal de depósito dos dados, mantendo apenas uma fração destes em memória RAM (*Random Access Memory*) na forma de *cache* ou *buffer* e utilizando *logs* de transação para garantir a durabilidade. Dentre as tecnologias de armazenamento persistente disponíveis atualmente, os dispositivos baseados em memória flash têm se tornado uma opção bastante relevante, especialmente por seu alto desempenho em relação aos tradicionais discos rígidos (*Hard Disk Drives*, ou HDDs) e pela redução gradual do preço por GB ao longo dos últimos anos. Por este motivo, boa parte dos motores de armazenamento chave-valor atuais considera esta tecnologia de armazenamento em seus respectivos valores de referência para parâmetros de configuração ou como parte de suas decisões de projeto.

Apesar de apresentarem desempenho superior, os dispositivos de armazenamento baseados em memória flash são muito mais complexos que os tradicionais HDDs. Devido às

características intrínsecas deste tipo de memória, especialmente a necessidade de apagamento dos dados contidos nas células flash antes que estas possam receber novas gravações, os dispositivos que empregam tal tecnologia são equipados com um *software* embarcado, denominado *Flash Translation Layer* (FTL), o qual visa amortizar tais restrições e fornecer um mapeamento dinâmico e transparente entre o espaço de endereçamento de dados visível ao sistema operacional e a forma com que esses dados são realmente armazenados internamente. Contudo, muitos dos detalhes de implementação deste mapeamento não são divulgados pelos fabricantes, dificultando a análise e a previsibilidade de desempenho de aplicações que utilizam tais dispositivos. Além disso, este problema se torna ainda mais complexo em contextos onde este tipo de recurso é compartilhado por múltiplas aplicações, uma prática muito comum em ambientes de computação em nuvem.

A computação em nuvem é atualmente um dos mais bem sucedidos paradigmas adotados para a contratação e alocação de serviços e recursos computacionais sob demanda. Utilizando diversas técnicas de virtualização e gerenciamento de recursos, grande parte dos provedores de nuvens públicas e privadas busca reduzir os custos de aquisição e operação de infraestrutura por meio da consolidação de um grande número de clientes (inquilinos) ou serviços em um mesmo *hardware*. Esta prática é comumente conhecida como *multi-tenancy* ou multi-inquilinos.

Apesar de vantajosa em termos econômicos, a prática de *multi-tenancy* também pode proporcionar efeitos colaterais indesejados, especialmente a degradação de desempenho causada por "vizinhos barulhentos" (*noisy neighbors*) acessando concorrentemente os mesmos recursos computacionais [56]. Como forma de atenuar este problema de interferência de desempenho, muitos provedores de nuvem buscam priorizar e atribuir limites de utilização de recursos para cada inquilino por meio de mecanismos de escalonamento, ao mesmo tempo em que tentam separar inquilinos com maior potencial de interferência de desempenho em diferentes conjuntos de *hardware*, seja pelo secionamento deste dentro de um mesmo servidor (ex.: CPUs, memória, dispositivos de armazenamento e rede), seja pela migração de inquilinos entre servidores [21, 65, 97, 95, 54].

A prática de *multi-tenancy* e a necessidade de isolamento de desempenho entre inquilinos revela o caráter conflitante entre a reserva e o compartilhamento de recursos, ilustrado na Figura 1.1. Por um lado, o isolamento de desempenho remete ao fato que os recursos computacionais recebidos por um inquilino não são afetados pelo consumo de qualquer outro, produzindo assim a ilusão de um ambiente dedicado e caracterizando uma espécie de reserva de recursos. Embora esta reserva possa representar uma maior previsibilidade de desempenho, na maior parte dos casos ela também reduz a utilização desses recursos e aumenta os custos com infraestrutura. Por outro lado, o compartilhamento admite que os recursos recebidos por um inquilino variem de acordo com a demanda dos demais inquilinos. Esta condição geralmente permite aumentar a utilização desses recursos e reduzir os custos com infraestrutura. Contudo, ela também aumenta a variação e reduz a previsibilidade de desempenho observadas pelos inquilinos.

Figura 1.1: Conflitos entre reserva e compartilhamento de recursos.

Semelhante a outros sistemas de armazenamento de bancos de dados orientados a disco, boa parte dos motores de armazenamento chave-valor é altamente sensível ao desempenho do dispositivo de armazenamento persistente. Por este motivo, uma prática comum durante a implantação de motores de armazenamento em ambientes de computação em nuvem é justamente

evitar a interferência de desempenho por meio do uso de dispositivos de armazenamento dedicados ou por meio da locação de recursos de armazenamento com prioridades elevadas de desempenho [11, 46, 67, 66]. Tal prática tende a explorar regiões mais à esquerda da escala apresentada na Figura 1.1, o que consequentemente remete a um custo mais elevado de infraestrutura para este tipo de serviço.

Devido à relutância ao uso compartilhado de dispositivos de armazenamento persistente em diversos ambientes de produção, a interferência de desempenho sofrida por motores de armazenamento chave-valor é um campo insuficientemente explorado pelo atual estado da arte, deixando de considerar questões mais à direita da Figura 1.1. Embora ferramentas de *benchmark* representativas, como o *Yahoo! Cloud Serving Benchmark* (YCSB) [25, 24] e db_bench [43, 19], tenham sido desenvolvidas e amplamente utilizadas para avaliar o desempenho de SGBDs NoSQL e motores de armazenamento chave-valor, tais opções não avaliam a interferência de desempenho causada por cargas de trabalho concorrentes. Em ambientes compartilhados, o desempenho de um motor de armazenamento chave-valor precisa ser medido em relação a outras cargas de trabalho, algo que não faz parte do projeto dessas ferramentas.

Ao utilizarem metodologias de teste baseadas em ambientes de recursos computacionais não compartilhados, muitos dos trabalhos recentes dedicados a analisar o desempenho e as estruturas de dados utilizadas por motores de armazenamento chave-valor desconsideram o efeito de "vizinhos barulhentos" em suas avaliações. Por exemplo, os estudos apresentados por Athanassoulis et al. [15], por Dayan et al. [27, 28, 29, 30] e por Idreos et al. [50, 51, 49, 48] tem sido bastante relevantes no sentido de propor estruturas de dados customizadas para cada tipo de carga de trabalho. Outros trabalhos, como os apresentados por Luo e Carey [63], Lomet [62] e Yoon et al. [98], também consideram questões internas dessas estruturas, tais como o escalonamento de mecanismos de mesclagem de dados, *cache* e distribuição de dados em dispositivos de armazenamento diferentes. Contudo, uma vez que a interferência de desempenho não é considerada como parte desta modelagem, tais contribuições tendem, de certo modo, a retroalimentar a demanda por isolamento de desempenho.

Embora outras ferramentas e *frameworks* de *benchmark* tenham sido propostas para avaliar ambientes com múltiplos inquilinos, tais como o SPEC Cloud IaaS [84], Mowgli [81], e BigDataBench [94], tais opções não consideram as características intrínsecas tanto do dispositivo de armazenamento persistente como dos motores de armazenamento chave-valor. Mesmo que essas ferramentas incluam diferentes cargas de trabalho chave-valor em suas implementações, a abordagem adotada por elas ignora as condições internas de dispositivos de armazenamento baseados em memória flash e sua relação com as estruturas internas do motor de armazenamento. Conforme descrito por Didona et al. [32], a observância de tais condições é essencial para a reprodutibilidade das avaliações de desempenho, sendo ainda mais crítica em ambientes de recursos compartilhados.

1.1 PERGUNTA DE PESQUISA E HIPÓTESE

Devido à falta de análises mais detalhadas sobre o emprego de motores de armazenamento chave-valor em ambientes de recursos compartilhados e a complexidade intrínseca dos dispositivos de armazenamento baseados em memória flash, este estudo busca avaliar a seguinte pergunta de pesquisa:

Q1 Qual o impacto de cargas de trabalho concorrentes sobre o desempenho de um motor de armazenamento chave-valor quando ambos compartilham um mesmo dispositivo de armazenamento baseado em memória flash?

A hipótese avaliada neste caso é que a resposta para tal pergunta de pesquisa não pode ser simplesmente obtida sem a consideração de uma série de fatores, em especial:

- As cargas de trabalho envolvidas, tanto as submetidas ao motor de armazenamento chave-valor como as cargas de trabalho concorrentes;
- O modelo do dispositivo de armazenamento avaliado; e
- A implementação do sistema operacional que gerencia o ambiente compartilhado.

1.2 CONTRIBUIÇÃO

Para responder a pergunta de pesquisa e avaliar a hipótese apresentada na Seção 1.1, este estudo propõe o **Storiks**, um *framework* flexível projetado para avaliar a interferência de desempenho sofrida por um motor de armazenamento chave-valor [59]. Este *framework* utiliza experimentos como abordagem principal de avaliação de desempenho, através dos quais é possível combinar diversas cargas de trabalho submetidas ao motor de armazenamento com uma grande variedade de padrões distintos de cargas de trabalho intensivas em E/S, exercitando assim aspectos intrínsecos da tecnologia utilizada pelo dispositivo de armazenamento e do ambiente computacional compartilhado. Tais aspectos não são considerados simultaneamente por outras ferramentas de *benchmark* que utilizam experimentos para a avaliação de desempenho, tais como YCSB, db_bench, SPEC Cloud IaaS, Mowgli e BigDataBench.

A abordagem utilizada neste estudo também difere de propostas como Amber* [47] e MQSIM [86], as quais utilizam simulações para modelar o desempenho de dispositivos de armazenamento baseados em memória flash. Embora tais simulações sejam utilizadas para avaliar diferentes componentes e implementações internas deste tipo de dispositivo, sua aplicabilidade em relação a ambientes reais de produção é mais limitada e complexa, especialmente devido à ausência de informações fornecidas pelos fabricantes de dispositivos baseados em memória flash disponíveis no mercado e por não contemplarem toda a pilha de *software* envolvida, incluindo o sistema operacional e o sistema de arquivos. Tais restrições prejudicam a análise da hipótese apresentada na seção anterior.

Tanto o Storiks como a metodologia de avaliação desenvolvida neste estudo levam em consideração diversos requisitos importantes, como fácil utilização, portabilidade, reprodutibilidade, autonomia de execução dos experimentos, significância dos componentes avaliados e extensibilidade dos experimentos e do ambiente de análise de resultados. Definida como *pressão*, este *framework* permite relacionar o desempenho obtido por uma carga de trabalho submetida ao motor de armazenamento com cada tipo de carga de trabalho concorrente especificada pelo usuário por meio de diversos parâmetros de configuração. Além de medir o desempenho do motor de armazenamento, o Storiks é capaz de coletar uma grande variedade de contadores e informações estatísticas provenientes deste motor, do sistema operacional e do *hardware*. Tais informações podem ser posteriormente analisadas pelo usuário por meio de diversas estruturas de dados e gráficos disponíveis pelo *framework* em um ambiente Jupyter Notebook interno [53].

Para avaliar a eficácia do *framework* Storiks, foram realizados diversos experimentos compostos por duas cargas de trabalho submetidas ao motor de armazenamento chave-valor, setenta cargas de trabalho concorrentes intensivas em E/S, três dispositivos de armazenamento flash e duas versões do kernel Linux. Os resultados obtidos por esses experimentos demonstram que a interferência de desempenho pode assumir padrões distintos de acordo com cada um desses fatores, variando desde valores próximos a zero (nenhuma interferência) até condições altamente

degradantes, onde o desempenho do motor de armazenamento chave-valor é reduzido em mais de 90%.

Tanto o desenvolvimento do *framework* Storiks como as análises apresentadas neste estudo têm por foco principal a aplicabilidade de motores de armazenamento chave-valor em ambientes de recursos compartilhados. Em especial, regiões mais à direita da Figura 1.1, onde os custos com infraestrutura são mais baixos. Além de permitir explorar e identificar quais cenários compartilhados exercem maior ou menor influência sobre o desempenho de um motor de armazenamento chave-valor, este estudo visa ainda fomentar futuros projetos de motores de armazenamento mais resilientes às variações de desempenho causadas por cargas de trabalho concorrentes.

1.3 ORGANIZAÇÃO

O restante desta tese está organizado da seguinte forma: O Capítulo 2 apresenta uma visão geral sobre os dispositivos de armazenamento baseados em memória flash e os motores de armazenamento chave-valor, além dos trabalhos relacionados com a avaliação de desempenho utilizando tais tecnologias. O Capítulo 3 apresenta a arquitetura e os componentes principais do *framework* Storiks, bem como as principais definições e requisitos utilizados para sua implementação. O Capítulo 4 avalia a pergunta de pesquisa e a hipótese apresentadas neste estudo por meio de uma série de experimentos realizados a partir deste *framework*. Por fim, o Capítulo 5 conclui esta tese, apresentando ainda algumas perspectivas relevantes de trabalhos futuros.

2 CONCEITOS BÁSICOS E TRABALHOS RELACIONADOS

Este capítulo detalha os principais conceitos e referências bibliográficas utilizadas para fundamentar o desenvolvimento deste estudo. Primeiramente, a Seção 2.1 descreve a arquitetura básica e o funcionamento dos dispositivos de armazenamento baseados em memória flash. Já a Seção 2.2 apresenta o funcionamento básico de motores de armazenamento chave-valor, incluindo suas principais estruturas de armazenamento e uma implementação de motor bastante relevante em ambientes de produção. Em seguida, as Seções 2.3, 2.4 e 2.5 apresentam os trabalhos relacionados com a simulação de dispositivos baseados em memória flash, com o *benchmark* de motores de armazenamento chave-valor e infraestruturas de nuvem com múltiplos inquilinos e com a interferência de desempenho em ambientes de recursos compartilhados. Por fim, a Seção 2.6 conclui este capítulo fazendo um breve resumo dos trabalhos relacionados.

2.1 DISPOSITIVOS DE ARMAZENAMENTO FLASH

A memória flash é um tipo de memória não volátil presente na maioria dos dispositivos de armazenamento de estado sólido (*Solid State Drives*, ou SSDs) comercializados atualmente. Tais dispositivos vêm sendo largamente empregados nos últimos anos como um substituto aos antigos modelos de armazenamento baseados em disco rígido (*Hard Disk Drives*, ou HDDs). Conforme ilustrado na Figura 2.1, a arquitetura típica de um SSD baseado em memória flash é composta por duas partes principais: uma parte computacional (*computation complex*) e outra de armazenamento persistente (*storage complex*).

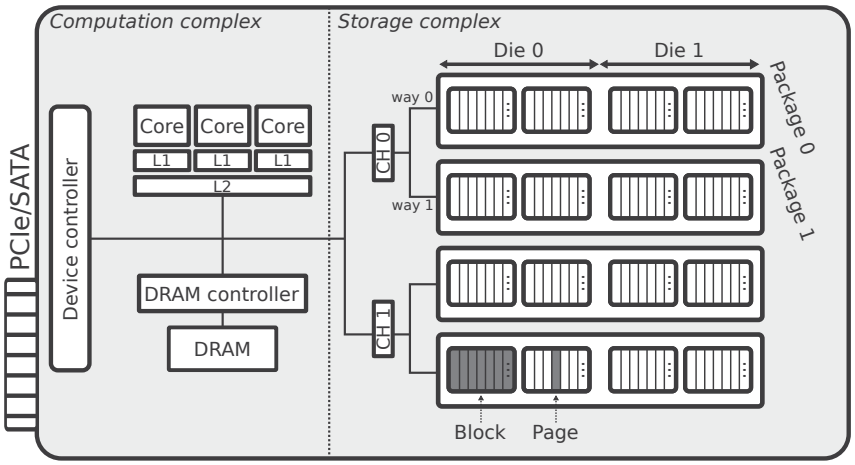


Figura 2.1: Arquitetura interna de um dispositivo de armazenamento baseado em memória flash.

A parte de armazenamento corresponde à memória flash em si, a qual serve para persistir todos os dados gravados no dispositivo. Existem dois tipos principais de memória flash: NAND e NOR. Para evitar possíveis ambiguidades, este estudo tem por foco principal o tipo de memória NAND, comumente utilizada na fabricação de SSDs devido ao seu menor custo, maior capacidade de armazenamento e maior desempenho de gravação em relação à memória do tipo NOR.

Fisicamente, a memória do tipo NAND é composta por um grande número de *células flash* capazes de armazenar cargas elétricas por longos períodos de tempo sem a necessidade atualizações periódicas como as memórias RAM. Tais células flash são agrupadas em *páginas*

(*pages*), as quais são acionadas eletricamente durante as operações de leitura ou gravação, formando assim a menor unidade para esses dois tipos de operações. Dependendo da fabricação, o tamanho das páginas flash pode variar geralmente entre algumas centenas de bytes até dezenas de KiB. Este tamanho costuma não ser explicitamente informado pelos fabricantes de SSDs em suas respectivas especificações técnicas.

Além de ser estruturada em páginas para a realização de operações de leitura e gravação, uma característica importante da memória flash é que cada página necessita ser apagada antes de receber uma nova gravação (ou programação). Uma vez que o acionamento elétrico deste processo de apagamento requer uma latência adicional, este custo é amortizado por meio de uma abordagem de gravação *out-of-place*, ou seja, as atualizações de dados são armazenadas em outras páginas flash que já foram apagadas anteriormente. Já o apagamento de páginas com dados antigos é feito posteriormente pelo dispositivo de armazenamento por meio de um mecanismo de coleta de lixo (*garbage collection*, ou GC). Eletricamente, a unidade mínima para o acionamento de uma operação de apagamento é de um *bloco* (*block*). Dependendo do modelo do dispositivo, este tamanho de bloco pode variar entre poucas a várias centenas de páginas flash.

Em seu processo de fabricação, os blocos de página flash são agrupados em *planes*, os quais são agregados em *dies* e finalmente empacotados em *chips* (*packages)* para então serem soldados à placa do dispositivo de armazenamento. Cada dispositivo pode conter um ou mais desses *chips* conectados à parte computacional por meio de canais (*channels*) e *ways*. Para melhorar o desempenho e diminuir as latências de leitura e gravação, a maioria dos dispositivos flash atuais implementa mecanismos de paralelismo interno, permitindo controlar múltiplos *planes* ou *dies* simultaneamente através dos diferentes canais ou *ways* disponíveis.

Para viabilizar este processo *out-of-place* de gravação de dados, os dispositivos de armazenamento flash implementam um mecanismo de indireção para indicar qual página flash contém os dados mais recentes gravados pelo usuário. Além disso, este mecanismo precisa controlar ainda quais páginas ainda estão livres para receber novas programações e quais delas podem ser eventualmente apagadas pela coleta de lixo. Todo este processo é controlado por um *software* embarcado em execução na parte computacional do dispositivo, chamado *Flash Translation Layer* (FTL). A FTL é responsável pelo mapeamento entre os endereços de blocos lógicos (LBAs) endereçáveis pelo sistema operacional e as páginas flash internas[1].

A parte computacional de um SSD corresponde ao *hardware* responsável pelo processamento da FTL e pela interface de comunicação com o restante do sistema. Fisicamente, esta parte é composta por um controlador do dispositivo (*device controller*), o qual implementa o protocolo de comunicação utilizado pela interface (ex., SATA ou PCIe), por um ou mais núcleos de CPU para a execução de toda a parte lógica embarcada, além de um ou mais módulos de memória DRAM (incluindo controladores) para o armazenamento temporário dos dados lidos e gravados na parte de armazenamento e da tabela de tradução de blocos lógicos para páginas flash utilizada pela FTL.

Toda essa lógica embarcada para o processamento da FTL é significativamente mais complexa que os tradicionais HDDs, principalmente devido à necessidade de apagamento das células flash antes de qualquer gravação e à consequente adoção de uma estratégia de atualização de dados *out-of-place*. Sempre que o sistema operacional requisita a reescrita de dados no dispositivo, referentes a um ou mais LBAs, a FTL precisa utilizar outra(s) página(s) flash para armazenar esses dados novos e atualizar a tabela de mapeamento para os respectivos LBAs. Ao final deste processo, as páginas flash que armazenam dados antigos, não possuindo mais LBAs associados a elas, são marcadas como invalidadas.

[1]Mais detalhes sobre as diferentes formas de organização e endereçamento de dados entre dispositivo de armazenamento, sistema operacional e aplicação serão apresentados no capítulo seguinte.

À medida que o número de páginas flash invalidadas aumenta e o número de páginas livres para receber novas gravações diminui, o dispositivo flash precisa realizar procedimentos periódicos de coleta de lixo. Uma vez que o processo de apagamento de páginas é feito somente em nível de bloco, a coleta de lixo deve preferencialmente selecionar blocos com grandes quantidades de páginas invalidadas. Neste processo, caso o bloco selecionado ainda possua páginas flash com dados válidos (associados a LBAs), tais páginas precisam ser copiadas para outros blocos, além de terem seus respectivos LBAs atualizados na tabela de mapeamento da FTL.

Outro aspecto essencial que precisa ser considerado nesta dinâmica de gravações de páginas e coleta de lixo é o número limitado de apagamentos de dados que cada bloco pode suportar durante sua vida útil (*wear-out*). Como forma de contornar este problema, a maioria dos fabricantes de dispositivos flash implementa diversas otimizações pertinentes à seleção de páginas para gravação e coleta de lixo com o intuito de distribuir de maneira uniforme o número de apagamentos entre os blocos do dispositivo (*wear-leveling*).

Apesar de serem mais complexos, os dispositivos de armazenamento baseados em memória flash tendem a superar em desempenho os tradicionais HDDs. Por não haverem componentes mecânicos, esta tecnologia mais recente elimina as conhecidas latências de posicionamento e rotação presentes nos HDDs, tornando o desempenho dos SSDs menos suscetíveis aos acessos aleatórios de uma carga de trabalho. Embora os valores de desempenho divulgados pelos fabricantes de SSDs ainda apontem uma vantagem maior de operações sequenciais sobre as operações aleatórias (ex. [80, 79, 78]), tais diferenças de desempenho estão relacionadas a outras características, como o tipo de operação (leitura ou gravação), o nível de paralelismo interno proporcionado pela parte de armazenamento, além de sua capacidade de processamento, memória e lógica de controle implementadas na parte computacional do dispositivo. Algumas dessas informações não são detalhadas pelo fabricante, o que torna importante o teste de desempenho de cada dispositivo em diferentes cargas de trabalho.

Embora os dispositivos de armazenamento flash apresentem diversas vantagens, os HDDs ainda possuem um custo menor por GB, de modo que a aplicação desta tecnologia mais antiga está sendo gradativamente direcionada para funções de arquivamento digital de grandes volumes de dados com baixa frequência de leituras enquanto que os SSDs estão assumindo cargas de trabalho com leituras e gravações mais frequentes e de menor latência. Devido a essas características de desempenho, uma parte significativa dos sistemas gerenciadores de bancos de dados atualmente em produção utiliza dispositivos SSDs como recurso principal de armazenamento persistente.

2.2 MOTORES DE ARMAZENAMENTO CHAVE-VALOR

Os motores de armazenamento chave-valor [48] tem sido largamente utilizados por diversas aplicações atuais, abrangendo desde sistemas minimalistas e embarcados de Internet das Coisas (*Internet of Things*, ou IoT) até grandes e complexas aplicações científicas e de redes sociais. Como o próprio nome sugere, tais motores de armazenamento são projetados para gerenciar um ou mais bancos de dados compostos por pares <chave, valor>. A chave precisa ser única em cada banco de dados, a qual é utilizada como referência para três consultas ou operações básicas:

- *Get* (ou *point lookup*), a qual recupera o valor de uma chave;
- *Put*, a qual insere ou atualiza o valor de uma chave; e

- *Delete*, a qual remove uma chave e seu respectivo valor do banco de dados.

Além destas, outras operações são eventualmente implementadas por alguns motores de armazenamento chave-valor, especialmente as consultas por intervalo de chaves (*range queries*). Devido à simplicidade deste modelo de banco de dados e o uso de consultas bem menos complexas que a SQL tradicional, a implementação deste tipo de motor é muito mais simples que os SGBDs relacionais, podendo alcançar inclusive taxas maiores de desempenho.

Dependendo da forma com que um motor de armazenamento é projetado, o mesmo pode ser facilmente incorporado a projetos maiores, inclusive como camada de armazenamento para SGBDs relacionais e NoSQL. Por exemplo, o motor de armazenamento RocksDB [43] é utilizado pelo ZippyDB [42], um SGBD NoSQL geograficamente distribuído desenvolvido pela Meta/Facebook, e pelo MyRocks [35], uma versão do SGBD relacional MySQL. O WiredTiger, por sua vez, é utilizado como motor de armazenamento no SGBD NoSQL MongoDB [69].

2.2.1 Árvores LSM

Do ponto de vista de estrutura de dados, as árvores LSM (*Log-Structured Merge Trees*, ou *LSM-Trees*) são largamente utilizadas para o armazenamento persistente de banco de dados chave-valor [73, 64]. Alguns exemplos relevantes de projetos que usam este tipo de estrutura são o BigTable [20], HBase [1], Cassandra [2], LevelDB [44], RocksDB [43] e AsterixDB [10]. Por este motivo, este estudo terá um foco maior sobre as árvores LSM e sua implementação específica dentro do motor de armazenamento RocksDB, o qual é utilizado pelo *framework* Storiks. Maiores detalhes sobre este tipo de estrutura de dados podem ser encontrados no trabalho apresentado por Luo e Carey [64].

De um modo geral, estruturas de dados otimizadas para operações de leitura tendem a divergir de estruturas otimizadas para gravações [50, 29]. Por exemplo, vetores ordenados e árvores de busca são bastante eficientes para operações de leitura, mas exigem um certo custo de manutenção quando são alteradas. Por outro lado, arquivos de *log* sequenciais são bastante eficientes para gravações, mas ineficientes para operações de busca. As árvores LSM exploram alternativas intermediárias deste espaço de possíveis implementações de estruturas de dados, buscando proporcionar um desempenho equilibrado tanto de leituras como de gravações.

Uma vez que tanto os HDDs como os SSDs são dispositivos de bloco, cuja granularidade mínima de leitura e gravação varia tipicamente entre centenas de bytes e algumas dezenas de KiB, as árvores LSM utilizam uma estrutura logarítmica de armazenamento *out-of-place* para atualizar os pares de dados <chave, valor> e reduzir a amplificação de escrita. Em sua forma geral, este tipo de estrutura organiza os dados em vários níveis hierárquicos ($L_0, \ldots, L_N$). Cada nível possui um ou mais componentes para o armazenamento desses dados. O nível L_0 corresponde ao nível de menor capacidade e armazena os dados mais recentes, enquanto que o nível L_N corresponde ao nível de maior capacidade e contém os dados mais antigos.

De acordo com cada implementação, o nível L_0 da árvore LSM pode ou não residir em memória e tende a apresentar uma maior otimização de escrita, enquanto que os demais níveis residem em armazenamento persistente e tendem a ser mais otimizados para operações de leitura. Diferente de outras estruturas que atualizam os dados diretamente onde estes estão armazenados, as atualizações em uma árvore LSM geralmente produzem novos registros no nível de menor capacidade, os quais são posteriormente consolidados nos níveis subsequentes por operações sucessivas de mesclagem (*merge* ou *compaction*) [64]. As operações de exclusão (*delete*) também obedecem esta regra, as quais são registradas por meio de um registro especial chamado "anti-matéria", indicando que a referida chave não mais existe no banco de dados.

A forma com que as operações de mesclagem são realizadas em uma árvore LSM depende da política de mesclagem adotada (*merge policy*). Embora possam existir implementações intermediárias, essas políticas podem ser classificadas em dois tipos principais, *leveling* e *tiering*, de acordo com a quantidade de componentes (ou *runs*) em cada nível da árvore. A Figura 2.2 ilustra a estrutura básica de uma árvore LSM de acordo com essas duas políticas de mesclagem.

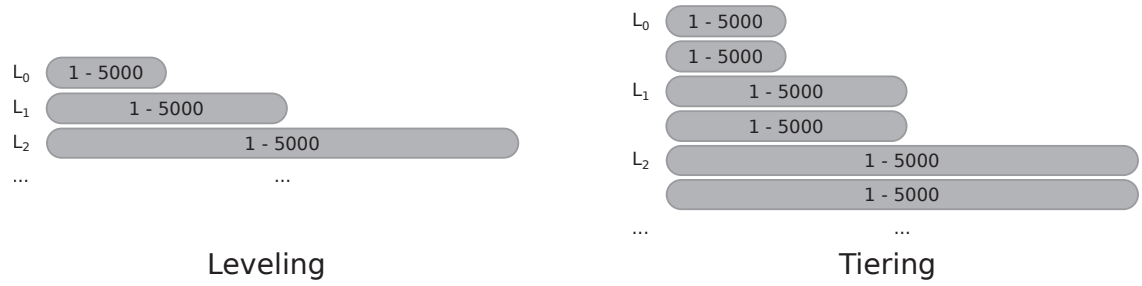


Figura 2.2: Estrutura básica de uma árvore LSM de acordo com as políticas de mesclagem *leveling* e *tiering*.

A política de mesclagem *leveling* considera apenas um componente por nível. Cada nível obedece um tamanho máximo, sendo a razão entre o tamanho máximo de um nível L_{i+1} e o nível L_i igual a um valor $T > 1$ (normalmente, $T = 10$). Os dados são primeiramente inseridos no nível menor (L_0). Conforme ilustrado na Figura 2.3, quando L_0 atinge o tamanho máximo, uma parte ou todos os pares de dados <chave, valor> deste nível são então mesclados com os pares de dados de L_1 e finalmente desvinculados de L_0. Logicamente, este processo é semelhante à fase de mesclagem do algoritmo de ordenação *merge-sort*. Caso a mesma chave exista tanto em L_0 como em L_1, o valor armazenado em L_0 deve prevalecer, visto ser mais recente que L_1. Este mesmo procedimento é executado sucessivamente para os demais níveis da árvore à medida que cada um deles atinge seu respectivo tamanho máximo ($L_1 \rightarrow L_2$, $L_2 \rightarrow L_3$, ..., $L_{N-1} \rightarrow L_N$).

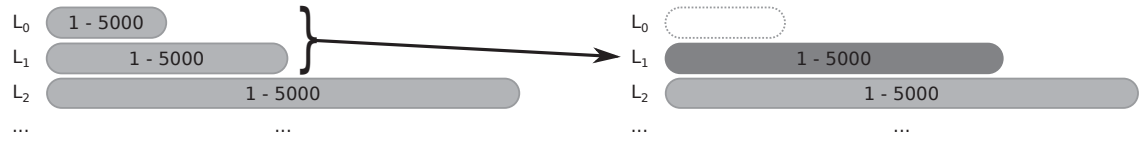


Figura 2.3: Mesclagem entre os níveis L_0 e L_1 usando *leveling*.

Já a política de mesclagem *tiering* comporta mais de um componente por nível. Sendo T a razão de tamanho máximo entre os níveis subjacentes, cada nível comporta até T componentes. No nível L_0, os componentes são criados e preenchidos um de cada vez até o tamanho máximo. Quando o número de componentes completos neste nível atinge o limite T, estes são mesclados e armazenados como um novo componente em L_1, conforme ilustrado na Figura 2.4. Este mesmo processo de mesclagem ocorre nos demais níveis conforme cada um deles recebe novos componentes e eventualmente atingem este número máximo de T componentes.

Independente da política de mesclagem adotada, este processo gradual de sucessivas mesclagens implica que uma mesma chave possa existir em mais de um nível com diferentes versões de valores. Portanto, encontrar o valor mais recente de uma chave requer um trabalho extra de leitura de cada nível (do menor ao maior), até que a referida chave seja encontrada. Este custo adicional é conhecido por amplificação de leitura (*read amplification*) [15].

Do ponto de vista de implementação, diversas técnicas são adotadas para mitigar a amplificação de leitura, embora estas representem uma demanda adicional de memória e espaço

no dispositivo de armazenamento. Uma delas, denominada *fence pointers*, consiste em armazenar o intervalo de chaves de um componente de dados como um todo, ou mesmo de suas divisões internas. Além desta técnica, estruturas auxiliares como os filtros de Bloom (*Bloom filters* [17]) também são largamente empregadas para diminuir a quantidade de acessos a componentes de dados que não possuem as chaves requisitada [64].

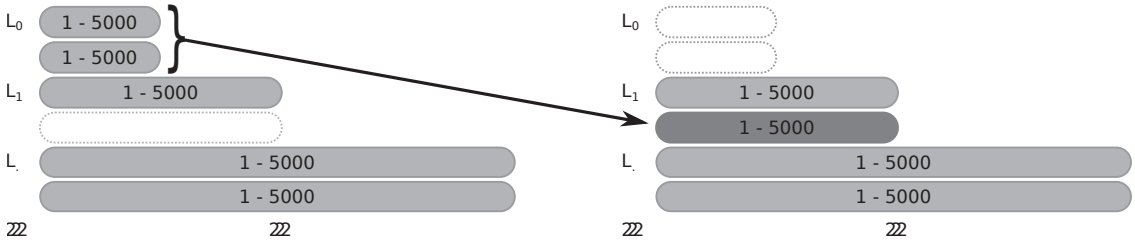


Figura 2.4: Mesclagem entre os componentes do nível L_0 usando *tiering*.

A Tabela 2.1 apresenta os principais custos de acesso ao dispositivo de armazenamento persistente por uma árvore LSM com base nas políticas de mesclagem *leveling* e *tiering* [64]. A razão de tamanho máximo entre os níveis é representada por T. Já N corresponde ao número de níveis da árvore e B representa a quantidade de registros armazenados por página de cada componente de armazenamento. O número total de registros é indicado por R. Já M representa o número total de bits usados pelo filtro de Bloom, enquanto $O(e^{-M/R})$ corresponde à taxa de falsos positivos gerada por este filtro para as operações de busca por chave. A letra s indica a quantidade de chaves buscadas por intervalo, sendo considerados intervalos curtos aqueles que satisfazem a condição $\frac{s}{B} \leq 2N$. Para valores de s maiores, as operações de busca são consideradas de intervalo longo.

Tabela 2.1: Resumo dos custos de uma árvore LSM de acordo com as políticas de mesclagem *leveling* e *tiering* [64].

Política de Mesclagem	Escrita	Busca por chave (existente/não existente)	Busca Intervalo (curto)	Busca Intervalo (longo)	Amplificação de Espaço
Leveling	$O\left(T \cdot \frac{N}{B}\right)$	$O(1)$ / $O\left(N \cdot e^{-\frac{M}{R}}\right)$	$O(N)$	$O\left(\frac{s}{B}\right)$	$O\left(\frac{T+1}{T}\right)$
Tiering	$O\left(\frac{N}{B}\right)$	$O(1)$ / $O\left(T \cdot N \cdot e^{-\frac{M}{R}}\right)$	$O(T \cdot N)$	$O\left(T \cdot \frac{s}{B}\right)$	$O(T)$

Conforme demonstrado na Tabela 2.1, a relação entre os custos de leitura, escrita e espaço de armazenamento varia de acordo com a política de mesclagem [15, 64]. A política *tiering* é geralmente mais eficiente em operações de escrita (inserções, alterações e exclusões de registros) que a política de mesclagem *leveling* devido ao uso de múltiplos componentes de dados em cada nível. Em contrapartida, este número adicional de componentes precisa ser examinado em cada operação leitura, aumentando assim o custo desta política de mesclagem para este tipo de operação.

2.2.2 RocksDB

O RocksDB é um motor de armazenamento chave-valor que usa árvores LSM como estrutura de dados principal. Sendo uma implementação de código aberto relativamente recente e robusta, este motor tem se mostrado relevante tanto por sua adoção em grandes empresas de conteúdo na Internet como por trabalhos de pesquisa produzidos nos últimos anos [29, 30, 19, 85, 40]. Esta seção apresenta apenas uma visão geral do RocksDB e seu

funcionamento padrão. Maiores detalhes e outras funcionalidades podem ser encontradas no repositório do projeto [43].

Desenvolvido pelo Facebook a partir do LevelDB [44], o RocksDB é escrito em linguagem C++ na forma de uma biblioteca, podendo ser incorporada por outros sistemas ou aplicações maiores. Este motor não implementa funcionalidades como comunicação por rede e linguagem de consulta, cabendo aos sistemas que o utilizam implementarem tais funcionalidades, como é o caso do MySQL com MyRocks [35] e de diversos sistemas internos do Facebook [19].

A biblioteca do RocksDB disponibiliza um conjunto de APIs para a manipulação de dados. Dentre as operações básicas, pode-se citar os métodos `Get` (busca pontual), que retorna o valor de uma determinada chave; `Put`, que escreve um par chave-valor na base de dados; e `Delete`, que marca uma chave como excluída. As buscas por intervalo são realizadas por meio de iteradores (`rocksdb::Iterator`) e laços de repetição. Além deste, o método `MultiGet` permite retornar os valores de um conjunto de chaves não essencialmente consecutivas. O RocksDB implementa ainda diversos outros métodos, os quais estão descritos na página de documentação do projeto [36].

A Figura 2.5 apresenta a arquitetura básica do RocksDB [19]. Ela consiste em uma ou mais árvores LSM, chamadas de famílias de colunas (*column families*), as quais podem ser usadas tanto para armazenar bancos de dados distintos como para particionamentos. Além destas árvores, alguns mecanismos auxiliares são compartilhados entre as famílias de colunas, como *log* de transações, metadados gerais e *cache* de blocos em memória. No RocksDB, as árvores LSM adotam uma política de mesclagem essencialmente do tipo *leveling* para todos os níveis, exceto para o nível L_0, no qual é implementada a política *tiering* [34, 64].

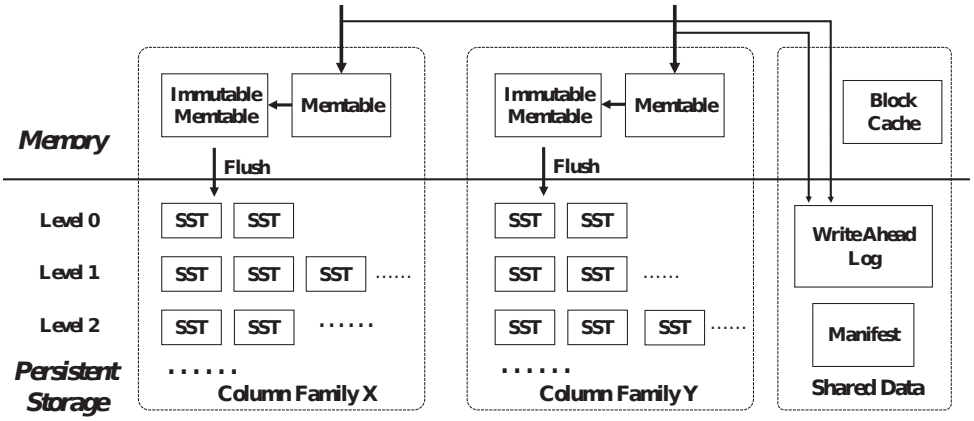


Figura 2.5: Arquitetura básica do RocksDB [19].

Inicialmente, operações de escrita, como `Put` e `Delete`, são registradas em componentes que residem na memória, chamados *memtables*, e no *log* de transações (WAL ou *write-ahead log*) para recuperação. Cada família de coluna possui apenas um *memtable* ativo. Quando o tamanho de um *memtable* atinge um limite estabelecido (parâmetro `write_buffer_size`), este se torna imutável e um novo *memtable* é criado. Os *memtables* imutáveis são então gradativamente transformados em arquivos SST (*Sorted Sequence Table* [37]) e salvos no nível L_0 da árvore que fica no dispositivo de armazenamento persistente. O número de *memtables* imutáveis mesclados em um único SST no nível L_0 é definido pelo parâmetro `min_write_buffer_number_to_merge`. Após o SST ser salvo em L_0, os *memtables* e os registros no WAL referentes a esses dados já não são mais necessários e podem ser descartados.

Os SSTs [37] são arquivos que residem no dispositivo de armazenamento persistente e contém um conjunto de pares chave-valor de um determinado nível da árvore LSM. Os

dados armazenados nesses arquivos são ordenados pelas respectivas chaves e subdivididos em blocos de dados organizados consecutivamente a partir do início do arquivo, podendo ou não estarem compactados. O tamanho de cada bloco é definido pelo parâmetro `block_size`. Após os blocos de dados, esses arquivos armazenam ainda diversos blocos de metadados, como informações dos filtros utilizados (incluindo filtros de Bloom), dicionários de compressão, índices, intervalos de chaves (*fence pointers*) e informações estatísticas.

O tamanho base para os arquivos SST no nível L_1 é estabelecido por parâmetro de configuração (`target_file_size_base`). Para os níveis $L_{i>1}$, outro parâmetro de configuração define a razão de tamanho entre os níveis L_{i+1} e L_i (`target_file_size_multiplier`). Por padrão, este parâmetro tem valor 1, de modo que todos os arquivos SST nos níveis $L_{i\geq 1}$ possuem o mesmo tamanho máximo [38]. Em L_0, por sua vez, o tamanho de cada SST é determinado pelo tamanho e número de *memtables* mesclados em um mesmo arquivo.

A Figura 2.6 ilustra como os intervalos de chaves dos arquivos SST são organizados nas árvores LSM do RocksDB [34]. Com exceção de L_0, todos os demais níveis não apresentam sobreposição de intervalos de chaves entre seus respectivos arquivos. Embora possam existir mais de um arquivo SST em cada nível, esta não sobreposição de intervalos de chaves permite que os níveis $L_{i\geq 1}$ sejam considerados como do tipo *leveling*. Esta organização facilita as operações de busca, visto que no máximo um SST por nível pode conter a chave procurada.

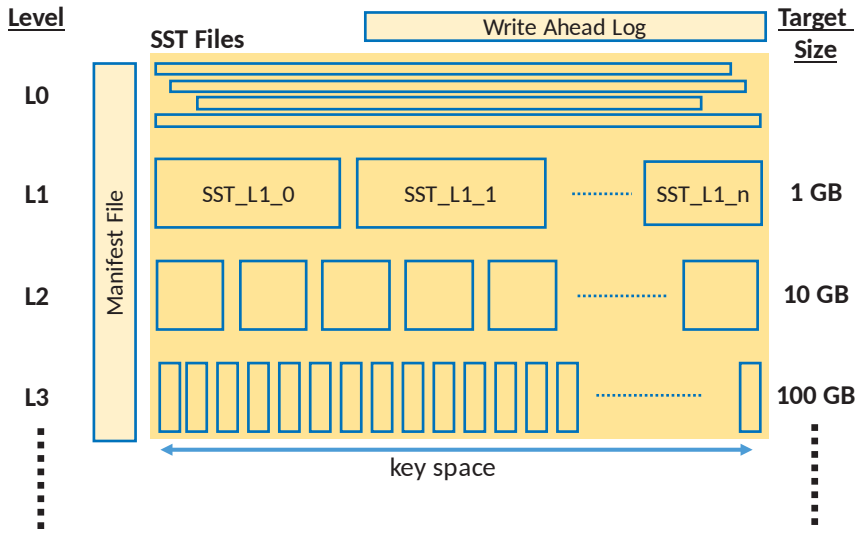


Figura 2.6: Intervalos de chaves dos arquivos SST em cada nível da árvore LSM do RocksDB [34].

No nível L_0, por sua vez, a política *tiering* adotada permite a sobreposição de intervalos de chaves. Esta forma de organização apresenta um custo menor para as operações de escrita, uma vez que cada SST gerado em L_0 não depende de consultas ou mesclagens com os demais arquivos SST deste mesmo nível. Por outro lado, esta política também impõe um custo maior para as operações de busca, conforme descrito anteriormente.

A Figura 2.7 ilustra o processo de mesclagem no RocksDB, chamado *compaction* [34, 39]. Com exceção do nível L_0, quando o tamanho de um nível L_i atinge seu respectivo máximo, a operação de mesclagem seleciona um arquivo SST para ser mesclado com um ou mais arquivos SST que correspondem ao mesmo intervalo de chaves no nível L_{i+1}. O tamanho máximo do nível L_1 é definido pelo parâmetro `max_bytes_for_level_base`, enquanto que os demais níveis $L_{i>1}$ seguem uma proporção de tamanho em relação ao nível L_1 (parâmetro `max_bytes_for_level_multiplier`). O processo de mesclagem gera novos arquivos SST sem modificar os existentes. As chaves marcadas para exclusão no nível L_i são finalmente

removidas, no caso de L_N, e as chaves com valores antigos no nível L_{i+1} são atualizadas. Após o término da mesclagem, os novos arquivos SST são então vinculados ao nível L_{i+1} e os SSTs antigos são removidos dos níveis L_i e L_{i+1}.

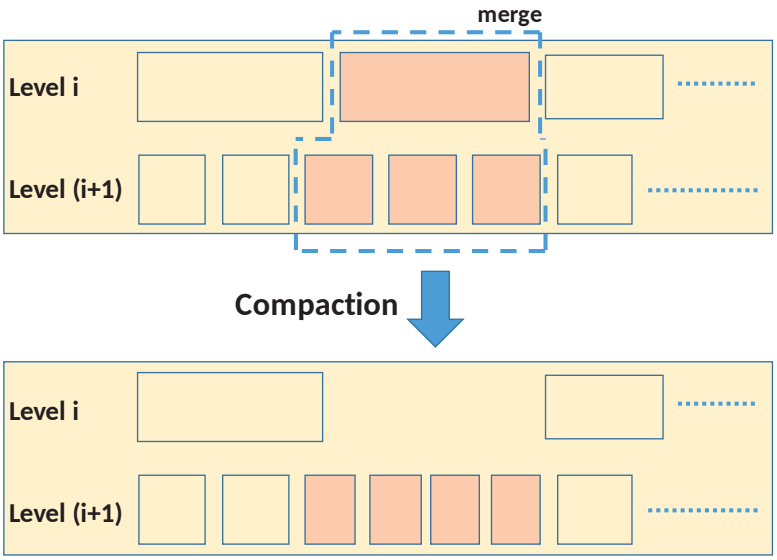


Figura 2.7: Mesclagem de arquivos SST realizada no RocksDB [34].

No nível L_0, o processo de mesclagem é ativado de acordo com a quantidade de arquivos SST existentes, a qual pode ser definida por parâmetro de configuração (`level0_file_num_compaction_trigger`). Neste caso, os arquivos SST no nível L_0 são mesclados com arquivos SST no nível L_1 que correspondem ao intervalo de chaves dos arquivos no nível L_0. Uma vez que cada SST em L_0 pode conter um intervalo de chaves que se estenda por todo o espaço de chaves da base de dados, este processo pode envolver todos os arquivos SST no nível L_1.

No RocksDB, a operação de busca pontual (`Get`) precisa verificar tanto os *memtables* como todos os demais níveis da árvore LSM até que o valor mais recente para da chave desejada seja encontrado ou até que a referida chave não seja encontrada no último nível (L_N). Primeiramente, todos os *memtables* e os SSTs contidos em L_0 devem ser verificados, uma vez que esses componentes admitem a sobreposição de intervalos de chaves. Por não estarem ordenados, os *memtables* precisam ser varridos sequencialmente. Em seguida, cada um dos demais níveis é verificado sucessivamente (L_1, L_2, ..., L_N), primeiramente identificando o SST que contém o intervalo de chaves esperado e em seguida o respectivo bloco de dados que potencialmente possa ter a chave desejada. Neste processo, os filtros de Bloom armazenados em cada SST e mantidos em *cache* servem para reduzir a quantidade de operações de leitura no dispositivo de armazenamento [34].

2.3 SIMULADORES DE SSDS

Um direcionamento de pesquisa importante que vem sendo considerado para entender melhor a dinâmica de desempenho dos dispositivos de armazenamento baseados em memória flash é o uso de simuladores. O MQSIM [86] e o Amber* [47] são propostas recentes para este tipo de abordagem que consideram arquiteturas modernas de SSDs. Tais simuladores buscam capturar um grande número de opções relacionadas ao projeto deste tipo de dispositivo, além de tentar relacionar como estas opções impactam seu desempenho e consumo de energia.

Apesar de serem contribuições significativas à análise de desempenho neste campo de pesquisa, especialmente para o desenvolvimento de protótipos, a maioria dos fabricantes de SSDs não divulga muitos dos detalhes relacionados ao projeto de seus respectivos produtos, fato este que dificulta a aproximação de desempenho em ambientes simulados com ambientes reais de produção. Além disso, tais simuladores não abrangem a interferência de desempenho causada por cargas de trabalho concorrentes nem o relacionamento desses dispositivos de armazenamento com outros componentes de *software*, como sistema operacional e sistema de arquivos, os quais podem inclusive variar de acordo com cada versão instalada. O mapeamento de todos esses detalhes em um ambiente completo de simulação acaba sendo excessivamente complexo, podendo prejudicar a acurácia dos dados de desempenho obtidos.

O trabalho apresentado por Yadgar et al. [96] analisa o desempenho de diferentes cargas de trabalhos executadas sobre dispositivos de armazenamento baseados em memória flash. De acordo com os autores, esta caracterização foi extensivamente investigada no passado utilizando HDDs. Contudo, devido às mudanças significativas de arquitetura e de complexidade intrínseca dos atuais SSDs, a maioria das análises anteriores não reflete o desempenho dos cenários de produção atuais, onde uma parte significativa de aplicações utiliza dispositivos baseados em memória flash para armazenamento persistente de dados. Utilizando traces públicos de cargas de trabalho provenientes de diferentes aplicações, incluindo RocksDB, Yadgar et al. implementaram dois simuladores simplificados para medir as amplificações de leitura e gravação das cargas de trabalho analisadas, variando o número de partições usadas para separar dados pouco acessados (frios) de dados frequentemente acessados (quentes), incluindo a granularidade (tamanho de página) utilizada para as operações de leitura e gravação. Embora os autores destaquem a importância desta análise considerando as especificidades desta tecnologia de armazenamento, este trabalho não considera a interferência causada por múltiplas cargas de trabalho alocadas no mesmo dispositivo.

2.4 BENCHMARKS

Esta seção apresenta as principais ferramentas de *benchmark* e avaliações de desempenho relacionadas com este trabalho. Primeiramente, a Seção 2.4.1 descreve duas ferramentas relevantes de *benchmark* para bancos de dados chave-valor. Em seguida, a Seção 2.4.2 lista diferentes ferramentas destinadas a avaliar o desempenho de infraestruturas de computação em nuvem e de *Big Data*. Por fim, a Seção 2.4.3 apresenta uma análise sobre o desempenho de motores de armazenamento chave-valor em dispositivos baseados em memória flash.

2.4.1 Benchmarks Chave-Valor

O *Yahoo! Cloud Serving Benchmark* (YCSB) [25, 24] é atualmente considerado um dos mais representativos *benchmarks* para bancos de dados chave-valor, sendo amplamente utilizado tanto pela comunidade científica como pela indústria de *software*. Implementado em linguagem de programação Java, este *benchmark* constitui um projeto flexível que suporta um grande número de SGBDs relacionais e NoSQL, além de diversos motores de armazenamento embutidos na forma de bibliotecas.

A Figura 2.8 ilustra a arquitetura adotada pelo YCSB. Para suportar diferentes SGBDs e motores de armazenamento, boa parte de sua implementação é feita de forma genérica, incluindo a geração de cargas de trabalho, controle de *threads* para simular as conexões de clientes com o banco de dados e a coleta de estatísticas. A tradução desta representação genérica é posteriormente feita para cada tipo de banco de dados suportado por meio de uma interface denominada *DB Interface Layer*.

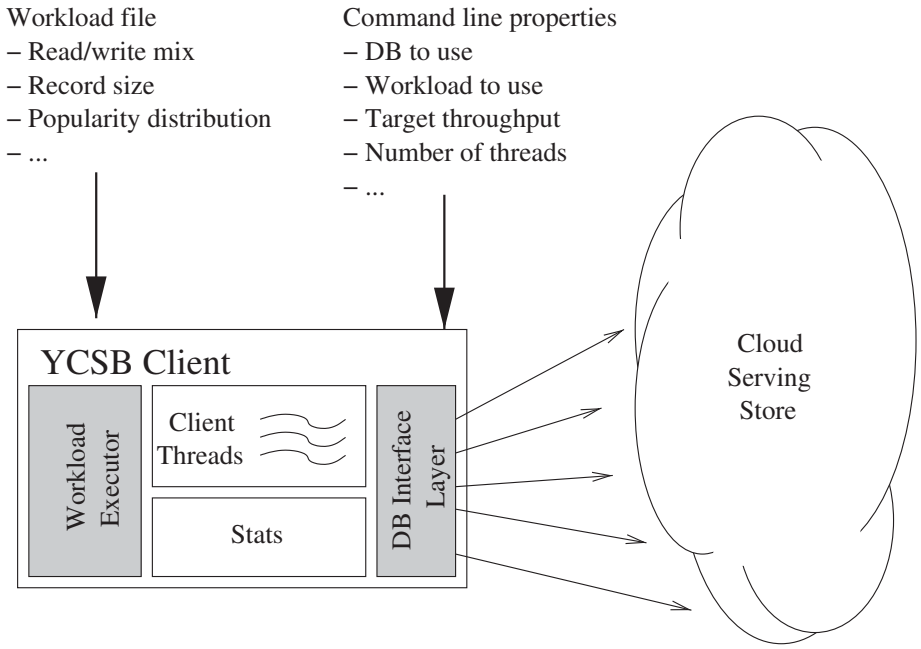


Figura 2.8: Arquitetura do YCSB [25].

No YCSB, o banco de dados é composto por um conjunto de registros (tuplas) na forma de pares <chave, valor>, cuja quantidade é definida pelo parâmetro `recordcount`. Em cada registro, a chave é definida textualmente na forma "userX", sendo "X"um valor numérico. O tamanho da chave pode variar de acordo com este valor numérico ou ser fixado com o uso de zeros à esquerda por meio do parâmetro `zeropadding`. Dependendo do valor atribuído para o parâmetro `recordcount`, este tamanho de chave pode chegar a 20 bytes ou mais. Exemplo: "user9999971396733075 41".

O valor de cada registro do banco de dados é composto por um ou mais campos contendo bytes aleatórios. A quantidade de campos é definida pelo parâmetro `fieldcount` e a quantidade de bytes por campo obedece o parâmetro `fieldlength`. Por padrão, `fieldcount` é igual a 10 e `fieldlength` é igual a 100, o que corresponde a 1000 bytes de valor para cada registro do banco de dados.

Diferente de *benchmarks* para bancos de dados relacionais, como o TPC-C e TPC-H [91, 92], o YCSB usa tipos de operações bastante simples para compor suas cargas de trabalho, sendo que a proporção destas é controlada na forma de parâmetros de configuração:

- `readproportion` (valores de 0 a 1): define a proporção de operações de leitura de pares <chave, valor> (equivalente a consultas *point lookup*);
- `updateproportion` (valores de 0 a 1): define a proporção de operações de atualização de valores para chaves já existentes no banco de dados;
- `insertproportion` (valores de 0 a 1): define a proporção de operações de inserção de novos pares <chave, valor> no banco de dados;
- `readmodifywriteproportion` (valores de 0 a 1): define a proporção de leituras seguidas por modificações de valores de uma mesma chave;
- `scanproportion` (valores de 0 a 1): define a proporção de operações de consulta por intervalo de chaves (equivalente a consultas do tipo *range query*).

Além da proporção de cada tipo de operação que compõe uma carga de trabalho gerada pelo YCSB, a distribuição com que as chaves do banco de dados são selecionadas é controlada pelo parâmetro `requestdistribution`. Dentre as mais utilizadas estão a distribuição uniforme (`uniform`), de Zipf (`zipfian`) e chaves inseridas ou acessadas recentemente (`latest`). O número de *threads* utilizadas para simular conexões simultâneas de clientes é controlado pelo parâmetro `threadcount`.

Embora o YCSB permita que o usuário personalize cargas de trabalho a partir dos parâmetros de proporcionalidade descritos acima, este *benchmark* fornece um conjunto de seis cargas de trabalho principais pré-definidas (*core workloads*):

- Carga de trabalho A (`workloada`): Composta por 50% de operações de leitura e 50% de operações de atualização, é considerada uma carga de trabalho de atualizações intensivas. Parâmetros:
  - `readproportion=0.5`;
  - `updateproportion=0.5`;
  - `requestdistribution=zipfian`;
- Carga de trabalho B (`workloadb`): Composta por 95% de leituras e 5% de atualizações, é considerada uma carga de trabalho predominantemente de leitura. Parâmetros:
  - `readproportion=0.95`;
  - `updateproportion=0.05`;
  - `requestdistribution=zipfian`;
- Carga de trabalho C (`workloadc`): Composta por 100% de operações de leitura. Parâmetros:
  - `readproportion=1`;
  - `requestdistribution=zipfian`;
- Carga de trabalho D (`workloadd`): Composta por 95% de operações de leitura e 5% de inserções de novos registros. Os novos registros são considerados como os mais populares para as operações de leitura posteriores. Parâmetros:
  - `readproportion=0.95`;
  - `insertproportion=0.05`;
  - `requestdistribution=latest`;
- Carga de trabalho E (`workloade`): Composta por 95% de operações de consulta por intervalo e 5% de operações de inserção de novos registros. O intervalo de cada consulta é selecionado aleatoriamente entre 1 e 100 usando uma distribuição uniforme. Parâmetros:
  - `scanproportion=0.95`;
  - `insertproportion=0.05`;
  - `requestdistribution=zipfian`;
  - `minscanlength=1`;

  - `maxscanlength=100`;
  - `scanlengthdistribution=uniform`;

- Carga de trabalho F (`workloadf`): Composta por 50% de operações de leitura e 50% de leituras seguidas de modificações. Parâmetros:

  - `readproportion=0.5`;
  - `readmodifywriteproportion=0.5`;
  - `requestdistribution=zipfian`.

O db_bench é uma ferramenta de *benchmark* chave-valor desenvolvida como parte do projeto LevelDB e posteriormente herdada pelo projeto RocksDB [43]. Esta ferramenta implementa mais de quarenta opções de cargas de trabalho destinadas a avaliar diferentes aspectos deste motor de armazenamento. Algumas dessas opções seguem os mesmos propósitos de avaliação utilizados pelas cargas de trabalho principais do YCSB. Por exemplo, a carga de trabalho `readrandomwriterandom` pode ser configurada para realizar diferentes proporções de leituras e atualizações por meio do parâmetro `readwritepercent`. Embora semelhantes em configuração, a implementação adotada pelo db_bench diverge em diversos aspectos da forma com que o YCSB foi implementado, inclusive na seleção aleatória de chaves, na composição dos dados vinculados a cada chave e no controle de *threads* que simulam os acessos simultâneos ao banco de dados.

Além de poder produzir cargas de trabalho semelhantes ao YCSB, o db_bench implementa ainda algumas opções que exercitam questões específicas do RocksDB, como as operações de mesclagem de registros (ex.: `mergerandom` e `readrandommergerandom`). Além destas opções, outras cargas de trabalho derivam observações de sistemas reais em produção, conforme apresentado por Cao et al. [19] (ex.: `mixgraph`).

Embora ferramentas de *benchmark* como o YCSB e db_bench sejam importantes para avaliar o desempenho de sistemas gerenciadores de bancos de dados e motores de armazenamento chave-valor em diferentes configurações e tipos de *hardware*, o projeto dessas ferramentas considera essencialmente bancos de dados executados separadamente, ignorando condições onde múltiplas aplicações ou inquilinos compartilham os mesmos recursos de armazenamento persistente, de processamento e memória. Devido à falta deste tipo de avaliação na dinâmica de testes dessas ferramentas, outras propostas de *frameworks* maiores geralmente incorporam um ou mais *benchmarks* isolados com o intuito de avaliar cenários mais complexos e dinâmicos. Algumas dessas propostas serão descritas na Seção 2.4.2.

2.4.2 Benchmarks para IaaS e Big Data

O *SPEC Cloud IaaS 2018* [84] é um *benchmark* desenvolvido pela *Standard Performance Evaluation Corporation* para avaliar o desempenho de provedores de nuvem que fornecem infraestrutura como serviço (*Infrastruture as a Service*, ou IaaS). O principal objetivo deste *benchmark* é avaliar a escalabilidade de uma infraestrutura de nuvem à medida que mais aplicações são gradualmente instanciadas nela. Para simular as aplicações de nuvem, o SPEC Cloud IaaS 2018 usa dois tipos de cargas de trabalho:

- A carga de trabalho D do YCSB utilizando o Apache Cassandra [2] como SGBD chave-valor para simular cargas de trabalho intensivas em E/S sobre os recursos de armazenamento persistente; e

- Uma implementação do algoritmo de agrupamento K-Means para simular cargas de trabalho intensivas em CPU. Esta implementação é parte da ferramenta de *benchmark* HiBench da Intel [52], a qual utiliza o Apache Hadoop [4] como plataforma de execução distribuída.

Cada uma dessas cargas de trabalho é composta por seis máquinas virtuais. Em seu ciclo de teste, este *benchmark* aumenta progressivamente o número de cada uma dessas cargas de trabalho até que uma das condições de parada pré-estabelecidas seja alcançada. Por exemplo: percentual de falhas de provisionamento de novas aplicações ou violações de qualidade de serviço (*Quality of Service*, ou QoS).

Embora o SPEC Cloud IaaS 2018 possa ser considerado uma boa opção para avaliar a escalabilidade de infraestruturas de nuvem com múltiplos inquilinos ao simular um número crescente de aplicações compostas por várias máquinas virtuais, a abordagem de teste adotada por este *benchmark* é de propósito geral, misturando cargas de trabalho intensivas em processamento com cargas intensivas em E/S. Este tipo de abordagem não é adequado para testar as características do dispositivo de armazenamento persistente compartilhado, uma vez que gargalos de CPU e de E/S podem se misturar durante os testes. Além disso, este *benchmark* também não exerce controle sobre a alocação de máquinas virtuais em um mesmo *hardware*, ficando a cargo do provedor de IaaS as decisões sobre a orquestração e o provisionamento de recursos.

Proposto por Seybold et al. [81], o *Mowgli* é outro *framework* de *benchmark* desenvolvido para avaliar o desempenho de SGBDs NoSQL em ambientes de nuvem que fornecem recursos de infraestrutura como serviço (IaaS). Diferente do SPEC Cloud IaaS 2018, a abordagem adotada por este *framework* considera uma única instância de banco de dados por teste realizado, embora o número de opções disponíveis para definir esta instância seja muito mais flexível que no outro caso, incluindo o tipo do SGBD, número de nodos e garantia de consistência das transações. Com base nessas opções, o Mowgli consegue automatizar todo o ciclo do experimento, desde a inicialização e carga do banco de dados até a coleta de estatísticas de desempenho. Contudo, esta ferramenta não controla quais as cargas de trabalho estão executando concorrentemente na mesma infraestrutura nem considera as características do *hardware* que a compõe.

Desenvolvido por Wang et al. [94], o *BigDataBench* é outra proposta de *benchmark* que contempla diversas bases de dados e cargas de trabalho extraídas de ambientes reais de produção para avaliar o desempenho de infraestruturas computacionais destinadas ao processamento de aplicações de *Big Data*. Neste *benchmark*, tanto os dados como as cargas de trabalho são bastante variadas, abrangendo aplicações transacionais, de comércio eletrônico, redes sociais e motores de busca. A maior parte dessas cargas de trabalho é implementada com base no modelo map-reduce e utiliza o Apache Hadoop [4] como plataforma de execução distribuída. Algumas dessas cargas de trabalho são projetadas para avaliar as operações básicas de bancos de dados chave-valor, como leituras, gravações e buscas por intervalo, as quais utilizam o HBase [1] e o Cassandra [2] como SGBDs. Embora o BigDataBench seja voltado para infraestruturas de nuvem ou *clusters* de processamento paralelo, sua abordagem não considera a interferência de desempenho como parte de seu projeto nem questões intrínsecas aos dispositivos de armazenamento baseados em memória flash.

2.4.3 Motores de Armazenamento Chave-Valor e SSDs Baseados em Memória Flash

O trabalho apresentado por Didona et al. [32] analisa o desempenho de dois tipos de motores de armazenamento chave-valor (RocksDB [43] e WiredTiger [69]) utilizando dispositivos de armazenamento baseados em memória flash. De acordo com os autores, tais dispositivos implementam lógicas complexas em seus respectivos controladores para lidarem com

as características intrínsecas deste tipo de memória persistente, podendo produzir comportamentos inesperados de desempenho, além de comprometer a confiabilidade e a reprodutibilidade de resultados experimentais. Através de uma série experimentos, Didona et al. enumeraram sete possíveis armadilhas (*pitfalls*) que podem ocorrer ao avaliar o desempenho de motores de armazenamento chave-valor utilizando tais dispositivos, apresentando também algumas recomendações importantes para evitá-las:

1. *Execução de testes curtos:* Os dispositivos flash podem requerer um certo tempo para estabilizar seu desempenho, de modo que *benchmarks* curtos podem produzir medições de desempenho incompatíveis com execuções mais longas.

2. *Ignorar a amplificação de escrita do dispositivo (WA-D):* Os autores definem como WA-D a razão entre a quantidade de dados efetivamente gravada no dispositivo de armazenamento e a quantidade de dados que o sistema operacional solicitou para gravação. Valores de WA-D maiores que 1 podem indicar que o dispositivo flash está realizando operações de coleta de lixo, podendo resultar em degradação de desempenho.

3. *Ignorar o estado interno do dispositivo:* Dependendo do motor de armazenamento chave-valor, os autores demonstram que o estado interno das páginas flash do SSD (especialmente a quantidade de páginas apagadas) pode influenciar as medições de desempenho.

4. *Ignorar o tamanho do banco de dados:* A quantidade de dados requerida pelo banco de dados pode influenciar o estado interno do dispositivo de armazenamento e, consequentemente, seu desempenho.

5. *Ignorar o espaço extra de armazenamento que o motor de armazenamento chave-valor precisa para organizar os dados e armazenar metadados adicionais:* Além do tamanho do banco de dados, a amplificação de espaço também exerce influência sobre a quantidade de dados que precisa ser armazenada no dispositivo de armazenamento persistente.

6. *Ignorar o excesso de provisionamento do SSD:* A quantidade de páginas flash reservadas no dispositivo de armazenamento para provisionamento em excesso também pode influenciar o desempenho, uma vez que elas podem reduzir o WA-D.

7. *Ignorar o efeito da tecnologia de armazenamento utilizada pelo dispositivo:* Cada modelo de SSD pode apresentar algum tipo de lógica de programação e *hardware* diferente em relação a outros modelos, podendo influenciar no desempenho do motor de armazenamento chave-valor.

Embora Didona et al. tenham apresentado considerações importantes em relação à avaliação de desempenho de motores de armazenamento chave-valor em SSDs baseados em memória flash, a análise apresentada por eles se restringe a cargas de trabalho isoladas, ignorando a presença de múltiplas aplicações compartilhando o mesmo dispositivo de armazenamento.

2.5 MÚLTIPLOS INQUILINOS, INTERFERÊNCIA E ISOLAMENTO DE DESEMPENHO

A interferência de desempenho entre cargas de trabalho tem sido investigada por diferentes estudos utilizando diversos tipos de recursos computacionais com o objetivo de melhorar o isolamento entre inquilinos e a utilização de infraestruturas de computação em nuvem.

O trabalho apresentado por Mars et al. [65] analisa a sensitividade de diferentes cargas de trabalho em produção na Google em relação ao uso compartilhado de memória RAM. Neste estudo, os autores propõem um método de caracterização de desempenho chamado *bubble-up*, no qual uma carga de trabalho é executada concorrentemente com outro *microbenchmark* (chamado *bubble*, ou bolha) que progressivamente aumenta sua demanda sobre o sistema de memória, representando uma "expansão de bolha". À medida que esta bolha aumenta, o desempenho da carga de trabalho analisada é monitorado com o objetivo de determinar sua degradação relativa. Com base na curva obtida de degradação de desempenho em relação ao *microbenchmark bubble*, Mars et al. determinam a melhor co-alocação de cargas de trabalho sensíveis a latência (*latency-sensitive*) e em lote (*batch*) com o objetivo de maximizar a utilização da infraestrutura de nuvem sem comprometer significativamente a qualidade de serviço das cargas de trabalho sensíveis a latência.

Já o trabalho apresentado por Yang et al. [97] estende o mecanismo proposto por Mars et al.. Denominado *bubble-flux*, o mecanismo proposto por esses autores é mais dinâmico que o *bubble-up*, o qual realiza avaliações curtas de interferência de desempenho (*dynamic bubble*) ao mesmo tempo que monitora a qualidade de serviço das cargas de trabalho sensíveis a latência. Embora os trabalhos apresentados por Mars, Yang et al. busquem avaliar a interferência de desempenho de cargas de trabalho concorrentes, tais estudos consideram a memória RAM como recurso de infraestrutura compartilhado. Contudo, a avaliação deste tipo de interferência em dispositivos de armazenamento baseados em memória flash representa um desafio adicional devido à dinâmica e complexidade intrínseca deste tipo de armazenamento persistente.

A interferência de desempenho entre diferentes inquilinos em ambientes de nuvem que fornecem infraestrutura como serviço também foi investigada por Novakovic et al. [71]. Usando métricas de CPU e memória para analisar o comportamento de desempenho de aplicações executadas em máquinas virtuais (MVs), os autores propõem um mecanismo chamado *DeepDive* para detectar as MVs com possíveis problemas de desempenho. Neste processo, as MVs detectadas pelo *DeepDive* são então duplicadas para outro servidor físico e executadas isoladamente para comparar seu desempenho em ambas as circunstâncias (recursos compartilhados *versus* isolados). Caso seja confirmada a interferência de desempenho, a MV é então migrada para outro servidor.

No trabalho apresentado por Xu et al. [95], os autores propõem um mecanismo chamado *Pythia* para gerenciar a co-alocação de aplicações sensíveis a latência com cargas de trabalho em lote. Neste estudo, Xu et al. combinam *microbenchmarks* e modelos de regressão linear para caracterizar a interferência de desempenho entre cargas de trabalho, considerando as contenções de CPU e memória como principais fontes de degradação de desempenho.

Considerando os mesmos recursos que Xu et al., o trabalho apresentado por Kannan et al. [54] propõe um método chamado *Caliper*, o qual se baseia em micro-experimentos para estimar a interferência de desempenho entre inquilinos em ambientes de produção. Em vez de executar *microbenchmarks* para produzir a interferência de desempenho, o *Caliper* realiza pequenas pausas (em torno de milissegundos) nas cargas de trabalho concorrentes de uma determinada aplicação para estimar seu desempenho quando executada isoladamente. Embora as abordagens adotadas por *Pythia* e *Caliper* possam ser adequadas para estimar possíveis contenções de CPU e memória causadas por cargas de trabalho concorrentes, a janela de tempo usada por essas abordagens é muito pequena para recursos de armazenamento persistente baseados em memória flash.

Proposto por Delimitrou e Kozyrakis [31], o *Paragon* representa outro mecanismo relevante de escalonamento para ambientes multi-inquilinos de infraestrutura como serviço, o qual visa minimizar a interferência de desempenho de cargas de trabalho concorrentes. Ao invés de considerar primariamente métricas de CPU e memória para estimar esta interferência, os

autores implementaram um *microbenchmark* ajustável para cada tipo de recurso compartilhado avaliado: memória, *cache* de CPU e largura de banda de rede e armazenamento. Apesar de considerarem um grande número de recursos compartilhados, o simples ajuste de largura de banda para estimar a interferência de desempenho em dispositivos de armazenamento não é realista o suficiente para mapear a multidimensionalidade dos atuais SSDs baseados em memória flash [32].

Considerando mais especificamente os dispositivos de armazenamento baseados em memória flash, os estudos apresentados por Kim et al. [55] e Kwon et al. [56] propõem duas abordagens semelhantes para mitigar a interferência de desempenho entre inquilinos. Usando o simulador DiskSim [18], Kim et al. demonstram que a coleta de lixo é uma das principais fontes de interferência de desempenho entre cargas de trabalho. De acordo com essas observações, os autores propõem um mecanismo para separar os blocos de memória flash alocados para cada inquilino. Já no trabalho de apresentado por Kwon et al., os autores apresentam um protótipo de *hardware* com múltiplas regiões de memória flash, as quais são usadas para isolar fisicamente cada inquilino. Embora essas duas abordagens sejam relevantes para o estudo de isolamento de desempenho, ambas requerem tipos de *hardware* e de controle não disponíveis na maioria dos dispositivos SSDs comercializados atualmente. De qualquer modo, tal objetivo pode ser similarmente alcançado com a introdução de múltiplos dispositivos de armazenamento em um mesmo servidor, embora isto seja, de certa forma, contraditório à proposta de compartilhamento de recursos.

2.6 SUMÁRIO

A Tabela 2.2 sumariza os trabalhos relacionados descritos neste capítulo considerando três critérios principais relacionados com a pergunta de pesquisa Q1 apresentada na Seção 1.1:

- Avaliação de interferência de desempenho e de infraestruturas com múltiplos inquilinos;
- Uso de motores de armazenamento chave-valor ou SGBDs NoSQL como principal critério de avaliação de desempenho; e
- Uso de SSDs baseados em memória flash como principal tecnologia de armazenamento persistente.

Para cada critério da Tabela 2.2, o símbolo “R” indica que o mesmo foi considerado pelos autores como fator relevante para a elaboração do referido estudo. Em contrapartida, o símbolo “–” indica que o critério não foi mencionado ou não influenciou esta elaboração. Em uma escala intermediária, esta tabela também utiliza o símbolo “P” para indicar a presença de motores de armazenamento chave-valor ou SGBDs NoSQL como parte dos experimentos em conjunto com outros tipos de cargas de trabalho, embora estes não sejam avaliados do ponto de vista arquitetural. Para os trabalhos sobre interferência de desempenho, o símbolo “I” indica que a estratégia adotada pelos autores consiste em isolar os componentes de *hardware* que compõem o dispositivo de armazenamento.

Os trabalhos listados na Tabela 2.2 demonstram a ausência de ferramentas e análises de desempenho que contemplem simultaneamente todos os três critérios apresentados, justificando assim o desenvolvimento de um *framework* de teste que responda a pergunta de pesquisa Q1. Atualmente, as ferramentas de *benchmark* desenvolvidas para avaliar motores de armazenamento chave-valor e SGBDs NoSQL executados isoladamente não consideram questões intrínsecas da tecnologia de armazenamento em memória flash [25, 24, 43]. Tais questões também não são

consideradas em ferramentas de *benchmarks* destinadas a avaliar infraestruturas com múltiplos inquilinos [84, 81] e de Big Data [94].

Conforme apresentado nas Seções 2.1 e 2.2 e pelos trabalhos que analisam o comportamento de dispositivos de armazenamento baseados em memória flash [86, 47, 55, 56, 96, 32], tanto a dinâmica de funcionamento deste tipo de dispositivo como sua relação com o desempenho de motores de armazenamento chave-valor possuem características complexas que dependem tanto do projeto e estado de cada dispositivo como do estado do próprio banco de dados. A introdução de cargas de trabalho concorrentes neste contexto representa uma complexidade adicional a esta relação de desempenho. Em contrapartida, os trabalhos que avaliam a interferência de desempenho consideram predominantemente recursos de CPU e memória, ignorando esta dinâmica de funcionamento dos dispositivos baseados em memória flash [65, 97, 71, 95, 54, 31].

Tabela 2.2: Sumário dos trabalhos relacionados de acordo com cada item avaliado.

Trabalho Relacionado	**Interferência de Desempenho**	**Bancos de Dados Chave-Valor**	**Dispositivos Flash**
Tavakkol et al. [86]	–	–	R
Gouk [47]	–	–	R
Yadgar et al. [96]	–	R	R
YCSB [25, 24]	–	R	–
db_bench [43]	–	R	–
SPEC Cloud IaaS 2018 [84]	R	P	–
Seybold et al. [81]	R	R	–
Wang et al. [94]	–	R	–
Didona et al. [32]	–	R	R
Mars et al. [65]	R	P	–
Yang et al. [97]	R	P	–
Novakovic et al. [71]	R	P	–
Xu et al. [95]	R	P	–
Kannan et al. [54]	R	–	–
Delimitrou e Kozyrakis [31]	R	–	–
Kim et al. [55]	I	–	R
Kwon et al. [56]	I	–	R

O símbolo “R” indica que o critério foi considerado relevante para o estudo, “–” indica que o critério não foi mencionado ou não influenciou sua elaboração, “P” indica a presença de motores de armazenamento chave-valor ou SGBDs NoSQL, mas sem análise de suas arquiteturas, e “I” indica que a estratégia adotada é de isolamento de recursos.

3 FRAMEWORK STORIKS

Este capítulo apresenta a arquitetura e os principais componentes do *framework* Storiks, proposto para avaliar a interferência de desempenho sofrida por um motor de armazenamento chave-valor. Primeiramente, a Seção 3.1 descreve as definições principais, necessárias para a metodologia proposta. Em seguida, a Seção 3.2 traz uma visão geral dos requisitos e limitações considerados para a construção deste *framework*. A arquitetura e os principais componentes são apresentados na Seção 3.3, enquanto que as cargas de trabalho são descritas na Seção 3.4.

3.1 DEFINIÇÕES PRINCIPAIS

Antes de descrever a arquitetura e os componentes do *framework* Storiks, esta seção organiza os principais elementos necessários para a elaboração de sua metodologia e implementação, os quais são uma extensão de trabalhos apresentados anteriormente [60, 61]. Devido à complexidade de *hardware* e *software* envolvida, a técnica usada aqui para avaliar a interferência de desempenho se baseia essencialmente em experimentações.

A partir da pergunta de pesquisa Q1, é possível fazer as seguintes suposições para o desenvolvimento deste *framework*:

1º – O ambiente a ser analisado é composto por múltiplas cargas de trabalho;

2º – Uma dessas cargas de trabalho é executada sobre um motor de armazenamento chave-valor, cujo desempenho representa a principal referência de análise;

3º – A outra ou as demais cargas de trabalho, chamadas aqui de cargas de trabalho concorrentes, exercem um certo nível de contenção ao acessar os mesmos recursos computacionais compartilhados; e

4º – O recurso computacional de interesse neste estudo corresponde a dispositivos de armazenamento baseados em memória flash.

Considerando as suposições apresentadas acima, faz-se necessário estabelecer um relacionamento entre o desempenho do motor de armazenamento chave-valor e as cargas de trabalho concorrentes, podendo ser matematicamente definido da seguinte forma:

Definição 1 *Seja D o conjunto de possíveis cargas de trabalho concorrentes e C um espaço linear representando o critério de desempenho a ser analisado para um motor de armazenamento chave-valor. A função de* ***pressão*** *$\rho : D \rightarrow C$ representa o mapeamento funcional entre D e C.*

Neste nível de abstração, a noção de desempenho representada por C na Definição 1 não é específica a qualquer métrica ou escala. Contudo, conforme necessário para a definição seguinte, C precisa apresentar uma ordem linear estabelecida, de forma que seja possível determinar o maior e menor desempenho entre quaisquer dois valores não iguais em C. Embora esta definição possa ser facilmente aplicada para qualquer critério de desempenho linearmente ordenado, este estudo considera como C a taxa de transações por segundo (tx/s) alcançada pelo motor de armazenamento chave-valor.

A função de pressão ρ, descrita pela Definição 1, considera que $d_i \in D$ é a principal fonte de interferência sobre o desempenho do motor de armazenamento chave-valor. Para

comparar o efeito de diferentes cargas de trabalho sobre este desempenho, faz-se necessário estabelecer uma ordem entre as instâncias de D, a qual pode ser formalizada conforme a seguinte definição:

Definição 2 *Seja $(C, \preceq_C)$ um conjunto linearmente ordenado, onde $c_a \preceq_C c_b$ significa "c_a é um valor de desempenho inferior ou igual a c_b" para todo $c_a, c_b \in C$. Define-se por* ***escala de pressão*** *a pré-ordem linear $(D, \succeq_D)$, tal que $d_a \succeq_D d_b$ se, e somente se, $\rho(d_a) \preceq_C \rho(d_b)$. A relação binária $d_a \succeq_D d_b$ significa "d_a exerce mais ou a mesma pressão que d_b."*

Em outras palavras, a Definição 2 estabelece que, quanto menor o desempenho obtido pelo motor de armazenamento chave-valor executado concorrentemente com d_i, maior a pressão exercida por d_i sobre este motor de armazenamento. No caso de C corresponder ao critério de desempenho tx/s, o conjunto linearmente ordenado $(C, \preceq_C)$ pode ser trivialmente definido como $(C, \leq)$. Uma vez que $(D, \succeq_D)$ é uma pré-ordem linear, existe a possibilidade de mais de uma carga de trabalho concorrente exercer a mesma pressão sobre o motor de armazenamento. Duas cargas de trabalho d_a e $d_b \in D$ são consideradas equivalentes, se $\rho(d_a) = \rho(d_b)$.

Para simplificar a análise numérica, pode-se ainda normalizar a pressão produzida por cada carga de trabalho concorrente de acordo com a seguinte definição:

Definição 3 *Considere d_0 um caso particular de D representando a ausência de qualquer carga de trabalho concorrente. Define-se por* ***pressão normalizada*** *de $d_i \in D$, representada por $\overline{\rho}(d_i)$, a proporção em termos de degradação de desempenho que d_i representa em relação a d_0:*

$$\overline{\rho}(d_i) = \begin{cases} (\rho(d_0) - \rho(d_i))/\rho(d_0), & se\ (C, \preceq_C) = (C, \leq) \\ (\rho(d_i) - \rho(d_0))/\rho(d_0), & se\ (C, \preceq_C) = (C, \geq) \end{cases} \tag{3.1}$$

Conforme apresentado na Definição 3, conhecer o desempenho do motor de armazenamento chave-valor na ausência de cargas de trabalho concorrentes (ou seja, $\rho(d_0)$) é importante para representar a degradação de desempenho em termos relativos. Por exemplo, caso $\overline{\rho}(d_i)$ seja igual a 0,1, a carga de trabalho concorrente d_i exerce uma degradação de desempenho de 10% sobre este motor de armazenamento em relação a $\rho(d_0)$.

O cálculo da pressão normalizada, representada por $\overline{\rho}(d_i)$, requer que somente d_0 e d_i variem durante as medições de $\rho(d_0)$ e $\rho(d_i)$. Isso significa que outras condições de sistema e cargas de trabalho precisam ser aproximadamente as mesmas, incluindo a carga de trabalho submetida ao motor chave-valor, o dispositivo de armazenamento e a pilha de *software* utilizados nesta execução, além de seus respectivos estados internos. De acordo com o Capítulo 1, algumas dessas condições são consideradas na hipótese deste trabalho, de forma que tanto $\rho(d_0)$ como $\rho(d_i)$ precisam ser medidos novamente a cada variação dessas condições. As condições relacionadas com esta hipótese e a metodologia usada para obter os valores de desempenho serão analisadas mais adiante neste e no próximo capítulo.

3.2 REQUISITOS DE IMPLEMENTAÇÃO

Esta seção discute os principais requisitos e limitações consideradas durante o desenvolvimento do *framework* Storiks. Semelhantemente a Seybold et al. [81], foram enumerados seis requisitos desejáveis para o desenvolvimento deste *framework*:

- (R1) Fácil utilização: O *framework* precisa ser de simples implantação e prover uma interface amigável com o usuário.

- (R2) Portabilidade: Deve ser possível executá-lo em diferentes cenários e ambientes.
- (R3) Reprodutibilidade: As avaliações de desempenho não devem mudar de uma execução para outra e ainda devem apresentar resultados comparáveis.
- (R4) Autonomia: O *framework* deve prover mecanismos que automatizem o processo de avaliação de desempenho.
- (R5) Significância: Os cenários de bancos de dados e as cargas de trabalho devem ser representativas.
- (R6) Extensibilidade: Os usuários podem estender o *framework*, os cenários de avaliação e as análises de acordo com suas necessidades.

O conceito de fácil utilização em R1 depende das capacidades esperadas para o usuário que utilizará o *framework*. No contexto deste trabalho, é esperado que os usuários sejam administradores de infraestruturas computacionais e de bancos de dados. Uma vez que esses usuários estão geralmente familiarizados com linhas de comando e alguma linguagem de programação, o *framework* Storiks foi projetado para prover uma interface de comandos e subcomandos de terminal bem documentada, além de uma interface de programação Jupyter Notebook (em Python3) acessível via navegador de Internet.

Considerando o requisito R2, optou-se por utilizar *containers* baseados em Docker [33] para encapsular a maior parte dos programas e bibliotecas que compõem o Storiks. Containers têm sido comumente utilizados em diversas aplicações em nuvem justamente para evitar as quebras indesejáveis de dependência entre aplicações e bibliotecas externas, o que também atende em parte o requisito R5. Uma vez que *containers* dependem do sistema operacional da máquina hospedeira, a implementação do *framework* Storiks se restringiu à ABI do Linux x86_64, largamente utilizada atualmente (R5). Para facilitar o processo de inicialização do *framework* (R1), a maior parte de seus componentes foram "empacotados" em uma imagem de *container* disponibilizada no Docker Hub [57].

A reprodutibilidade (R3) de uma avaliação de interferência de desempenho depende que as mesmas cargas de trabalho sejam executadas nas mesmas condições de *software* e *hardware*. O *framework* Storiks foi projetado para reproduzir as mesmas cargas de trabalho para as mesmas especificações de experimentos. Contudo, sua atual implementação não garante que o ambiente de teste seja o mesmo, especialmente no tocante aos respectivos estados internos do motor de armazenamento chave-valor e do dispositivo de armazenamento em flash. Embora o projeto atual deste *framework* seja capaz de prover detalhes suficientes para avaliar essas condições, o usuário deve preparar e monitorar cuidadosamente cada experimento.

Os mecanismos de automação (R4) considerados para o projeto do Storiks consistem na automação da inicialização e destruição dos *containers* relacionados com cada experimento, além de um mecanismo de agendamento para a execução sequencial de uma fila de experimentos. Uma vez que um experimento de avaliação de interferência de desempenho pode levar vários minutos ou horas para completar, o mecanismo de agendamento permite ao usuário estabelecer uma lista de experimentos a ser executada, controlando o estado de término de cada um deles (sucesso ou falha).

O requisito de significância (R5) foi abordado em diferentes aspectos no projeto do *framework* Storiks, tal como o uso de *containers* Docker, a escolha do motor de armazenamento chave-valor, e as cargas de trabalho utilizadas. Conforme mencionado anteriormente, *containers* Docker são bastante utilizados em infraestrutura de nuvem. Sendo um método de virtualização em nível de sistema operacional, *containers* consomem muito menos recursos que as tradicionais

máquinas virtuais, tornando-os bastante atrativos para a consolidação de várias aplicações em um mesmo *hardware*.

Uma vez que *containers* e MVs provêm diferentes níveis de abstração para os recursos compartilhados, existe a possibilidade de que cada uma dessas abordagens produza diferentes comportamentos em relação à interferência de desempenho causada por cargas de trabalho concorrentes compartilhando o mesmo dispositivo de armazenamento. Contudo, a extensão do *framework* Storiks para o uso de MVs, além de uma análise comparativa dos padrões de interferência de desempenho entre essas duas abordagens pode ser considerada em trabalhos futuros.

O motor de armazenamento chave-valor escolhido para incorporar o *framework* Storiks foi o RocksDB [43] devido à sua representatividade na indústria e academia [29, 30, 19, 85, 32]. Este motor tem sido largamente utilizado em diversas aplicações em produção na Yahoo!, Meta/Facebook e LinkedIn [40, 42]. Implementado como uma biblioteca C++, o RocksDB é um projeto de código aberto que pode ser facilmente incorporado em projetos maiores. Com base em tal extensibilidade, foi possível implementar um mecanismo para monitorar o estado interno da árvore LSM usada por este motor para armazenar o banco de dados chave-valor. Esta implementação permite que o *framework* Storiks registre este estado durante a execução de cada experimento para análises posteriores.

Uma vez que o objetivo do *framework* Storiks é avaliar a interferência de desempenho sofrida por um motor de armazenamento chave-valor, existem dois tipos de cargas de trabalho a serem consideradas tanto pelo requisito R5 como pela Definição 1: (1) carga de trabalho submetida ao motor de armazenamento, chamada de **carga de trabalho primária**, e (2) **cargas de trabalho concorrentes**. Em vez de implementar um conjunto próprio de cargas de trabalho chave-valor, o Storiks foi projetado para incorporar dois utilitários para a geração dessas cargas: YCSB [25] e db_bench [43]. O YCSB é amplamente utilizado em trabalhos acadêmicos para avaliar o desempenho de sistemas de bancos de dados NoSQL e chave-valor, sendo uma condição estreitamente alinhada com o requisito R5. O db_bench, por sua vez, é parte do projeto do RocksDB e pode exercitar diversas funcionalidades específicas desse motor de armazenamento.

Como cargas de trabalho concorrentes, este estudo considera principalmente aquelas que sejam intensivas em leitura e escrita no dispositivo de armazenamento, de baixo consumo de CPU e memória e que apresentem diferentes padrões de acesso, como diferentes proporções de leitura/escrita e números de acessos concorrentes (iodepths). Tais características têm sido exploradas por alguns trabalhos relevantes e *microbenchmarks* desenvolvidos para avaliar o desempenho de dispositivos de armazenamento em memória flash [47, 74, 86, 16]. Para produzir diferentes padrões de leitura e escrita, foi implementado para o *framework* Storiks um *microbenchmark* chamado `access_time3`, o qual será descrito na Seção 3.4.2.

A extensibilidade (R6) considerada para a versão atual do *framework* Storiks abrange os diversos parâmetros de configuração por ele suportados, a possibilidade de se combinar diversos tipos diferentes de cargas de trabalho, e o ambiente de programação provido pelo servidor Jupyter Notebook para a elaboração de análises de dados. Embora o YCSB seja suficientemente extensível para suportar diversos SGBDs, a implementação atual do Storiks restringe sua compilação para apenas incluir o RocksDB. Esta restrição foi necessária para controlar melhor o estado interno deste motor de armazenamento durante os experimentos e para diminuir o tamanho da imagem de *container*. De qualquer modo, a extensão deste projeto para outros SGBDs e motores de armazenamento chave-valor pode ser realizada futuramente por meio da alteração nos comandos de compilação do YCSB dentro do código fonte do Storiks.

3.3 ARQUITETURA E COMPONENTES

O nome Storiks é uma abreviação para o termo "***Sto**rage **R**esource **I**nterferometer for **K**ey-Value **S**tores*", cuja arquitetura é apresentada na Figure 3.1. Embora o Storiks use apenas uma imagem de *container*, esta é instanciada múltiplas vezes para desempenhar diferentes tarefas, classificadas como Controle (*Controller container*) e Experimentos (*Experiment containers*). O *container* de controle é responsável por agendar os experimentos, gerenciar seus respectivos ciclos de vida e coletar estatísticas de sistema e das cargas de trabalho. Já os *containers* de experimentos são responsáveis por executar as cargas de trabalho de cada experimento.

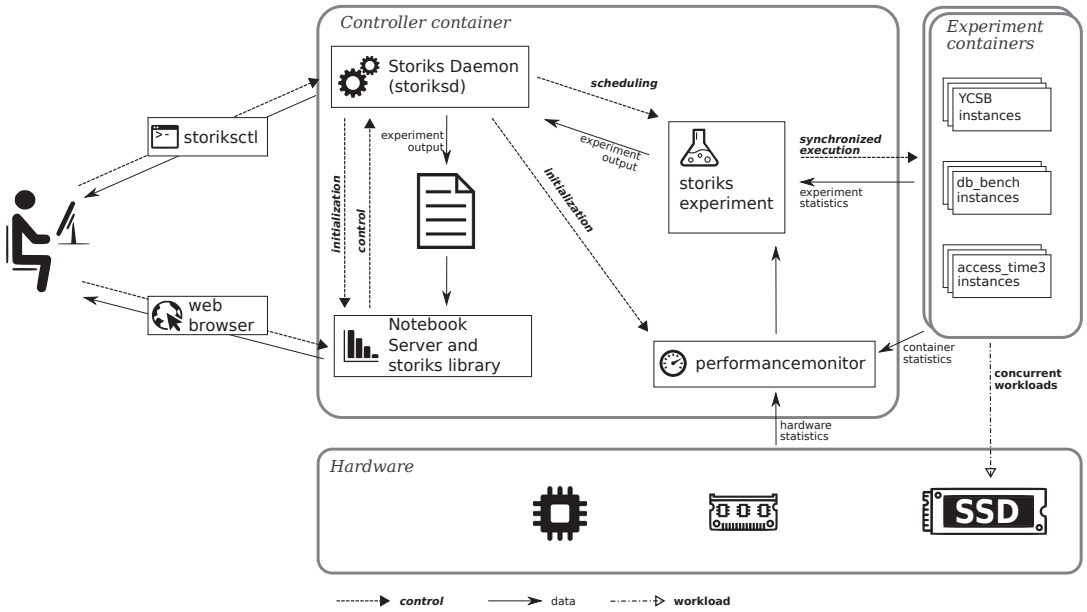


Figura 3.1: Arquitetura do *framework* Storiks.

A seguinte lista descreve os principais componentes do *framework* Storiks:

- Storiks Control (`storiksctl`) - É responsável por iniciar o *container* de controle e enviar comandos para o `storiksd` via terminal de comando.
- Storiks Daemon (`storiksd`) - É responsável por inicializar o ambiente interno do *container* de controle, incluindo seus principais processos, e controlar o agendamento dos experimentos.
- Experimento Storiks (programa `storiks`) - Controla o ciclo de vida de cada experimento, iniciando um *container* para cada carga de trabalho e reportando estatísticas de desempenho. Cada instância deste programa corresponde a um experimento.
- Performance Monitor (`performancemonitor`) [58] - Monitora e gera estatísticas sobre o sistema operacional, os *containers* e o *hardware* durante cada experimento.
- Servidor Jupyter Notebook [53] - É usado como uma interface Web que permite que os usuários controlem os experimentos e gerem gráficos sobre os arquivos de saída dos respectivos experimentos usando a biblioteca Storiks.
- Biblioteca Storiks (*Storiks Library*) - Conjunto de classes e funções usadas para se comunicar com o `storiksd` e para interpretar os arquivos de saída gerados pelos experimentos.

O `storiksctl` é o único componente a ser executado fora de *containers*. Implementado em Python 3, este programa foi projetado para ser uma interface em linha comando cujos objetivos principais são inicializar o *container* de controle e enviar comandos ao `storiksd`. Cada funcionalidade do `storiksctl` é acessível por meio de subcomandos cujas descrições podem ser obtidas por meio do parâmetro "`--help`".

O subcomando `storiksctl start` inicia o *container* de controle e o `storiksd`. Primeiramente, este subcomando faz o *download* da imagem do *framework* Storiks hospedada no Docker Hub [57], coleta algumas informações sobre o sistema hospedeiro e realiza uma série de validações de consistência sobre o mesmo e os argumentos passados pelo usuário. O usuário deve especificar onde o dispositivo de armazenamento a ser testado está montado ou um subdiretório do mesmo (parâmetro `--data_dir`) e o caminho usado para guardar os arquivos de saída dos experimentos (`--output_dir`). Após as validações de consistência, o `storiksctl` pode iniciar o *container* de controle com os respectivos mapeamentos de volume e as variáveis de ambiente requeridas pelo `storiksd`. Nesta fase, o `storiksctl` também atribui o *container* de controle como sendo privilegiado (*privileged*) devido a diversas permissões requeridas pelo `performancemonitor` e `storiksd`.

Outro subcomando importante é o `storiksctl send`, o qual encaminha comandos de controle ao `storiksd` através de um *socket* Unix compartilhado entre o sistema hospedeiro e o *container* de controle. Exceto o parâmetro "`--help`", todos os demais argumentos são repassados na íntegra e interpretados pelo `storiksd` sem a validação do `storiksctl`. O argumento `help` (sem hífens) lista os comandos reconhecidos pelo `storiksd`. Outros subcomandos relevantes do `storiksctl` são:

- `open`, o qual abre o navegador de Internet do sistema hospedeiro com a URL gerada pelo servidor Jupyter Notebook instanciado dentro do *container* de controle;
- `status`, o qual imprime o estado atual do *container* de controle e a URL do servidor Jupyter Notebook; e
- `terminal`, o qual abre um terminal de comando dentro do *container* de controle.

O Storiks Daemon (`storiksd`) é o principal processo em execução dentro do *container* de controle. Suas atribuições mais importantes são configurar o ambiente inicial deste *container* e as respectivas permissões de usuário, iniciar os processos auxiliares, tais como o servidor Jupyter Notebook e o `performancemonitor`, além de agendar e executar os experimentos submetidos pelo usuário. O *socket* Unix fornecido pelo `storiksd` age como uma interface de comunicação com o `storiksctl` e o servidor Jupyter Notebook. Neste último caso, a comunicação com o `storiksd` é controlada pela biblioteca Storiks, implementada em Python 3.

Os experimentos de interferência de desempenho são agendados pelo `storiksd` para execução sequencial. Cada experimento corresponde a uma instância do programa `storiks`, codificado em C++17, cujos parâmetros são especificados pelo usuário. O `storiksd` sempre verifica a existência de apenas uma instância do programa `storiks` em execução por vez, armazenando seu respectivo arquivo de saída no diretório especificado pelo parâmetro "`--output_dir`" do subcomando `storiksctl start`.

Para especificar um experimento, o usuário pode informar os parâmetros para o programa `storiks` usando uma lista de argumentos ou especificar um arquivo *flagfile* [45]. Os *flagfiles* são arquivos texto compostos por um parâmetro por linha, contribuindo assim para a reprodutibilidade dos experimentos (R3). Além disso, esses arquivos podem referenciar internamente outros arquivos *flagfile*, permitindo ao usuário criar uma cadeia hierárquica de arquivos para reduzir o número de parâmetros repetidos entre experimentos.

Cada instância do programa `storiks` é executada pelo `storiksd` dentro do *container* de controle. De acordo com os respectivos parâmetros informados pelo usuário, este programa inicia um *container* de experimento para cada carga de trabalho concorrente usando a mesma imagem do *container* de controle. Os processos em execução dentro de cada *container* de experimento são específicos de cada carga de trabalho, cujos estados são monitorados pelo programa `storiks`. Os detalhes sobre cada carga de trabalho serão descritas mais adiante na Seção 3.4.

Durante cada experimento, o programa `storiks` coleta estatísticas de desempenho providas por cada carga de trabalho, além de estatísticas do sistema hospedeiro fornecidas pelo `performancemonitor`. Esses dados são então enviados para o `storiksd`, os quais são armazenados cronologicamente dentro do arquivo de saída do experimento para análises posteriores usando o servidor Jupyter Notebook interno e a biblioteca Storiks.

O `performancemonitor` foi implementado como um projeto auxiliar para reportar as medições de desempenho do *hardware*, sistema operacional e *containers*. Seu código fonte está disponível em um repositório aparte do projeto Storiks [58]. Inicializado pelo `storiksd` junto com o servidor Jupyter Notebook interno, o `performancemonitor` consiste em um programa codificado em Python 3 que lê periodicamente um grande número de contadores de desempenho disponibilizados pelo sistema operacional através dos pseudo-sistemas de arquivos `/proc`, `/sys` e `/sys/fs/cgroup`. Para permitir o acesso a esses dados, o `storiksctl` mapeia esses sistemas de arquivos dentro do *container* de controle durante sua inicialização. Em cada experimento, o programa `storiks` periodicamente usa uma porta *socket* criada pelo `performancemonitor` para requisitar as medições de desempenho do sistema hospedeiro.

As medições de desempenho fornecidas pelo `performancemonitor` são extensas e podem abranger mais de 400 colunas, dependendo do *hardware* e do experimento realizado. Tais dados incluem os contadores de CPU disponíveis pelo kernel Linux em `/proc/stat` e os contadores do E/S disponíveis em `/proc/diskstats` para o dispositivo de armazenamento avaliado. O `performancemonitor` também usa o programa `smartctl` para coletar informações adicionais sobre este dispositivo de armazenamento. Dependendo do modelo do dispositivo, essas informações podem incluir o volume de dados efetivamente lidos e gravados em memória flash, bem como a quantidade de blocos lógicos com dados associados dentro do dispositivo.

Uma vez que cargas de trabalho concorrentes são executadas em *containers* diferentes em cada experimento, o `performancemonitor` também coleta informações de desempenho de leitura e escrita desses *containers* sobre o dispositivo de armazenamento avaliado. Essas informações são coletadas por meio do módulo `blkio` do subsistema `cgroup` do kernel Linux (`/sys/fs/cgroup/blkio`).

O *framework* Storiks também provê um servidor Jupyter Notebook interno que funciona como uma interface web para o usuário. Este servidor é iniciado dentro do *container* de controle pelo `storiksd` e inclui diversos pacotes Python bastante utilizados para análise de dados e geração de gráficos, tais como NumPy [7], Pandas [8], Seaborn [9] e Matplotlib [6]. Este servidor também acompanha a biblioteca Storiks, a qual disponibiliza uma série de rotinas necessárias para controlar os experimentos e manipular seus respectivos arquivos de saída.

A biblioteca Storiks é um pacote Python 3 composto por dois componentes principais: `run`, o qual se comunica com o `storiksd` e é usado para agendar, listar e cancelar experimentos; e `plot`, o qual produz diversos gráficos e estruturas de dados a partir do arquivo de saída de cada experimentos ou de um conjunto destes. No repositório do projeto Storiks, existem diversos exemplos de como usar esses dois módulos [59]. Além disso, esses exemplos também são

copiados pelo `storiksd` dentro do diretório de saída dos experimentos durante a inicialização do *container* de controle.

O módulo `plot` da biblioteca Storiks implementa um grande número de rotinas que podem ser usadas para interpretar o arquivo de saída de cada experimento, gerar uma série de estruturas de dados intermediárias e produzir mais de dezenove gráficos de desempenho diferentes, tanto para experimentos individuais, como para grupos de experimentos. O Capítulo 4 apresenta os gráficos mais relevantes desta biblioteca usados para avaliar a pergunta de pesquisa e hipótese deste trabalho. Embora as rotinas implementadas neste módulo sejam importantes para analisar este cenário proposto, os usuários não são limitados a elas. Uma vez que o Jupyter Notebook é um ambiente de programação bastante flexível e robusto, essas funcionalidades podem ser facilmente estendidas para atender novos gráficos e análises estatísticas (requisito R6).

3.4 CARGAS DE TRABALHO

As cargas de trabalho possuem um papel essencial no *framework* Storiks, cujas principais características são descritas nesta seção. De acordo com a Seção 3.1, as cargas de trabalho necessárias para este ambiente de teste são divididas em duas classes:

- *Carga de trabalho primária* – a qual representa a carga de trabalho submetida ao motor de armazenamento chave-valor; e
- *Cargas de trabalho concorrentes* – as quais correspondem às cargas de trabalho usadas para estressar o dispositivo de armazenamento ao mesmo tempo que a carga de trabalho primária está em execução.

As subseções seguintes detalham os aspectos relevantes desses dois tipos de cargas de trabalho incluindo suas respectivas implementações.

3.4.1 Carga de Trabalho Primária

Considerando a Definição 1, uma carga de trabalho primária é aquela submetida ao motor de armazenamento chave-valor cujo desempenho representa o critério principal de análise (C). Embora o programa `storiks` possa executar simultaneamente vários motores de armazenamento chave-valor com diferentes cargas de trabalho, apenas o primeiro motor de armazenamento é considerado para este critério, o que é suficiente para avaliar a hipótese deste trabalho. As demais cargas de trabalho são consideradas como cargas concorrentes pelo *framework* Storiks (D).

Antes de detalhar esta classe de cargas de trabalho e as cargas de trabalho concorrentes (Seção 3.4.2), faz-se necessário discutir algumas condições relacionadas com as definições introduzidas na Seção 3.1. Conforme observa-se a partir da Definição 3, a função $\overline{\rho}(d_i)$ requer que apenas d_0 e d_i variem durante as medições de $\rho(d_0)$ e $\rho(d_i)$. Embora seja um requisito aparentemente fácil, existem diversos detalhes importantes a serem considerados durante a obtenção de cada um desses valores de desempenho.

A maior parte da complexidade da Definição 3 está relacionada com a estabilidade e reprodutibilidade do ambiente computacional ao medir cada $\rho(d_i)$. Caso o desempenho do motor de armazenamento chave-valor tenda a aumentar ou diminuir ao longo do tempo sem que d_i mude, outras variáveis não consideradas estão provavelmente influenciando esta tendência de mudança de desempenho, o que pode levar a comparações incorretas de ρ se essas variáveis não puderem ser reproduzidas. Existem diversas condições relacionadas com sistemas de banco de dados,

sistemas operacionais e *hardware* que podem influenciar esta estabilidade e reprodutibilidade. A seguinte lista descreve algumas das condições mais relevantes, tendo em vista a arquitetura do motor de armazenamento chave-valor e o tipo de *hardware* utilizados neste trabalho:

- *Composição da carga de trabalho.* Tipos diferentes de transações podem exercitar diferentes mecanismos internos de um motor de armazenamento chave-valor ou mesmo requerer diferentes recursos computacionais, o que pode consequentemente influenciar o desempenho alcançado por este motor.

- *Cache do sistema operacional e de banco de dados.* Logo após iniciar um sistema de bancos de dados, muitos dos dados que são sensíveis ao desempenho do mesmo ainda não foram carregados do dispositivo de armazenamento para a memória. Esta falta de dados nos diferentes mecanismos de *buffer* e *cache* podem reduzir inicialmente o desempenho do sistema avaliado, razão pela qual muitas ferramentas de *benchmark* realizam algum tipo de "aquecimento" prévio (*warm-up*) antes de iniciar as medições de desempenho.

- *Tamanho do banco de dados.* O tamanho do banco de dados influencia a razão entre a memória disponível e os dados contidos em armazenamento persistente. À medida que o banco de dados aumenta, a memória necessária para armazenar os filtros de Bloom e outras estruturas importantes aumenta, reduzindo a memória disponível para outras operações do motor de armazenamento. Além disso, este aumento no tamanho do banco de dados também pode gradativamente requerer níveis adicionais na árvore LSM, aumentando a amplificação de escrita produzida pelas operações de compactação.

- *Estado interno do banco de dados.* Devido a algumas otimizações realizadas durante o processo de carga ou restauração de backup de um banco de dados, a distribuição desses dados na árvore LSM logo após um processo como este pode não condizer com a configuração esperada para um motor de armazenamento em produção. Caso os dados tenham sido carregados inicialmente no último nível da árvore, o motor de armazenamento chave-valor tende a apresentar um desempenho inicial superior tanto de leitura (uma vez que as consultas podem ser realizadas diretamente no último nível), como de escrita (visto não existem muitos dados para serem compactados entre os níveis intermediários). Caso os dados iniciais estejam armazenados nos primeiros níveis, o desempenho inicial tende a ser menor, visto que o motor de armazenamento precisa ainda realizar diversas operações de compactação até que os dados estejam devidamente acomodados nos níveis maiores.

- *Estado interno do dispositivo de armazenamento.* Diferente dos tradicionais HDDs, o desempenho de um dispositivo flash é particularmente influenciado por seu estado interno [32, 83, 77]. A maior parte desta influência é causada pela combinação de requisições de escrita e o número de páginas flash livres no dispositivo. Uma vez que cada página precisa ser apagada antes de ser escrita, essas operações impactam menos o desempenho quando muitas páginas livres estão disponíveis para persistir os dados recebidos. À medida que o número dessas páginas livres diminui, a FTL é pressionada a realizar mais coletas de lixo para disponibilizar essas páginas, reduzindo assim o desempenho do dispositivo neste processo. Ao avaliar este tipo de dispositivo de armazenamento, uma prática comum é manter a proporção de páginas flash livres o mais estável possível para evitar oscilações de desempenho durante os experimentos.

Considerando as prováveis fontes de variação de desempenho citadas acima, é possível enumerar dois requisitos principais para uma correta medição de $\rho(d_i)$:

- **WR1**: A carga de trabalho sobre o motor de armazenamento chave-valor precisa ser estável do ponto de vista do tamanho lógico do banco de dados e tipos de transações.
- **WR2**: Os estados internos do motor de armazenamento, do SO e dispositivo de armazenamento também devem permanecer estáveis durante o experimento.

O primeiro requisito está diretamente relacionado com as características da carga de trabalho primária, enquanto que o segundo tem relação com o motor de armazenamento e o ambiente experimental.

Conforme descrito na Seção 3.2, o *framework* Storiks incorpora o YCSB e o db_bench como geradores de cargas de trabalho. Embora todas as cargas de trabalho pré-definidas (*core workloads*) do YCSB sejam suficientemente estáveis do ponto de vista de tipos de transação, apenas as cargas listadas abaixo são elegíveis para serem cargas de trabalho primárias, uma vez que elas mantém estável o tamanho lógico do banco de dados:

- Carga de trabalho A (`workloada`): 50% de leituras (transações *get* ou *point lookups*) e 50% de atualizações (transações *put*);
- Carga de trabalho B (`workloadb`): 95% de leituras e 5% de atualizações;
- Carga de trabalho C (`workloadc`): 100% de leituras; e
- Carga de trabalho F (`workloadf`): 50% de leituras e 50% de atualizações após a leitura da mesma chave.

Em contrapartida, as cargas de trabalho D e E do YCSB não satisfazem WR1 uma vez que elas realizam inserções de novas chaves no banco de dados, aumentando o tamanho do mesmo durante o experimento e potencialmente afetando gradativamente o desempenho do motor de armazenamento chave-valor.

Como parte do projeto RocksDB, a ferramenta db_bench implementa mais de quarenta cargas de trabalho diferentes, desenvolvidas para avaliar diferentes aspectos deste motor de armazenamento. Devido ao grande número de cargas de trabalho e o tempo de vida limitado dos dispositivos flash avaliados, este trabalho se restringiu a avaliar um conjunto bastante restrito delas. A seguinte lista apresenta as cargas de trabalho avaliadas que satisfazem o requisito WR1:

- `readrandomwriterandom`: N *threads* executando leituras e atualizações aleatórias; e
- `readwhilewriting`: 1 *thread* executando atualizações aleatórias e N *threads* executando leituras aleatórias.

Para ambas as cargas de trabalho do db_bench listadas acima, o usuário pode especificar o número de *threads* (N) usando o parâmetro "`--db_threads`" do programa `storiks`. A carga de trabalho `readrandomwriterandom` segue os mesmos conceitos das cargas de trabalho A, B, C e F do YCSB. Neste caso, o percentual de leituras e atualizações submetidas ao motor de armazenamento chave-valor é controlado pelo parâmetro "`--db_readwritepercent`". Contudo, a implementação da carga de trabalho `readrandomwriterandom` diverge do YCSB em alguns aspectos, especialmente com relação ao motor de geração de chaves aleatórias e

no controle de *threads* que simulam os clientes do banco de dados. Por esta razão, o desempenho alcançado pelo db_bench não necessariamente equivale ao desempenho do YCSB, mesmo usando parâmetros de configuração equivalentes em ambos os *benchmarks*.

O segundo requisito (WR2) descreve que os estados internos do motor de armazenamento chave-valor, do SO e do dispositivo de armazenamento também precisam ser suficientemente estáveis para que o desempenho da carga de trabalho primária também permaneça estável por longos períodos. Conforme mencionado anteriormente, a *cache* do sistema operacional, o tamanho dos níveis da árvore LSM e o número de páginas flash livres e ocupadas no dispositivo de armazenamento são potenciais causas de alterações de desempenho. Mesmo assumindo que o requisito WR1 seja atendido, manter as condições de estabilidade descritas em WR2 depende do ambiente experimental, incluindo a definição do tempo de *warm-up* para carga de trabalho primária e a preparação do dispositivo de armazenamento, como particionamento e pré-condicionamento do sistema de arquivos.

Em sua implementação atual, o *framework* Storiks não pode garantir que WR2 seja atendido antes de realizar as medições de $\rho(d_i)$. Em vez disso, o usuário deve prepara o ambiente de teste cuidadosamente e verificar tais condições antes e depois dos experimentos. Contudo, as estatísticas de desempenho coletadas pelo programa `storiks` e pelo `performancemonitor` são bastante importantes para auxiliar o usuário neste processo de validação. O ambiente experimental usado neste trabalho para satisfazer o requisito WR2 será descrito no Capítulo 4.

3.4.2 Cargas de Trabalho Concorrentes e o Programa Access_time3

As cargas de trabalho concorrentes (D) são fundamentais nesta análise de interferência de desempenho, uma vez que precisa-se estabelecer uma relação entre D e C, a qual é representada na Definição 1 pela função ρ. Esta seção apresenta o programa `access_time3`, um *microbenchmark* flexível desenvolvido para gerar cargas de trabalho concorrentes, além dos principais aspectos considerados para produzir as cargas de trabalho utilizadas neste estudo.

3.4.2.1 Características e Abstrações dos Dispositivos de Armazenamento

Um dos principais aspectos considerados para o projeto e desenvolvimento do `access_time3` é a formas de representação e endereçamento de dados nos diferentes níveis de abstração desde o dispositivo de armazenamento até o espaço de usuário provido pelo sistema operacional para a execução de cada processo. Em primeiro lugar, tanto HDDs como SSDs são tradicionalmente dispositivos de bloco (*block devices*), de modo que os dados somente podem ser endereçados, lidos e gravados nesses dispositivos em uma granularidade geralmente muito maior que um único byte. Portanto, o `access_time3` foi projetado para realizar operações de leitura e gravação de blocos de dados em vez de acessá-los a nível de bytes individuais.

A noção de blocos, seus respectivos tamanhos e espaços de endereçamento pode envolver vários níveis de abstração dependendo do dispositivo de armazenamento e da configuração do sistema operacional. Entre SO e dispositivo de armazenamento, as operações de leitura e gravação são feitas em nível de blocos ou setores[1] lógicos. Embora alguns dispositivos permitam modificar este tamanho, os blocos lógicos possuem geralmente 512 bytes, os quais são enumerados de 0 até o número correspondentes de blocos disponíveis no dispositivo menos um. Tais índices de blocos lógicos são chamados de *Logical Block Addresses* (LBAs).

Do ponto de vista físico, cada dispositivo possui uma unidade mínima para operações de leitura e gravação. Podendo abranger um ou mais blocos lógicos, este tamanho de bloco físico

[1] O termo "setor" faz parte de uma terminologia herdada dos tradicionais HDDs.

varia de acordo com o projeto do fabricante e a tecnologia empregada. Nos HDDs tradicionais, tais blocos correspondem aos setores físicos do dispositivo, cujo tamanho é geralmente 512 bytes ou 4 KiB, embora isto não seja considerado uma regra. Tais dispositivos informam este tamanho ao sistema operacional, sendo que no Linux este tamanho de bloco físico pode ser consultado por meio do pseudo arquivo "`/sys/block/*/queue/physical_block_size`" de cada dispositivo de bloco.

Nos SSDs, a noção de blocos físicos é ainda mais complexa e menos transparente devido à grande quantidade de mecanismos de *software* e *hardware* empregada entre as requisições do sistema operacional e as operações efetivamente realizadas sobre a memória flash. Conforme descrito no Capítulo 2, este tipo de memória permite leituras e gravações em nível de página, cujo tamanho pode variar enormemente de um modelo para outro. Embora alguns modelos de SSDs empresariais de alto desempenho utilizem 256 bytes como tamanho de página, uma tendência atual é que este tamanho seja de 16 KiB, o que equivale a 32 LBAs de 512 bytes.

Embora muitos modelos de SSDs atuais utilizem tamanhos relativamente grandes para páginas flash (ex., 16 KiB), tal granularidade mínima de leitura e gravação não é adequada para uma grande variedade de aplicações [75, 82, 96], o que provavelmente pressionaria um aumento significativo da amplificação de escrita e comprometeria a vida útil desses dispositivos. Por meio do uso da FTL, os efeitos desta alta granularidade são geralmente amortizados por diversas outras otimizações internas que permitem subdividir o espaço de endereçamento dessas páginas. Além disso, outras otimizações implementadas na FTL e nos *dies* permitem que múltiplas páginas possam ser lidas e gravadas simultaneamente [76]. Contudo, tais detalhes de implementação não são abertamente divulgados pela maioria dos fabricantes de dispositivos flash.

Apesar de essencial para os dispositivos SSDs controlarem toda a dinâmica de leitura, escrita e limpeza de páginas flash, a indireção entre blocos lógicos e físicos provida pela FTL também pode ocultar detalhes sobre a localização física dos dados que possam exercer influência sobre o desempenho desses dispositivos. Uma vez que os LBAs são constantemente remapeados para páginas flash diferentes a cada operação de escrita e eventualmente remapeados por operações de coleta de lixo, o dinamismo deste cenário implica que o caminho interno necessário para ler cada LBA (*channel*, *package*, *die* e *plane*) pode variar a cada remapeamento. O impacto deste processo no desempenho do dispositivo depende da carga de trabalho submetida e de sua organização e paralelismo internos (geralmente não revelados pelo fabricante).

Embora muitos detalhes estejam inacessíveis aos usuários finais, algumas especificações industriais relevantes, como a *NVM Express Base Specification* [72], vêm sendo propostas com o objetivo de permitir que fabricantes informem de maneira padronizada a melhor forma de utilizar cada modelo de dispositivo de armazenamento. Particularmente, a especificação de SSDs NVMe é bastante flexível, permitindo que o mesmo seja subdividido em um ou mais espaços de LBAs distintos, chamados de *namespaces*. Cada modelo de dispositivo pode apresentar ao usuário uma lista de possíveis tamanhos de LBAs (ex., 512 bytes e 4 KiB), incluindo ainda um valor de preferência de desempenho para cada um desses tamanhos de blocos lógicos.

Dependendo da versão, a especificação NVMe prevê também algumas informações bastante relevantes sobre a granularidade e alinhamento recomendados para as operações de E/S em cada *namespace*, embora estas informações não sejam obrigatoriamente fornecidas pelos fabricantes:

- *Namespace Optimal I/O Boundary* (NOIOB) – Estabelece, em número de LBAs, o alinhamento recomendado para as operações de E/S. De acordo esta especificação, as operações de E/S enviadas pelo SO hospedeiro não devem cruzar este limite para obter um desempenho ótimo (a partir da versão 1.3);

- *Namespace Preferred Write Granularity* (NPWG) – Indica, em número de LBAs, a menor granularidade recomendada para uma operação de escrita (a partir da versão 1.4);
- *Namespace Preferred Write Alignment* (NPWA) – Alinhamento de escrita recomendado em número de LBAs (a partir da versão 1.4); e
- *Namespace Optimal Write Size* (NOWS) – Estabelece em LBAs o tamanho ótimo de escrita (a partir da versão 1.4).

Além da organização física e lógica dos dados dentro do dispositivo de armazenamento, os sistemas operacionais atuais proveem uma série de opções que permitem subdividir ou agregar esses dispositivos, formando assim camadas adicionais de abstração. Dentre as abstrações mais utilizadas estão as tabelas de partição, os diversos tipos de RAID (*Redundant Array of Independent Disks*) e o gerenciamento de volumes lógicos (*Logical Volume Management*, ou LVM). As tabelas de partição são as formas mais simples e mais utilizadas deste tipo de abstração, uma vez que elas apenas delimitam o primeiro e o último LBA de cada partição dentro do dispositivo de armazenamento. O uso de RAID e LVM é geralmente aplicado abaixo ou acima das tabelas de partição para melhorar o desempenho, a flexibilidade e/ou a confiabilidade deste recurso de armazenamento.

Apesar das diversas opções disponíveis para organizar logicamente um dispositivo de armazenamento, este trabalho considera apenas o uso de tabelas de partição entre o dispositivo e o sistema de arquivos descrito a seguir. Embora tal simplificação não afete o projeto do `access_time3`, ela foi considerada suficiente para avaliar a hipótese proposta. Avaliações adicionais de interferência de desempenho usando RAID com múltiplos dispositivos ou o uso de LVMs poderão ser tradadas futuramente em por outros trabalhos com o auxílio deste *framework*.

Tanto o espaço de endereçamento proporcionado pelos blocos lógicos de um dispositivo de armazenamento como suas organizações internas descritas acima são inadequados para a maioria das aplicações que usam recursos locais de armazenamento persistente. Em vez disso, tais aplicações são desenvolvidas para utilizarem sistemas de arquivos gerenciados pelo SO como uma forma muito mais prática de abstração sobre os LBAs. Um sistema de arquivos provê diversas funcionalidades adicionais importantes, como gerenciamento de espaço livre, indexação, gerenciamento de metadados e controle de permissão e concorrência.

Os sistemas de arquivos são tradicionalmente organizados na forma de uma árvore de diretórios. Cada diretório pode conter nenhum ou vários arquivos responsáveis pelo armazenamento de dados. Apesar de haverem diversos sistemas de arquivos disponíveis atualmente, este trabalho traz um foco maior sobre a terminologia e estruturas básicas do *ext4* (*fourth extended filesystem*) [87], um dos sistemas de arquivos mais populares em ambientes GNU/Linux. Outro sistema de arquivos bastante relevante em centros de dados é o XFS [90]. Devido às similaridades estruturais desses sistemas de arquivos, especialmente na forma de organização dos dados de cada arquivo, apenas uma breve apresentação do ext4 é suficiente para exemplificar como esta camada de abstração influenciou o desenvolvimento do `access_time3` e as cargas de trabalho concorrentes utilizadas neste trabalho.

Cada arquivo no ext4 corresponde a um *inode*, o qual é responsável por armazenar diversos metadados, incluindo tipo, permissões e tamanho [88]. Os dados de cada arquivo são armazenados diretamente no inode quando o espaço requerido por ele é inferior a 61 bytes (*inline data*). Caso o tamanho dos dados seja maior que 60 bytes, o ext4 utiliza *blocos de dados* (*data blocks*) para armazená-los, os quais são referenciados pelo inode usando uma das seguintes estruturas de dados:

- Uma lista de 15 ponteiros diretos e indiretos para esses blocos; ou

- Uma árvore de extensões (*extent tree*) que usa pares de ponteiro e comprimento para representar os blocos contíguos do arquivo.

O tamanho de cada bloco de dados é configurado quando o sistema de arquivos é criado para uma determinada partição. Por padrão, este valor é de 4 KiB e representa 8 blocos lógicos de 512 bytes.

Esta indireção adicional criada entre sistema de arquivos e o espaço de endereçamento de blocos lógicos (LBAs) pode proporcionar alguns problemas de alinhamento e fragmentação dependendo da tecnologia do dispositivo de armazenamento. A fragmentação tende a ocorrer quando os blocos de dados usados de um arquivo não são contíguos em relação aos respectivos LBAs do dispositivo. No caso dos HDDs, esta fragmentação era indesejada devido ao aumento da latência de posicionamento e rotação nas leituras sequenciais de arquivos. Devido a este problema, o ext4 implementa vários mecanismos para evitar esta fragmentação, tornando-o muito eficiente neste quesito em relação a outros sistemas de arquivos.

O alinhamento dos blocos de dados do sistema de arquivos com a granularidade e alinhamento de E/S esperados pelo dispositivo de armazenamento é uma prática importante para maximizar o desempenho de qualquer sistema. Caso contrário, cada operação de leitura ou escrita de um bloco de dados pode envolver mais dados que o necessário. Embora o sistema de arquivos possa lidar internamente com o problema de fragmentação, o alinhamento precisa ser considerado durante a partição do dispositivo [5]. Uma vez que o tamanho dos blocos lógicos pode ser menor que a granularidade de E/S do dispositivo de armazenamento, o primeiro LBA do sistema de arquivos deve coincidir com os limites desta granularidade. Além disso, a formatação do sistema de arquivos deve considerar um tamanho de bloco de dados (4 KiB por padrão) adequado às recomendações estabelecidas pelo dispositivo.

3.4.2.2 Requisitos e Escopo

O escopo e os requisitos considerados importantes para o projeto do `access_time3` e as cargas de trabalho concorrentes se baseiam nos três espaços de endereçamento descritos anteriormente (blocos lógicos, granularidade de E/S do dispositivo de armazenamento e blocos de dados do sistema de arquivos). Segue abaixo a descrição de cada um desses requisitos:

ATR1 - Nível de abstração baseado no sistema de arquivos. Uma vez que a maioria das aplicações usam o sistema de arquivos como camada de abstração para armazenar dados localmente, a implementação do `access_time3` se restringe a esta camada e ao conjunto de APIs disponíveis pelo sistema operacional para acessar o dispositivo de armazenamento. No kernel Linux, a interface entre processos e o sistema de arquivos é abstraída pelo *Virtual File System* (VFS) [89], o qual provê várias chamadas de sistema (*system calls*), incluindo desde funções POSIX básicas, como open(), read() e write(), até funções assíncronas mais avançadas de E/S, como a AIO [89, 3]. Essas duas interfaces foram consideradas no projeto do `access_time3`.

ATR2 - Alinhamento das cargas de trabalho. Embora os dispositivos de armazenamento considerados neste trabalho sejam endereçáveis por bloco, na maior parte dos casos, as chamadas de sistema disponíveis pelo VFS utilizam bytes como unidade de endereçamento. Consequentemente, a tradução dos endereços em nível de byte para os respectivos blocos de dados e LBAs é realizada pelo kernel do sistema operacional.

Para evitar possíveis amplificações de leitura e escrita de dados, as cargas de trabalho concorrentes precisam ser alinhadas com os blocos de dados do sistema de arquivos, assim como este precisa estar alinhado com a granularidade de E/S do dispositivo de armazenamento. O alinhamento entre carga de trabalho e os blocos de dados faz parte do projeto do `access_time3`.

Já o alinhamento entre sistema de arquivos com o dispositivo de armazenamento deve ser observado na elaboração do ambiente de testes, conforme descrito no Capítulo 4.

ATR3 - Cache. Uma forma de reduzir a quantidade de dados lidos e gravados nos dispositivos de armazenamento é explorar as localidades temporal e espacial de determinadas cargas de trabalho por meio do uso de *cache*. O VFS proporciona este tipo de funcionalidade, a qual é ativada por padrão quando um processo requisita a abertura de um arquivo, podendo ser desativada pela *flag* O_DIRECT [70]. Embora as cargas de trabalho utilizadas neste estudo não utilizem *cache*, a opção de ativação e desativação deste mecanismo de *cache* do VFS foi incorporada ao projeto do `access_time3` com o objetivo de torná-lo flexível para utilizações futuras do *framework* Storiks.

ATR4 - Sincronização de Escrita. A sincronização de escrita é um mecanismo importante provido pelo sistema operacional para confirmar que um ou mais *buffers* de dados foram realmente escritos na memória não volátil do dispositivo de armazenamento. Uma vez que este tipo de sincronização é usado pela maioria dos SGBDs para garantir a durabilidade de transações, a ativação e desativação deste mecanismo foi acrescentado ao projeto do `access_time3` para testar seu efeito em cargas de trabalho concorrentes.

Em nível de sistema de arquivos, existem dois modos principais de sincronização definidos pelo padrão POSIX e implementados no kernel Linux [70]:

- *Sincronização de dados e metadados*: Certifica que todos os dados e metadados de um arquivo foram escritos no dispositivo de armazenamento antes de sua confirmação.
- *Sincronização de dados*: Certifica que todos os dados e apenas os metadados necessários para acessá-los foram escritos no dispositivo antes de sua confirmação.

Com o objetivo de evitar a sobrecarga de operações de sincronização de metadados do sistema de arquivos, optou-se por utilizar a sincronização de dados durante o projeto do `access_time3`. Além disso, as cargas de trabalho concorrentes foram projetadas para usarem arquivos de tamanho fixo para evitar que os ponteiros para os blocos de dados sejam alterados durante os experimentos.

Dependendo do conjunto de APIs utilizadas por um determinado processo, o kernel Linux disponibiliza diferentes *system calls* e *flags* de funções para requisitar a sincronização de dados [70]. Durante o projeto do `access_time3`, a *flag* O_DSYNC foi utilizada na abertura do arquivo indicado para receber uma carga de trabalho baseada na função *write()* do padrão POSIX. Já a *flag* RWF_DSYNC foi utilizada em conjunto com a função *io_submit()* quando esta carga de trabalho é produzida com base nas APIs AIO.

ATR5 - Espaço de endereçamento e tipos de requisições. Este trabalho assume que as cargas de trabalho concorrentes produzidas pelo `access_time3` são uma sequência ou um conjunto de múltiplas requisições de E/S executadas simultaneamente. Cada requisição pode ser uma leitura ou escrita, cuja posição e extensão devem ser alinhadas com os blocos de dados do sistema de arquivos.

Conforme mencionado anteriormente, o arquivo utilizado para receber essas operações de E/S deve ser de tamanho fixo para evitar possíveis sobrecargas com os metadados do sistema de arquivos. Por convensão, considera-se que este arquivo tenha sido previamente criado e informado ao `access_time3` durante a inicialização de cada experimento. O tamanho deste arquivo também precisa ser suficientemente grande para acomodar múltiplas requisições de E/S sem a incidência de conflitos.

Para melhorar a flexibilidade do *framework* Storiks, o usuário poderá especificar o tamanho dessas requisições de E/S, as quais devem ser múltiplas do tamanho do bloco de dados

do sistema de arquivos. Uma vez definido o tamanho de cada requisição, o arquivo utilizado para receber a carga de trabalho é dividido logicamente em partes contíguas e não sobrepostas para serem endereçadas pelo `access_time3` durante a geração da carga de trabalho, conforme descrito no requisito ATR7.

ATR6 - Proporção de leituras e escritas. Durante o projeto do `access_time3`, considerou-se a proporção de requisições de leitura e escrita como sendo um parâmetro definido pelo usuário. Diferente da ferramenta de *benchmark* Fio [16], a qual apenas executa cargas de trabalho de somente leitura e somente escrita, este projeto considera um ajuste mais fino dessas proporções por meio da escolha de um valor entre 0 e 1, sendo 0 uma carga de trabalho somente de leitura e 1 uma carga de trabalho somente de escrita. Este tipo de granularidade é semelhante à forma de configuração utilizada pelo YCSB e db_bench para gerar cargas de trabalho chave-valor.

ATR7 - Acessos sequenciais *versus* aleatórios. As cargas de trabalho sequenciais e aleatórias também são consideradas por diversas ferramentas de *benchmark*, sendo que na maior parte dos casos somente acessos puramente sequenciais e aleatórios são implementados. Para melhorar a flexibilidade do *framework* Storiks, o projeto do `access_time3` seguiu o mesmo esquema de proporcionalidade de leituras e escritas para produzir esses tipos de acesso.

Conforme mencionado anteriormente, o arquivo informado pelo usuário é subdividido em partes contíguas e não sobrepostas, as quais são enumeradas de 0 até o número de partes menos 1. Para cargas completamente aleatórias, o `access_time3` seleciona uma dessas partes para receber a próxima requisição de E/S usando uma distribuição uniforme. Nas cargas de trabalho sequenciais, a próxima parte é sempre a parte subsequente do arquivo, retornando eventualmente à parte 0 sempre que este índice de seleção alcançar a última parte do arquivo.

Para valores intermediários, a estratégia utilizada pelo `access_time3` é primeiramente decidir se a requisição subsequente será aleatória ou sequencial de acordo com a proporção informada pelo usuário para esses dois padrões de acesso. Esta escolha é feita usando uma distribuição uniforme entre 0 e 1. Caso o próximo acesso seja aleatório, a mesma rotina anterior para acessos aleatórios é invocada. Caso contrário, aplica-se a rotina de acessos sequenciais para a próxima requisição de E/S.

ATR8 - Níveis de Paralelismo. Uma vez que as APIs fornecidas pelo AIO permitem múltiplas requisições assíncronas de E/S executadas simultaneamente, o projeto do `access_time3` utiliza esta funcionalidade para selecionar vários níveis de paralelismo em uma carga de trabalho concorrente. Conhecida como *iodepth*, esta opção é também parte de outras ferramentas de *benchmark* de E/S, como a Fio. Conforme apresentado no Capítulo 4, esta funcionalidade é essencial para avaliar o paralelismo interno dos dispositivos de armazenamento estudados e sua relação com desempenho do motor de armazenamento chave-valor.

3.4.2.3 Implementação

O `access_time3` foi implementado usando a linguagem C++17. A Tabela 3.1 resume seus principais parâmetros de configuração de acordo com as opções de projeto descritas na Seção 3.4.2.2. Os parâmetros `filename`, `io_engine`, `block_size`, `write_ratio`, `random_ratio`, `iodepth`, `o_direct` e `o_dsync` seguem os requisitos de projeto descritos em ATR1 até ATR8. O parâmetro `duration` define o tempo em segundos que o `access_time3` irá executar. Ao longo de cada experimento, este *microbenchmark* coleta estatísticas de desempenho sobre a carga de trabalho, as quais são reportadas periodicamente ao programa `storiks` de acordo com o intervalo de tempo estabelecido pelo parâmetro `stats_interval`.

Tabela 3.1: Parâmetros de configuração do access_time3.

Parâmetro	Tipo	Descrição
`filename`	string	Caminho do arquivo que será usado para receber a carga de trabalho [ATR1].
`io_engine`	opções	Motores de E/S implementados. Conjunto de APIs usadas para a geração das requisições de E/S. "`posix`" para as funções read() e write() tradicionais do padrão POSIX, e "`libaio`" para as APIs fornecidas pelo AIO [ATR1].
`block_size`	inteiro	Tamanho em KiB de cada requisição de E/S. Precisa ser múltiplo do tamanho do bloco de dados do sistema de arquivos [ATR2, ATR5].
`write_ratio`	decimal	Proporção de escritas na carga de trabalho (entre 0 e 1) [ATR6].
`random_ratio`	decimal	Proporção de acessos aleatórios na carga de trabalho (entre 0 e 1) [ATR7].
`iodepth`	inteiro	Número de requisições simultâneas (entre 1 e 128) [ATR8].
`o_direct`	lógico	Quando ativado, usa a *flag* O_DIRECT para evitar a *cache* do sistema operacional (VFS) [ATR3].
`o_dsync`	lógico	Quando ativo, força a durabilidade de cada requisição de escrita usando a sincronização de dados [ATR4].
`duration`	inteiro	Duração do experimento em segundos.
`stats_interval`	inteiro	Intervalo em segundos entre os relatórios de desempenho.
`command_script`	string	Lista de comandos que permitem agendar mudanças dos parâmetros block_size, write_ratio, random_ratio e iodepth durante a execução do experimento. Cada comando define o momento em segundos, o parâmetro e o valor desejado a ser mudado.

Para permitir maior flexibilidade, o `access_time3` permite a alteração dos parâmetros `block_size`, `write_ratio`, `random_ratio` e `iodepth`[2] em tempo de execução através do envio de comandos pela entrada padrão do processo (*stdin*) ou por agendamento usando o parâmetro `command_script`. A alteração desses parâmetros durante o experimento é uma funcionalidade importante para explorar o espaço de possíveis cargas de trabalho concorrentes, conforme demonstrado no Capítulo 4.

Tanto para os motores de E/S `posix` e `libaio` (parâmetro `io_engine`), o `access_time3` implementa um laço fechado dentro em uma *thread* para realizar as requisições de leitura e escrita no arquivo especificado pelo parâmetro `filename`. Caso `io_engine` seja atribuído para "`posix`", esta *thread* executa as chamadas de sistema *read()* e *write()* de acordo com a proporção especificada pelo parâmetro `write_ratio`, bloqueando a *thread* até que o sistema operacional ative-a novamente após a finalização da chamada. Caso o parâmetro `io_engine` seja igual a "`libaio`", devido às características não bloqueantes das APIs AIO, a mesma *thread* executa requisições usando a função *io_submit()* e periodicamente verifica quais requisições completaram usando a função *io_getevents()*. A quantidade de requisições submetidas simultaneamente ao SO usando a função *io_submit()* depende do parâmetro `iodepth`. Mais detalhes desta implementação estão disponíveis no repositório do projeto [59].

[2] O parâmetro `iodepth` não é utilizado quando `io_engine` for igual a "`posix`".

De acordo com a Figura 3.1, o `access_time3` foi incorporado à imagem de *container* do *framework* Storiks para melhorar a sua portabilidade. Em cada experimento, o usuário pode personalizar o número de instâncias desse *microbenchmark*, o padrão inicial de acesso que cada um delas irá executar e como esses padrões são alterados ao longo do experimento. Cada instância do `access_time3` definida pelo usuário para um determinado experimento é iniciada em um *container* separado pelo programa `storiks`, devendo também ser associada a um arquivo de dados separado para receber a carga de trabalho.

4 AVALIAÇÃO

Este capítulo avalia a pergunta de pesquisa e a hipótese apresentadas neste estudo por meio de uma série de experimentos realizados usando o *framework* Storiks apresentado no Capítulo 3. Na Seção 4.1, são discutidos alguns conceitos gerais usados para conduzir os experimentos. O *hardware* e o ambiente experimental são apresentados na Seção 4.2, enquanto que as cargas de trabalho primárias e concorrentes são apresentadas nas Seções 4.3 e 4.4. Em seguida, a Seção 4.5 apresenta os resultados dos experimentos realizados, ao passo que a Seção 4.6 conclui este capítulo por meio da análise dos resultados apresentados em relação à hipótese.

4.1 CONSIDERAÇÕES PRELIMINARES

De acordo com a Seção 1.1, a hipótese apresentada sugere que a interferência de desempenho sofrida por um motor de armazenamento chave-valor é afetada pela carga de trabalho exercida sobre este motor e pelas cargas de trabalho concorrentes, pelo dispositivo de armazenamento flash e pelo sistema operacional. Uma vez que cada uma das dimensões envolvidas compreende um número proibitivamente grande de alternativas passíveis de serem exploradas tanto individualmente como de forma combinada, este estudo se restringe uma amostra reduzida, porém representativa, de cenários, considerando a hipótese por satisfeita caso seja possível demonstrar este comportamento usando um número limitado de experimentos.

Conforme será detalhado nas Seções 4.2 a 4.5, o cenário proposto para as avaliações de desempenho combina duas cargas de trabalho primárias submetidas ao motor de armazenamento chave-valor, 70 cargas de trabalho concorrentes, 3 dispositivos de armazenamento flash e duas versões do kernel Linux, correspondendo a um conjunto de 840 observações de interferência no total. Usando o *framework* Storiks proposto no Capítulo 3, os experimentos realizados avaliaram cada um dos cenários propostos, comparando a interferência de desempenho em cada uma das dimensões descritas na hipótese (cargas de trabalho primárias e concorrentes, dispositivos de armazenamento e SOs). Conforme a Definição 1, o critério principal de avaliação utilizado (C) é a média de transações por segundo (tx/s) alcançada pelo motor de armazenamento chave-valor. Para simplificar a representação numérica de cada interferência de desempenho, utilizou-se a pressão normalizada ($\overline{\rho}$) descrita na Definição 3.

Conforme descrito nas Seções 3.1 e 3.4.1, o cálculo de $\overline{\rho}$ requer que as medições de desempenho do motor de armazenamento chave-valor sejam feitas com ($\rho(d_i)$) e sem ($\rho(d_0)$) cargas de trabalho concorrentes. Outras condições de sistema, incluindo a carga de trabalho primária, o dispositivo de armazenamento e o SO precisam ser suficientemente estáveis durante essas medições para evitar valores incorretos de $\overline{\rho}(d_i)$. As cargas de trabalho concorrentes produzidas neste estudo foram divididas em dez conjuntos, adicionando-se a cada um deles a carga de trabalho especial d_0. Para cada carga de trabalho primária, dispositivo de armazenamento e sistema operacional, o *framework* Storiks foi utilizado para executar esses conjuntos de cargas de trabalho concorrentes, obtendo-se os respectivos valores $\overline{\rho}(d_i)$ para análises posteriores.

4.2 HARDWARE E AMBIENTE DE TESTES

Esta seção detalha o *hardware* e o ambiente de teste empregados neste trabalho. Para a seleção do *hardware*, foram considerados os seguintes critérios principais: disponibilidade, desempenho e capacidade suficientes para executar os experimentos propostos.

De acordo com a Tabela 4.1, os experimentos foram conduzidos em um computador equipado com uma única CPU AMD Ryzen 3600, de arquitetura x86_64, contendo 6 núcleos e 12 *threads*. Conforme descrito na Tabela 4.2, os 32 GiB de memória RAM utilizados neste equipamento correspondem a 4 pentes de 8 GiB DDR4 operando a 3200 MHz, sendo esta a frequência máxima de memória suportada pela placa mãe sem o uso de *overclock* (Tabela 4.3). Esta configuração de *hardware* implementa um acesso uniforme da CPU aos endereços físicos de memória.

Tabela 4.1: CPU utilizada nos experimentos [12].

Fabricante	AMD
Nome	Ryzen 5 3600
Arquitetura	x86_64
Número de Núcleos	6
Threads/Núcleo	2
Base Clock	3.6 GHz
Boost Clock	4.2 GHz
Cache L1	384 KB
Cache L2	3 MB
Cache L3	32 MB

Tabela 4.2: Memória utilizada nos experimentos [26]*.

Fabricante	Corsair
Nome	VENGEANCE LPX
Numeração	CMK16GX4M2B3200C16
Tipo	DDR4
Tamanho	8 GB
Frequência	PC4-25600 (3200 MHz)

* Foram utilizados 4 módulos de memória deste modelo (4 x 8 GB = 32 GB).

Descrita na Tabela 4.3, a placa mãe utilizada neste estudo possui duas interfaces NVMe: uma PCI-Express (PCIe) Gen4 x4 e uma PCIe Gen3 x2. A primeira é uma interface PCIe dedicada, enquanto que a segunda compartilha sua interface com duas portas SATA. Para evitar possíveis gargalos de desempenho, os experimentos apresentados foram conduzidos utilizando a interface PCIe Gen4 x4. De acordo com as especificações da placa mãe, somente esta interface é compatível, em termos de desempenho, com todos os dispositivos NVMe analisados.

A Tabela 4.4 descreve os três dispositivos de armazenamento flash analisados neste trabalho. Como forma de simplificação, estes dispositivos serão referenciados por NVMeA, NVMeB e NVMeC. Embora todos os três dispositivos sejam provenientes do mesmo fabricante (Samsung) e utilizem a mesma tecnologia de três bits por célula flash (*triple-level cell*, ou TLC), tais modelos diferem entre si em diversos outros aspectos.

Considerando a capacidade de armazenamento, tanto o NVMeA como o NVMeB comportam 250 MB, enquanto que o NVMeC possui o dobro capacidade (500 GB). Uma vez que todos os dispositivos utilizam a tecnologia TLC, esta maior capacidade do NVMeC implica também em duas vezes mais células flash que os demais dispositivos avaliados. Esta capacidade extra de armazenamento precisa ser acomodada de alguma forma usando um número maior de camadas de memória flash (3D NAND), *planes*, *dies* ou *chips* (*packages*), informação

esta geralmente não detalhada pelo fabricante. Dependendo da forma com que cada um dos componentes é organizado, este *hardware* adicional pode aumentar o paralelismo interno e o desempenho final do dispositivo. Esta relação entre capacidade e desempenho pode ser observada comparando as diferentes capacidades dentro de uma mesma linha de produtos [80, 79, 78].

Tabela 4.3: Placa mãe utilizada nos experimentos [14].

Fabricante	ASRock
Nome	B550M Phantom Gaming 4
Número de Série	M80-D9031900219
CPU Sockets	1x AM4
Chipset	AMD B550
Slots de Memória	4x Dual Channel DDR4 3200 MHz
1a Interface NVMe	PCIe Gen4 x4
2a Interface NVMe	PCIe Gen3 x2
Interfaces SATA	4x SATA3

Tabela 4.4: Dispositivos de armazenamento flash avaliados [80, 79, 78].

Referência	**NVMeA**	**NVMeB**	**NVMeC**
Nome/Modelo	Samsung 980 PRO	Samsung 970 EVO Plus	Samsung 970 EVO
Capacidade	250 GB	250 GB	500 GB
Form Factor	M.2	M.2	M.2
Interface	PCIe Gen 4.0 x4, NVMe 1.3c	PCIe Gen 3.0 x4, NVMe 1.3	PCIe Gen 3.0 x4, NVMe 1.3
Controlador	Samsung Elpis	Samsung Phoenix	Samsung Phoenix
Memória Flash	V-NAND 3bit MLC (TLC)	V-NAND 3bit MLC (TLC)	V-NAND 3bit MLC (TLC)
Memória Cache	512MB LPDDR4	512MB LPDDR4	512MB LPDDR4

Todos os três dispositivos de armazenamentos descritos na Tabela 4.4 representam diferentes gerações de um mesmo fabricante. O NVMeA é mais avançado e traz um novo controlador Samsung Epic compatível com o PCIe 4.0, o qual suporta 128 filas de E/S. Os outros dois dispositivos de armazenamento (NVMeB e NVMeC) trazem um controlador mais antigo, Samsung Phoenix, sendo compatíveis com PCIe 3.0 e suportando 32 filas de E/S. Embora utilizem o mesmo controlador, a linha 970 EVO Plus do NVMeB oferece uma pequena vantagem de desempenho comparada com alinha 970 EVO do NVMeC [13]. Uma vez que os dispositivos testados possuem diferentes capacidades, outros fatores, como o paralelismo interno, podem compensar o projeto mais antigo do NVMeC.

Para isolar a carga de E/S dos experimentos de outras cargas de trabalho, o *hardware* utilizado neste estudo também incluiu um HDD SATA de 500 GB. Este dispositivo de armazenamento adicional serviu para conter os arquivos do sistema operacional e os resultados dos experimentos. Uma vez que o SO periodicamente persiste registros de *log* e alguns arquivos temporários, esta demanda adicional de E/S poderia perturbar as medições de desempenho sobre os NVMes avaliados. A mesma precaução foi tomada com relação ao diretório de saída do *framework* Storiks, de modo que os resultados obtidos nos experimentos foram armazenados neste dispositivo adicional.

O sistema operacional utilizado foi o GNU/Linux Ubuntu 20.04 LTS em duas versões de kernel: 5.4 e 5.11. Tais versões do kernel Linux serviram para avaliar a influência de cada uma delas sobre interferência de desempenho de cargas de trabalho concorrentes. O kernel 5.4 é a versão que acompanhou o lançamento do Ubuntu 20.04, sendo uma versão largamente utilizada em 2020 e ainda presente em diversas instalações baseadas nesta distribuição.

No momento em que os experimentos começaram ser realizados, o Ubuntu 20.04 também trazia outras versões de atualização do kernel Linux, sendo a 5.11 a mais recente naquele período. Desta forma, esta versão do kernel foi incluída nos experimentos, ignorando-se versões intermediárias e futuras devido ao limite de gravações suportado pela memória flash dos dispositivos de armazenamento analisados.

De acordo com a Seção 3.4.2, o particionamento e a formatação do sistema de arquivos precisam estar em conformidade com a granularidade e alinhamento de E/S esperados pelo dispositivo de armazenamento com o intuito de maximizar seu desempenho nas operações de leitura e escrita. Conforme descrito na Tabela 4.4, os dispositivos utilizados neste estudo seguem as versões 1.3 e 1.3c da especificação NVMe. Com base nas informações fornecidas pela interface de comando NVMe utilizada pelo SO GNU/Linux [93], os três dispositivos possuem apenas um tipo de *namespace* disponível com granularidade de blocos lógicos (LBAs) igual a 512 bytes. Apesar de previsto nestas versões da especificação, o valor fornecido pelos dispositivos para o parâmetro NOIOB é zero, indicando que não existe um alinhamento de E/S estipulado pelo fabricante. Ao analisar as informações providas por meio do comando `fdisk` e pelo pseudo-arquivo "`/sys/block/*/queue/physical_block_size`", também foi constatado que o kernel Linux reconheceu esses dispositivos como possuindo 512 bytes de granularidade de E/S.

Embora as informações fornecidas pelos dispositivos da Tabela 4.4 não apresentem granularidade ou alinhamento de E/S esperados, não é provável que o tamanho das páginas flash nesses dispositivos seja de 512 bytes [76]. Já ao analisar as especificações desses produtos, pode-se observar que as informações de desempenho divulgadas consideram um tamanho de bloco de 4 KiB para acessos aleatórios [80, 79, 78]. Independente do tamanho de página flash adotado internamente, essas informações de desempenho indicam que tais dispositivos possuem otimizações internas para uma granularidade de E/S de 4 KiB. Como o uso de 4 KiB para o tamanho de bloco é uma prática bastante adotada para este tipo de alinhamento, optou-se por utilizar este tamanho tanto no particionamento como na formatação do sistema de arquivos [5].

Conforme o requisito WR2 descrito na Seção 3.4.1, o estado interno do dispositivo de armazenamento flash deve ser mantido estável para evitar que eventuais mudanças na proporção de páginas flash livres e gravadas influenciem o desempenho do motor de armazenamento chave-valor durante a realização dos experimentos. Quanto maior a quantidade de páginas livres em um dispositivo, melhor é a acomodação das operações de escrita e menor é a pressão exercida sobre o mecanismo de coleta de lixo.

Uma técnica bastante eficaz para o aumento do número de páginas livres é o uso de excesso de provisionamento (*over-provisioning*, ou OP), o qual consiste em diminuir o espaço de endereçamento de blocos lógico (LBAs) em relação à capacidade física do dispositivo [77]. Existem dois tipos principais de excesso de provisionamento:

- De fábrica (*Factory OP*), no qual o próprio dispositivo de armazenamento é fabricado com uma quantidade menor de espaço endereçável em relação à capacidade física; e
- De usuário (*User OP*), no qual o próprio usuário decide reduzir ou não utilizar parte dos LBAs disponíveis.

Assim como para a granularidade e alinhamento de E/S, os dispositivos de armazenamento listados Tabela 4.4 também não apresentam informações sobre a quantidade de excesso de provisionamento de fábrica. Como forma de reduzir a pressão sobre o mecanismo de coleta de lixo e ao mesmo tempo manter uma quantidade estável de páginas flash ocupadas, todos os dispositivos avaliados foram configurados com um percentual fixo de excesso de provisionamento de usuário.

Para configurar este excesso de provisionamento, cada dispositivo foi subdividido em duas partições. A primeira, usada para receber a carga de trabalho dos experimentos, ocupou 70% do dispositivo e foi formatada usando o ext4 com 4 KiB de tamanho de bloco de dados. A segunda partição, usada como excesso de provisionamento, ocupou os 30% restantes[1] e foi completamente limpa usando o comando `blkdiscard`. Este comando informa ao dispositivo que todos os LBAs pertencentes à partição especificada podem ser desassociados de páginas flash gravadas, permitindo que o coletor de lixo apague essas páginas para usos posteriores. Durante os experimentos, os LBAs desta segunda partição não foram mais endereçados por qualquer operação de leitura ou escrita.

Além desta configuração inicial de particionamento e formatação, a partição contendo o sistema de arquivos ext4 também teve que ser pré-condicionada para que o estado das páginas flash associadas a ela não variasse significativamente durante as medições de desempenho. Neste sentido, esta partição foi montada usando o parâmetro `nodiscard` para evitar que o SO solicite ao dispositivo de armazenamento a desassociação de LBAs e a limpeza de páginas flash não utilizadas pelo sistema de arquivos. Além disso, o parâmetro `data=ordered` também foi atribuído durante a montagem do mesmo para forçar que apenas alterações de metadados dos arquivos fossem registradas no mecanismo de *journal* do sistema de arquivos.

Para finalizar o pré-condicionamento, o sistema de arquivos utilizado para receber as cargas de trabalho dos experimentos foi completamente escrito usando uma carga de trabalho completamente aleatória de 4 KiB. O objetivo desta fase foi definir uma organização interna inicial das páginas flash desses dispositivos de modo que fossem similares às cargas de trabalho concorrentes dos experimentos.

Além de estabilizar o estado interno do dispositivo de armazenamento, o estado de energia dos mesmos também foi mantido estável durante as medições de desempenho. Assim como as CPUs atuais, os dispositivos flash avaliados também possuem vários estágios de consumo de energia que podem ser selecionados dinamicamente de acordo com cada carga de trabalho. O mecanismo que faz este tipo de seleção é o *Autonomous Power State Transition* (APST), o qual foi desativado em todos os experimentos realizados, mantendo-se em desempenho máximo (`ps0`) o estado de energia desses dispositivos.

4.3 CARGAS DE TRABALHO PRIMÁRIAS

Dentre as opções disponíveis no *framework* Storiks, foram selecionadas as cargas de trabalho A e B do YCSB para representar as cargas primárias dos experimentos. Uma vez que as especificações do YCSB recomendam que as cargas de trabalho desta ferramenta sejam executadas na ordem A, B, C, F, D e E, optou-se por selecionar as cargas A e B por serem as primeiras desta sequência.

Conforme descrito na Seção 3.4, as cargas A e B do YCSB satisfazem o requisito WR1, uma vez que elas mantém a proporção de cada tipo de transação e não adicionam novos registros

[1] O excesso de provisionamento é calculado pelos fabricantes em relação à capacidade utilizada pelo usuário. Ou seja, (*capacidade física - capacidade do usuário*) / *capacidade do usuário*. Nesta forma de representação, uma partição de 30% do espaço total equivale a aproximadamente 43% de excesso de provisionamento.

durante os experimentos. A carga de trabalho A, composta por 50% de consultas *point lookup* e 50% de atualizações, representa uma carga intensiva em atualizações para este tipo de bancos de dados, exercitando a capacidade de ingestão do motor de armazenamentos chave-valor. Já a carga de trabalho B, composta por 95% de consultas *point lookup* 5% de atualizações, representa uma carga predominantemente de leitura.

Por padrão, cada registro no YCSB possui 20 bytes de tamanho para a chave e 1 KiB para o valor. O número de registros que compuseram o banco de dados utilizado nos experimentos foi de 50 milhões, o que corresponde a aproximadamente 47 GiB de espaço alocado em cada dispositivo de armazenamento. Esta quantidade de registros foi escolhida para melhor acomodar a utilização de espaço nos dispositivos de 250 GB (NVMeA e NVMeB). Considerando ainda outros 40 GiB de espaço alocado para as cargas de trabalho concorrentes produzidas pelo `access_time3` (Seção 4.4), o espaço total ocupado na partição de dados foi de aproximadamente 87 GiB.

O RocksDB provê atualmente dezenas de parâmetros de configuração que permitem ajustar diversas de suas funcionalidades e estruturas internas. Como forma de ajustar este motor de armazenamento para os experimentos realizados, foram consideradas como base de configuração as boas práticas descritas na documentação do projeto. A partir deste ajuste inicial, foram coletados os valores de desempenho das cargas de trabalho A e B. Em seguida, alguns dos parâmetros de configuração considerados relevantes por esta documentação foram novamente avaliados usando diferentes valores. A cada novo ajuste, os valores de desempenho do motor de armazenamento foram coletados para ambas as cargas de trabalho primárias, mantendo os ajustes que proporcionaram o melhor desempenho.

Como forma de permitir a reprodutibilidade dos experimentos, os parâmetros de configuração obtidos durante esta fase de ajuste do motor de armazenamento chave-valor foram codificados em um programa chamado `rocksdb_config_gen`. Disponível dentro da imagem de *container* do *framework* Storiks, este programa gera como saída um arquivo de configurações para RocksDB de acordo com o número de *template* especificado. Os experimentos descritos aqui utilizam o *template* "09", cujas características principais estão listadas na Tabela 4.5.

Tabela 4.5: Principais parâmetros de configuração do RocksDB codificados no *template* 09 programa rocksdb_config_gen.

Parâmetro de configuração do RocksDB	**Valor**
`block_size`	4 KiB
`compression`	kZSTD
`num_levels`	6
`level_compaction_dynamic_level_bytes`	true
`level0_file_num_compaction_trigger`	4
`max_bytes_for_level_base`	512 MiB
`max_bytes_for_level_multiplier`	10
`target_file_size_base`	64 MiB
`write_buffer_size`	128 MiB

O parâmetro `block_size` define como cada arquivo SST do banco de dados é subdividido, representando a unidade mínima de cada requisição de leitura feita por este motor ao sistema de arquivos e ao dispositivo de armazenamento. Usando um tamanho de 4 KiB para este parâmetro, tais operações de leitura se alinham com o tamanho de bloco de dados definido para o sistema de arquivos. De fato, os testes realizados com este parâmetro utilizando os valores 4 e 8 KiB demonstraram que 4 KiB foi o valor que apresentou o melhor desempenho.

O parâmetro `compression` define a forma de compressão dos pares de dados chave-valor dentro dos arquivos SSTs. Dentre as diversas opções de algoritmos disponíveis no RocksDB para esta função, o Zstandard [41] (kZSTD) apresentou o melhor desempenho para as cargas de trabalho A e B do YCSB.

O parâmetro `num_levels` define a quantidade de níveis na árvore LSM. Foram utilizados 6 níveis mas, devido ao parâmetro `level_compaction_dynamic_level_bytes` atribuído como verdadeiro, o volume de dados foi organizado entre o último nível (L_5) e o nível intermediário que melhor ajustou as proporções definidas pelos parâmetros `max_bytes_for_level_base` e `max_bytes_for_level_multiplier`. Dado o tamanho do banco de dados utilizado nesses experimentos, os níveis L_1 e L_2 não foram necessários nesta configuração, de modo que as operações de mesclagem de L_0 foram feitas diretamente em L_3.

A proporção de tamanho entre os níveis adjacentes (T) da árvore LSM, controlada pelo parâmetro `max_bytes_for_level_multiplier`, foi definida em 10, tanto por ser o valor recomendado, como por outros valores maiores e menores não apresentarem melhora de desempenho para as cargas de trabalho avaliadas. Os parâmetros `write_buffer_size` e `level0_file_num_compaction_trigger`, que controlam o tamanho máximo de cada memtable e o número máximo de componentes de dados (*runs*) em L_0, foram respectivamente atribuídos para 128 MiB e 4 *runs* por apresentarem uma melhor relação entre os desempenhos máximo e mínimo do motor de armazenamento para as cargas de trabalho avaliadas. Este desempenho será demonstrado na Seção 4.5.1. Maiores detalhes sobre os parâmetros de ajuste utilizados estão disponíveis no *template* "09" do programa `rocksdb_config_gen` e no repositório do projeto RocksDB [43].

4.4 CARGAS DE TRABALHO CONCORRENTES

Esta seção apresenta os critérios considerados relevantes para compor as cargas de trabalho concorrentes utilizadas neste estudo. Uma vez que essas cargas de trabalho representam apenas uma pequena parcela das possíveis alternativas de configuração que o *framework* Storiks pode produzir, futuros trabalhos podem explorar outras dimensões não contempladas pelo escopo deste estudo.

Considerando os parâmetros `block_size`, `write_ratio`, `random_ratio`, `iodepth`, `o_direct` e `o_dsync` descritos na Tabela 3.1, o número de possíveis cargas de trabalho que uma instância do programa `access_time3` pode produzir é substancialmente grande. Além disso, tais possibilidades crescem exponencialmente à medida que outras instâncias deste programa são adicionadas.

Dada a limitação física de gravações suportada pela memória flash de cada dispositivo de armazenamento avaliado, as cargas de trabalho concorrentes produzidas neste estudo se restringem a uma única instância do `access_time3` com alguns de seus parâmetros fixados em um único valor. Esta instância foi configurada para realizar operações de E/S em um arquivo de 40 GiB armazenado no mesmo sistema de arquivos do banco de dados chave-valor utilizado para executar as cargas de trabalho primárias, sendo o tamanho deste arquivo aproximadamente 85% do tamanho do banco de dados. Além do pré-condicionamento do sistema de arquivos, este arquivo de 40 GiB foi completamente reescrito antes da realização dos experimentos usando uma carga de trabalho aleatória de somente escrita com tamanho de bloco de 4 KiB.

Para compor D, o tamanho de cada requisição de E/S (parâmetro `block_size`) foi fixado em 4 KiB, sendo o mesmo tamanho utilizado para os blocos de dados do sistema de arquivos e para os blocos internos dos arquivos SST do RocksDB. Para evitar a localidade

espacial, o parâmetro `random_ratio` recebeu o valor 1, o que corresponde a cargas de trabalho 100% aleatórias.

Para evitar o mecanismo de *cache* do VFS do sistema operacional, todas as cargas de trabalho concorrentes produzidas pelo `access_time3` tiveram o parâmetro `o_direct` atribuído para verdadeiro (`true`). São dois os motivos considerados para esta configuração. Primeiro, ao utilizar cargas de trabalho 100% aleatórias, o mecanismo de *cache* se torna menos relevante, especialmente para um arquivo de 40 GiB, como é o caso deste estudo. Segundo, espera-se que cada leitura produzida pelo `access_time3` reflita efetivamente em uma requisição de leitura no dispositivo de armazenamento avaliado. A avaliação do mecanismo de *cache* do VFS em cargas de trabalho concorrentes está fora do escopo deste estudo, embora o *framework* Storiks possa ser usado para este fim.

Uma vez considerados como fixos os parâmetros `block_size`, `random_ratio` e `o_direct` do `access_time3`, a composição das cargas de trabalho concorrentes se concentrou nos parâmetros `write_ratio`, `iodepth` e `o_dsync` de acordo com três motivos: Primeiramente, as operações de escrita possuem maior custo para um dispositivo de armazenamento baseado em memória flash do que as operações de leitura. Deste modo, é importante avaliar o efeito de cargas de trabalho concorrentes com diferentes proporções de escrita (valores diferentes para o parâmetro `write_ratio`) sobre o desempenho do motor de armazenamento chave-valor.

Em segundo lugar, uma vez que a sincronização de escrita de dados (parâmetro `o_dsync`) requer um trabalho extra do dispositivo de armazenamento e do sistema operacional para confirmar que os dados foram efetivamente escritos em memória persistente, o efeito deste tipo de requisição em cargas de trabalho concorrentes sobre o desempenho do motor de armazenamento chave-valor também precisa ser avaliado. Em terceiro lugar, a intensidade com que essas operações são submetidas em paralelo ao dispositivo de armazenamento (parâmetro `iodepth`) pode produzir diferentes níveis de contenção sobre o desempenho do motor de armazenamento.

A Tabela 4.6 apresenta os parâmetros do `access_time3` utilizados para compor as cargas de trabalho concorrentes. Foram utilizados cinco valores diferentes para o `write_ratio` (`wr`), variando entre 0 (somente leitura) e 1 (somente escrita). Além de cobrir os valores mínimo, médio e máximo, buscou-se também analisar neste parâmetro os valores 10% acima e abaixo dos respectivos extremos.

Tabela 4.6: Parâmetros do access_time3 utilizados para compor as cargas de trabalho concorrentes.

Parâmetro	**Valor(es)***	**Tipo**
`io_engine`	libaio	
`block_size` (`bs`)	4 (KiB)	
`random_ratio` (`rr`)	1.0	
`o_direct`	true	
`write_ratio` (`wr`)	0.0, 0.1, 0.5, 0.9, 1.0	inter
`o_dsync`	true, false	inter
`iodepth`	1, 2, 4, 8, 16, 32, 64	intra

* Os valores decimais utilizam o ponto (.) como separador.

Para o parâmetro `iodepth`, foram selecionados valores de 1 a 64 em progressão geométrica, multiplicando-se por dois para se obter cada valor subsequente neste intervalo. Utilizou-se este tipo de progressão para melhor cobrir pequenos intervalos próximos ao valor mínimo e um valor máximo relativamente alto. Contudo, valores acima de 64 (ou seja, 128 e 256) não foram avaliados para poupar a vida útil dos dispositivos de armazenamento.

Considerando os parâmetros com mais de um valor na Tabela 4.6, o número de cargas de trabalho concorrentes distintas produzidas pelo `access_time3` corresponde a 70. Uma vez que dispositivos de armazenamento flash podem levar algum tempo para estabilizar o desempenho de uma carga de trabalho devido a acomodações internas de memória e páginas flash [83], além de especificar o conjunto de cargas de trabalho concorrentes, fez-se ainda necessário definir como estas cargas de trabalho foram executadas, de modo a evitar medições de desempenho em condições de instabilidade.

A estratégia utilizada para controlar a estabilidade de desempenho dessas cargas de trabalho concorrentes consiste em agrupá-las em experimentos separados de acordo com suas características, variando apenas a intensidade dessas cargas em cada experimento. Esta organização busca evitar que cargas de trabalho com características muito diferentes sejam postas em sequência, podendo gerar alterações abruptas no estado interno do dispositivo de armazenamento. Deste modo, os parâmetros com múltiplos valores da Tabela 4.6 foram organizados em dois tipos: **inter**-experimentos, os quais variaram apenas de um experimento para outro, e **intra**-experimento, o qual variou progressivamente dentro do mesmo experimento. Os parâmetros inter-experimentos (`write_ratio` e `o_dsync`) formaram 10 grupos contendo 7 cargas de trabalho concorrentes cada (`iodepth`). Cada um desses grupos foi executado para cada dispositivo de armazenamento e carga de trabalho primária analisados neste estudo.

4.5 EXPERIMENTOS

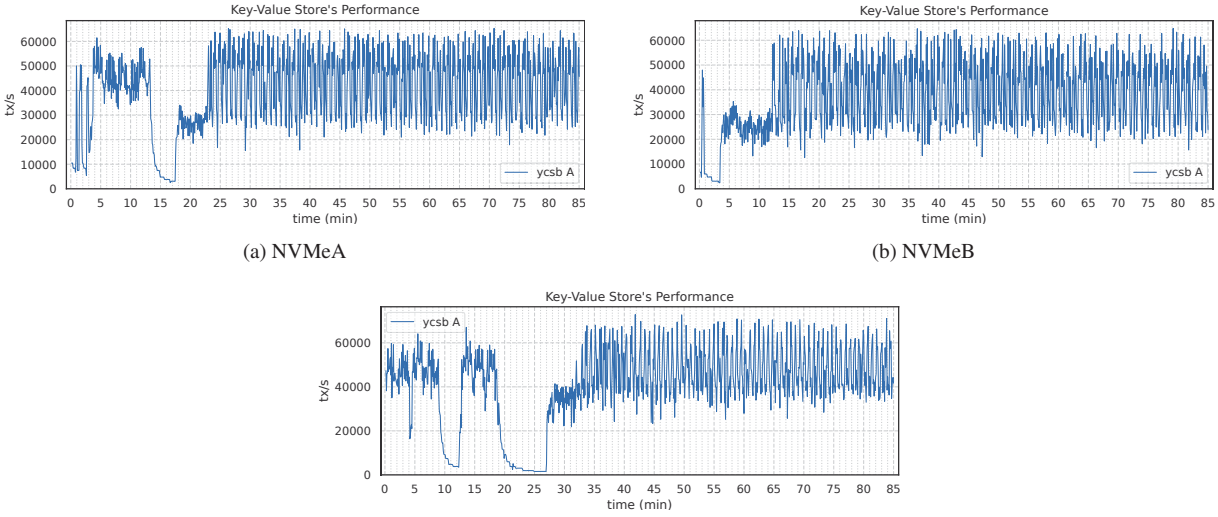
Esta seção apresenta os resultados dos experimentos obtidos por meio do *framework* Storiks. Primeiramente serão avaliados os desempenhos do motor de armazenamento chave-valor e das cargas de trabalho concorrentes executadas separadamente. Em seguida, serão apresentados os resultados utilizando de interferência de desempenho entre esses dois tipos de cargas de trabalho.

4.5.1 Desempenho do Motor de Armazenamento Chave-Valor

Os experimentos a seguir verificam o desempenho e a estabilidade do motor de armazenamento chave-valor sem o uso de cargas de trabalho concorrentes. Conforme descrito na Seção 3.1, esta é uma condição importante para as avaliações apresentadas na Seção 4.5.3. Caso o desempenho deste motor não se mantenha estável durante longos períodos, esta relação entre tempo e desempenho pode comprometer uma comparação correta entre diferentes cargas de trabalho concorrentes. Esta seção apresenta os dados obtidos com a versão 5.11 do Linux, enquanto que a Seção 4.5.3.4 apresenta e compara os dados obtidos com a versão 5.4.

Uma vez que as cargas de trabalho A e B do YCSB são estáveis e em conformidade com WR1, é preciso verificar se os estados internos do motor de armazenamento chave-valor e do dispositivo de armazenamento flash não alteram de padrão ao longo do tempo (WR2). Utilizando o *framework* Storiks, o banco de dados YCSB foi carregado (*bulk load*) conforme as especificações descritas na Seção 4.3. Em seguida, a carga de trabalho A do YCSB foi executada durante 90 minutos para acomodar os níveis internos da árvore LSM e estabilizar sua estrutura.

A Figura 4.1 apresenta o desempenho do RocksDB em transações por segundo (tx/s) nesta fase de preparação para cada um dos dispositivos avaliados. Os 5 primeiros minutos de execução foram removidos de cada gráfico pelo Storiks uma vez que estes foram configurados como período de aquecimento (*warm-up)*. A partir desses gráfios, pode-se observar que RocksDB apresentou estabilidade de desempenho após aproximadamente 30 minutos com o dispositivo NVMeA, 25 minutos com o NVMeB e 45 minutos com o NVMeC.



(a) NVMeA

(b) NVMeB

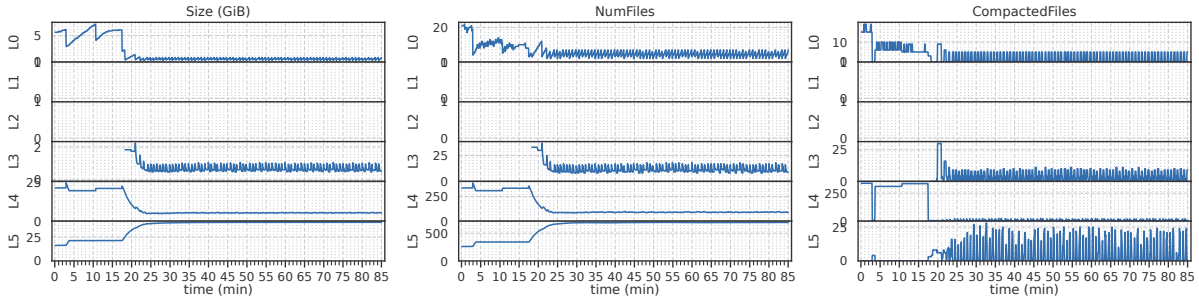
(c) NVMeC

Figura 4.1: Desempenho do motor RocksDB com a carga de trabalho A do YCSB logo após a carga do banco de dados (*bulk load*) e após um aquecimento de 5 minutos.

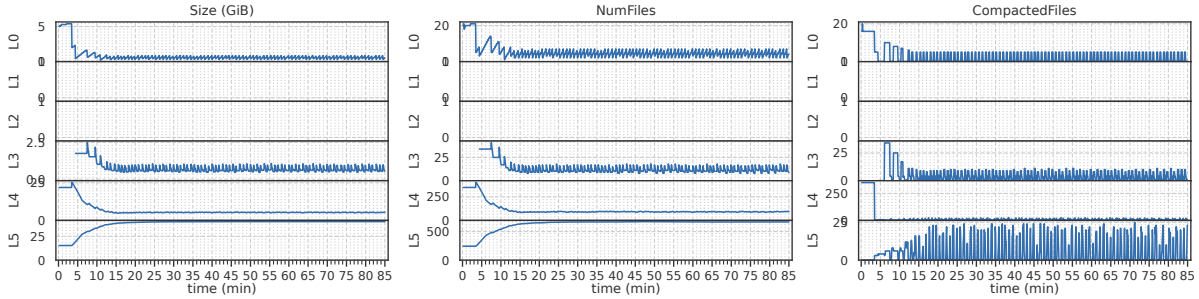
Para confirmar a estabilidade do motor de armazenamento, a Figura 4.2 apresenta, para cada um dos dispositivos, os gráficos de tamanho, número de arquivos SST e número de arquivos compactados em cada nível da árvore LSM extraídos pelo Storiks durante os experimentos representados pela Figura 4.1. Após a carga do banco de dados, o nível menor da árvore LSM (L_0) tende a possuir um volume muito grande de dados, os quais são gradativamente mesclados para os níveis maiores conforme a carga de trabalho A do YCSB é executada. A estabilidade de desempenho do RocksDB com esta carga de trabalho coincide com a estabilidade de tamanho e número de arquivos compactados em cada nível desta árvore, sendo possível prosseguir com os demais experimentos.

Uma vez confirmada a estabilidade do motor de armazenamento chave-valor, as cargas de trabalho A e B do YCSB foram executadas para mais medições de desempenho durante 45 minutos cada. Os primeiros 15 minutos foram utilizados para aquecimento do sistema, incluindo a *cache* do sistema operacional, sendo removidos dessas medições. As Figuras 4.3 e 4.4 demonstram, respectivamente, o desempenho em transações por segundo (tx/s) do RocksDB para essas duas cargas de trabalho no dispositivo NVMeA. Cada ponto de observação desses gráficos corresponde a uma média de 5 segundos.

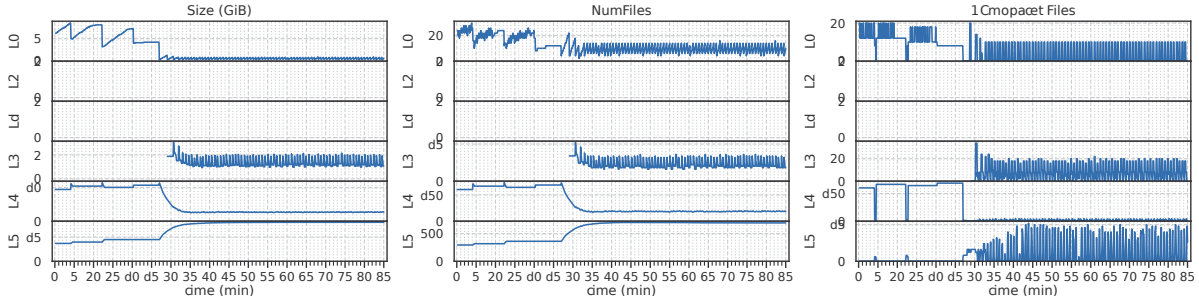
Conforme observado nas Figuras 4.3 e 4.4, o desempenho do RocksDB apresentou oscilações periódicas sem alterar este padrão ao longo do tempo (de 25K a 60K tx/s para o YCSB A e de 75K a 105K tx/s para o YCSB B). As oscilações observadas nesses gráficos são uma característica comum das árvores LSM, causadas pelas operações regulares de mesclagem acionadas quando o primeiro nível (L_0) atinge o número de *runs* especificado pelo parâmetro `level0_file_num_compaction_trigger`.



(a) NVMeA



(b) NVMeB



(c) NVMeC

Figura 4.2: Informações sobre os níveis da árvore LSM do RocksDB durante a execução da carga de trabalho A do YCSB logo após a carga do banco de dados e 5 minutos de aquecimento.

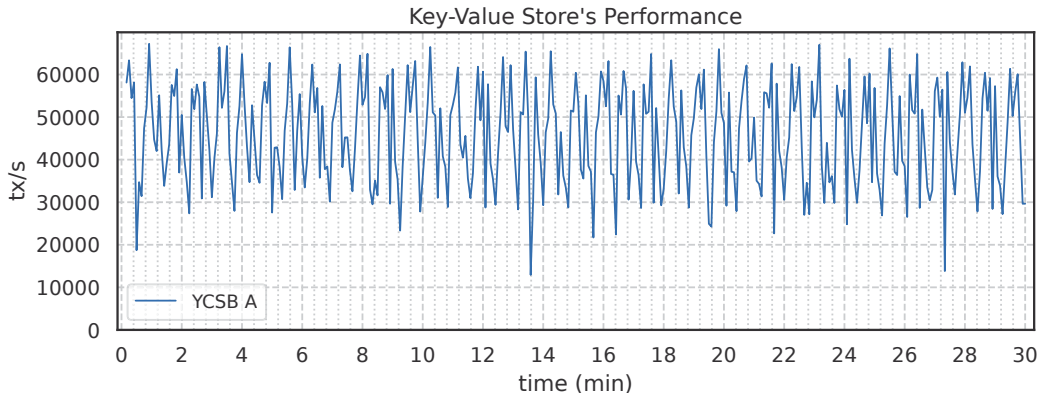


Figura 4.3: Desempenho do RocksDB com a carga de trabalho A do YCSB (NVMeA) após 15 minutos de aquecimento.

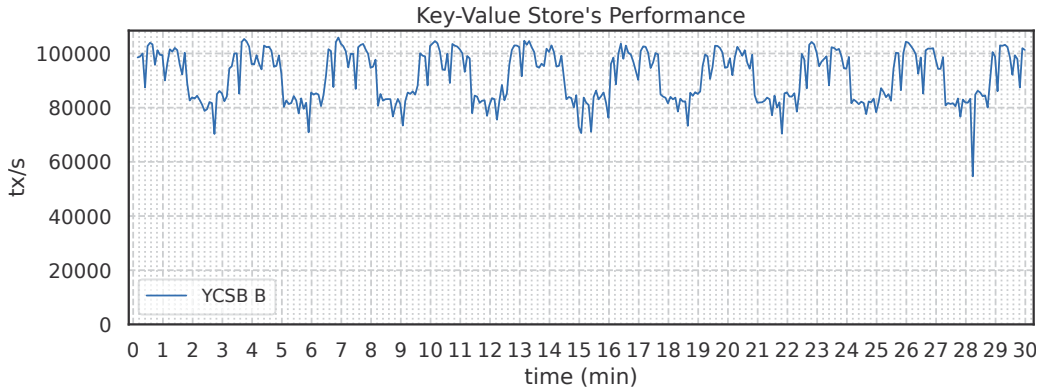
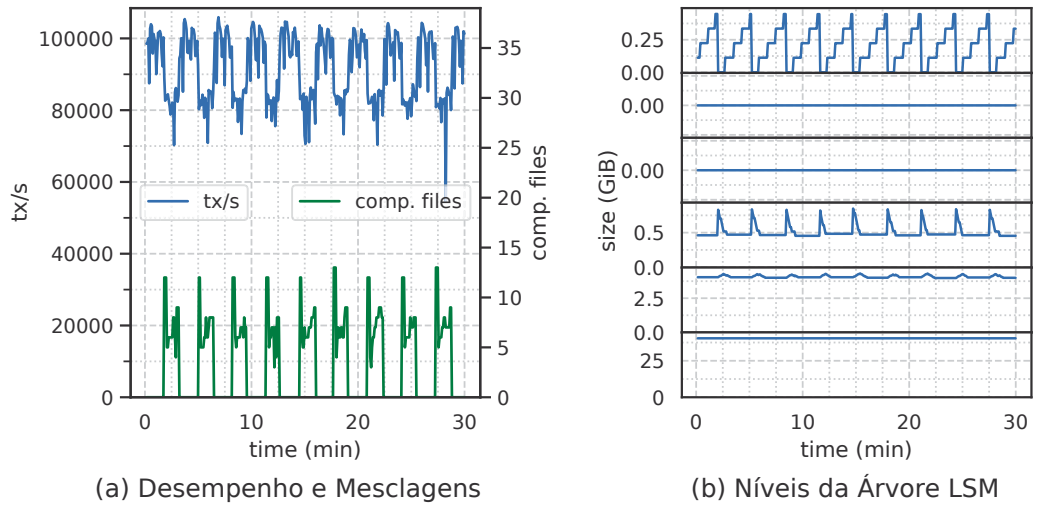


Figura 4.4: Desempenho do RocksDB com a carga de trabalho B do YCSB (NVMeA) após 15 minutos de aquecimento.

A Figura 4.5 ilustra o relacionamento entre o desempenho e as operações de mesclagem (ou compactação) usando as estatísticas coletadas pelo *framework* Storiks no experimento com o YCSB B executando sobre o dispositivo NVMeA. Nesta figura, o gráfico 4.5a apresenta o desempenho do motor de armazenamento chave-valor (tx/s) e o número de arquivos sendo compactados ao longo do tempo (comp. files). Já o gráfico 4.5b mostra o tamanho de cada nível da árvore LSM. A partir desses dois gráficos, é possível observar o aumento e a diminuição periódica de L_0. Quando este nível atinge 512 MiB (quatro *runs* de 128 MiB), o processo de compactação é acionado pelo RocksDB, produzindo um pico no número de arquivos compactados e uma consequente queda de desempenho (tx/s).



(a) Desempenho e Mesclagens (b) Níveis da Árvore LSM

Figura 4.5: Relação entre desempenho e operações de mesclagem ou compactação (YCSB B, NVMeA). A escala de tempo apresentada no eixo x desses gráficos também desconsidera o tempo de aquecimento do experimento.

Como forma de avaliar a estabilidade do dispositivo de armazenamento, a Figura 4.6 apresenta o percentual de blocos lógicos (LBAs) utilizados durante o experimento com a carga de trabalho A do YCSB (Figura 4.3) coletados por meio do `performancemonitor`. Durante todo o experimento, este percentual se manteve estável em aproximadamente 70% da quantidade total de blocos lógicos disponíveis no dispositivo. Conforme descrito na Seção 4.2, tanto o excesso de provisionamento de usuário como o pré-condicionamento da partição utilizada para armazenamento dos dados foram suficientes para manter este nível estabilidade. Este mesmo comportamento foi observado em todos os demais experimentos realizados neste estudo para todos os dispositivos de armazenamento avaliados.

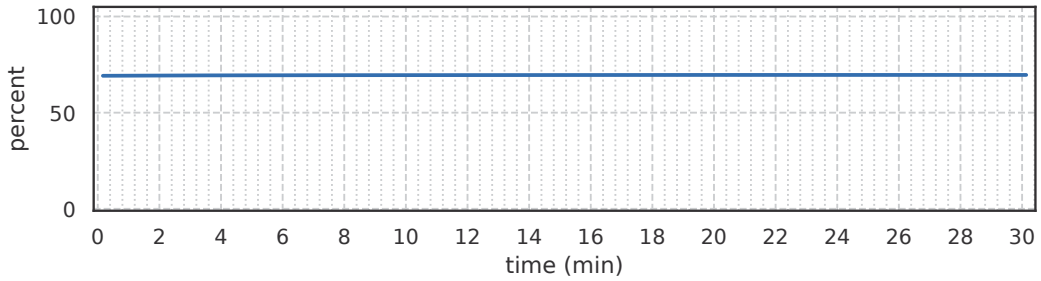
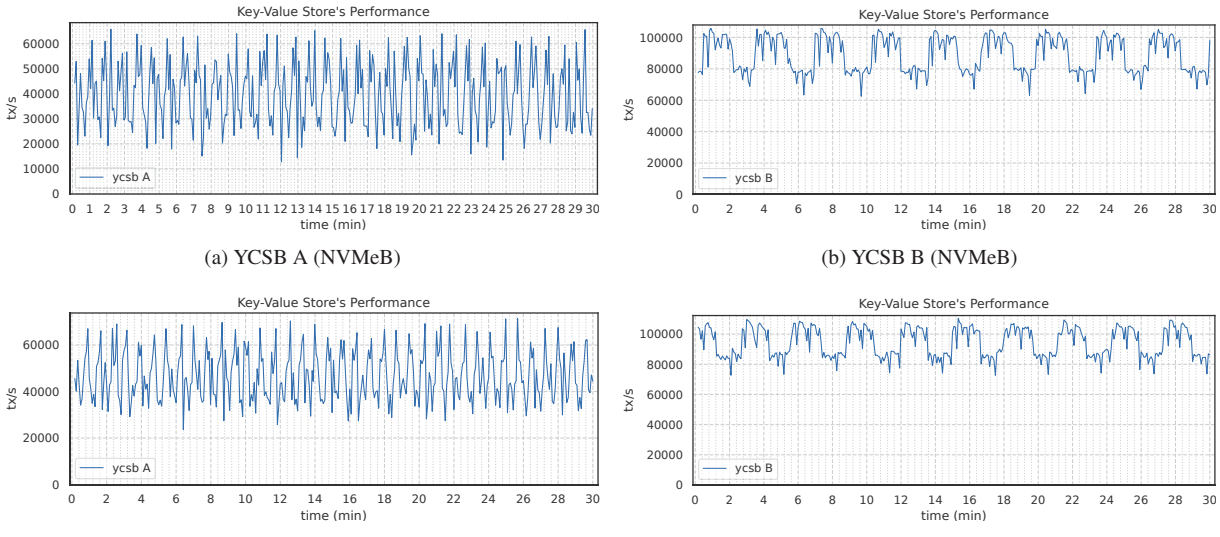


Figura 4.6: Percentual de blocos lógicos utilizados pelo dispositivo de armazenamento durante a carga de trabalho A do YCSB (NVMeA) após 15 minutos de aquecimento.

Os experimentos realizados com os dispositivos NVMeB e NVMeC apresentaram comportamentos similares aos das Figuras 4.3 e 4.4, cujos gráficos estão dispostos em tamanho reduzido na Figura 4.7. Já a Figura 4.8 resume o desempenho das cargas de trabalho A e B do YCSB para todos os dispositivos avaliados usando a função de distribuição cumulativa (CDF). Esta figura mostra que os dispositivos NVMeA e NVMeC apresentaram desempenhos ligeiramente superiores em relação ao NVMeB. Embora da tecnologia empregada no dispositivo NVMeC seja mais antiga que a do NVMeA, sua capacidade dobrada de armazenamento consegue prover um paralelismo interno superior, o que permite ao NVMeC ultrapassar o desempenho do NVMeA por uma pequena margem.



(a) YCSB A (NVMeB)

(b) YCSB B (NVMeB)

(c) YCSB A (NVMeC)

(d) YCSB B (NVMeC)

Figura 4.7: Desempenho do RocksDB com as cargas de trabalho A e B do YCSB (NVMeB e NVMeC) após 15 minutos de aquecimento.

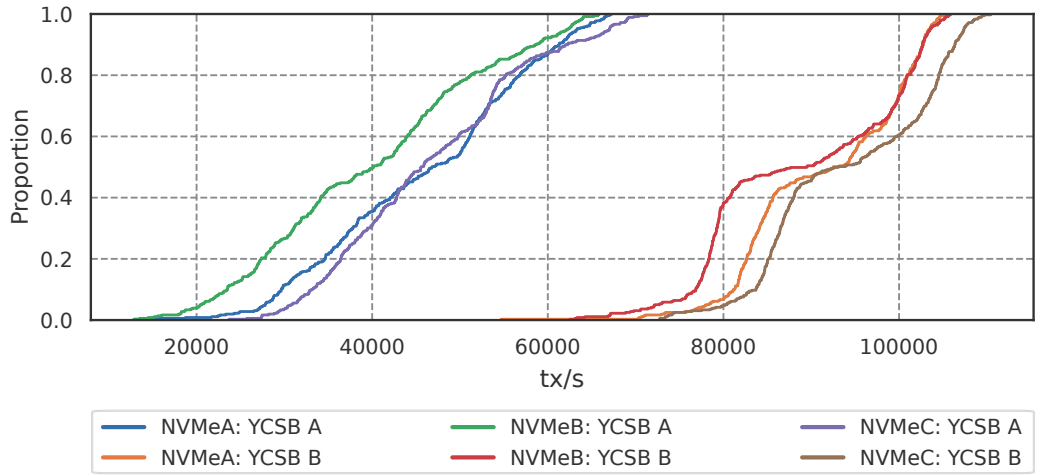


Figura 4.8: Distribuição cumulativa de desempenho de todas as cargas de trabalho YCSB e dispositivos avaliados.

4.5.2 Desempenho das Cargas de Trabalho Produzidas pelo Access_time3

Esta seção avalia o desempenho das cargas de trabalho produzidas pelo `access_time3`, conforme descrição feita na Seção 4.4. Cada um dos 10 grupos produzidos pela variação dos parâmetros `write_ratio` e `o_dsync` representou um experimento que foi executado sobre um arquivo de 40 GiB armazenado e pré-condicionado no mesmo sistema de arquivos onde o banco de dados chave-valor foi armazenado. Este procedimento foi realizado para cada dispositivo de armazenamento avaliado. Esta seção apresenta os dados obtidos com a versão 5.11 do kernel Linux, enquanto que a Seção 4.5.3.4 apresenta os dados obtidos com a versão 5.4.

Conforme a Tabela 4.6, o único parâmetro que variou em cada experimento foi o `iodepth`, representando um progressivo aumento, em sete níveis, no grau de concorrência das cargas de trabalho (de 1 a 64). Cada experimento iniciou com `iodepth` 1 por 5 minutos de aquecimento, seguido por 10 minutos para medição de desempenho. Em seguida, os demais valores de `iodepth` foram sequencialmente atribuídos, obedecendo um intervalo de 10 minutos entre cada um deles. Cada mudança de parâmetro do `access_time3` foi registrada pelo `storiksd` no arquivo de saída do experimento para determinar o início e o fim de cada carga de trabalho.

As Figuras 4.9 e 4.10 mostram as medições de desempenho de cada um dos dispositivos de armazenamento com o parâmetro `o_dsync` ativado e desativado, respectivamente. Cada ponto de observação nessas figuras representa a vazão média em MiB/s dessas cargas de trabalho executadas em intervalos de 10 minutos. Em ambas as figuras, o desempenho obtido pelas cargas de trabalho somente leitura (`wr` = 0) foi aproximadamente o mesmo devido à ausência de requisições de escrita. Neste caso, observa-se uma progressiva melhora de vazão do dispositivo à medida que se aumentou o valor de `iodepth`. O NVMeA apresentou o melhor desempenho somente leitura entre os dispositivos avaliados, seguido pelo NVMeC. Apesar do NVMeB ser um modelo com tecnologia ligeiramente superior ao NVMeC, a capacidade dobrada de armazenamento deste último consegue fornecer um paralelismo interno maior e, consequentemente, um melhor desempenho. Em contrapartida, a mesma capacidade dos dispositivos NVMeA e NVMeB evidencia a tecnologia superior deste primeiro em relação ao último.

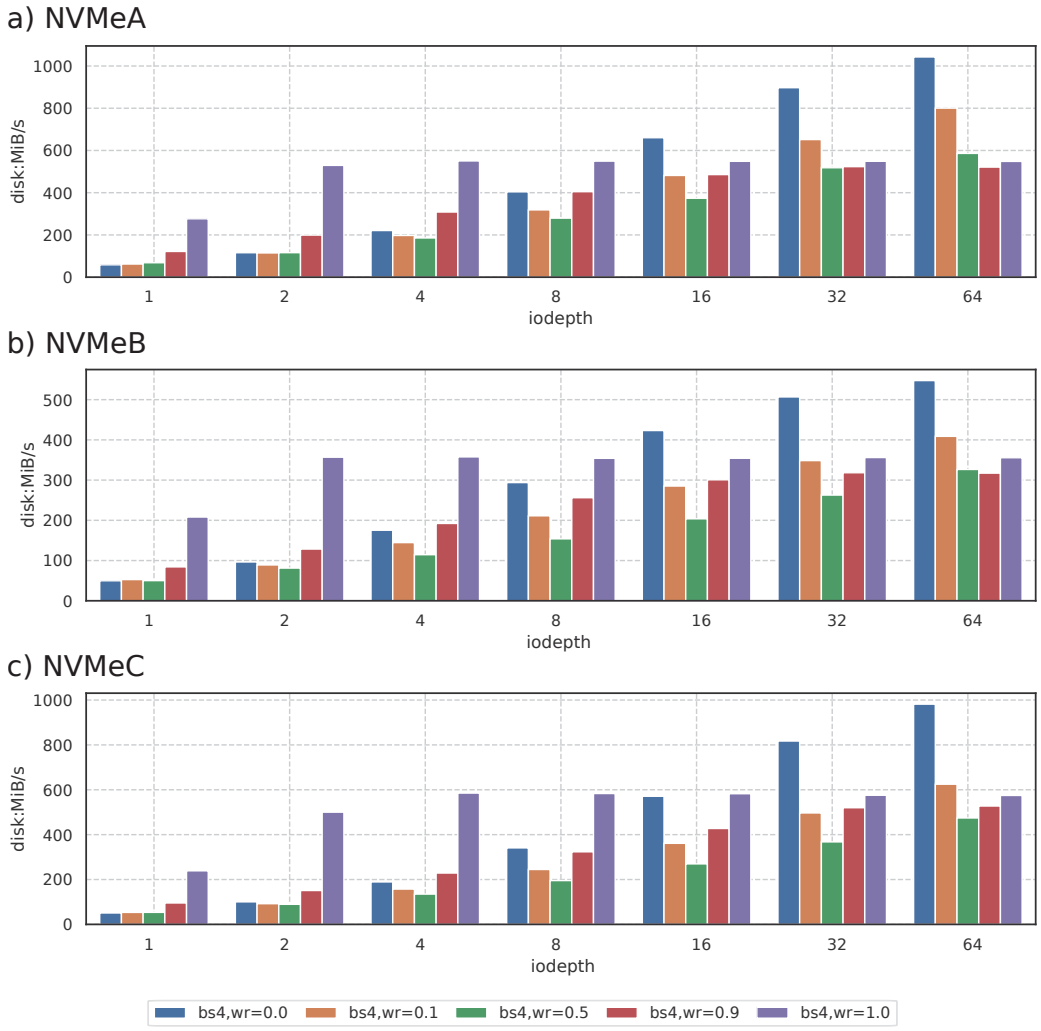


Figura 4.9: Desempenho dos dispositivos de armazenamento com 4 KiB de tamanho de bloco, acessos uniformemente aleatórios e sincronização de dados desativada.

Os demais oito experimentos analisam o desempenho dos dispositivos de armazenamento com diferentes proporções de operações de escrita na carga de trabalho (`wr` = 0.1 a 1). Nesses casos, as Figuras 4.9 e 4.10 mostram uma clara distinção entre os resultados com a sincronização de dados ativada e desativada. De acordo com a Figura 4.9, o desempenho de cada dispositivo de armazenamento é substancialmente superior com esta opção desativada. Esta figura mostra ainda que o padrão de desempenho variou em função do `iodepth`. Para valores baixos de `write_ratio` (`wr` = 0.1 a 0.5), o desempenho alcançado por cada dispositivo de armazenamento seguiu um desempenho similar ao observado em `wr` = 0, embora em uma escala menor. Interessantemente, o desempenho alcançado pelas cargas de trabalho com `wr` = 0.5 foi inferior aos outros valores de `wr` na maioria dos experimentos da Figura 4.9, revelando uma maior dificuldade deste tipo de dispositivo de armazenamento em suportar cargas de trabalho mistas de leitura e escrita.

No lado oposto desse espectro, a Figura 4.9 também mostra que o desempenho dos dispositivos de armazenamento foi superior com `wr` = 0.9 e 1 do que para outros valores de `write_ratio` nos casos onde `iodepth` se manteve baixo. Com `wr` = 1, observa-se que o desempenho melhora de `iodepth` = 1 para `iodepth` = 2, permanecendo estável a partir daí, tendo seu desempenho ultrapassado pelas cargas de trabalho somente leitura no `iodepth` = 16. Devido à progressão geométrica do parâmetro `iodepth`, tais experimentos não são precisos o

suficiente para identificar o ponto de intersecção entre diferentes valores `write_ratio`. Tal análise não é considerada como parte do escopo deste estudo.

Conforme demonstrado pela Figura 4.10, a sincronização de dados representa uma degradação significativa de desempenho para as cargas de trabalho, mesmo em pequenas proporções deste tipo de requisição. Uma vez que essas requisições precisam ter sua durabilidade confirmada pelo sistema operacional, elas geralmente possuem uma latência alta, retardando requisições subsequentes de E/S da carga de trabalho. Exceto `wr` = 0, esta figura demonstra uma margem pequena de ganho de desempenho com o aumento de `iodepth` e uma pequena degradação de desempenho com o aumento de `write_ratio`.

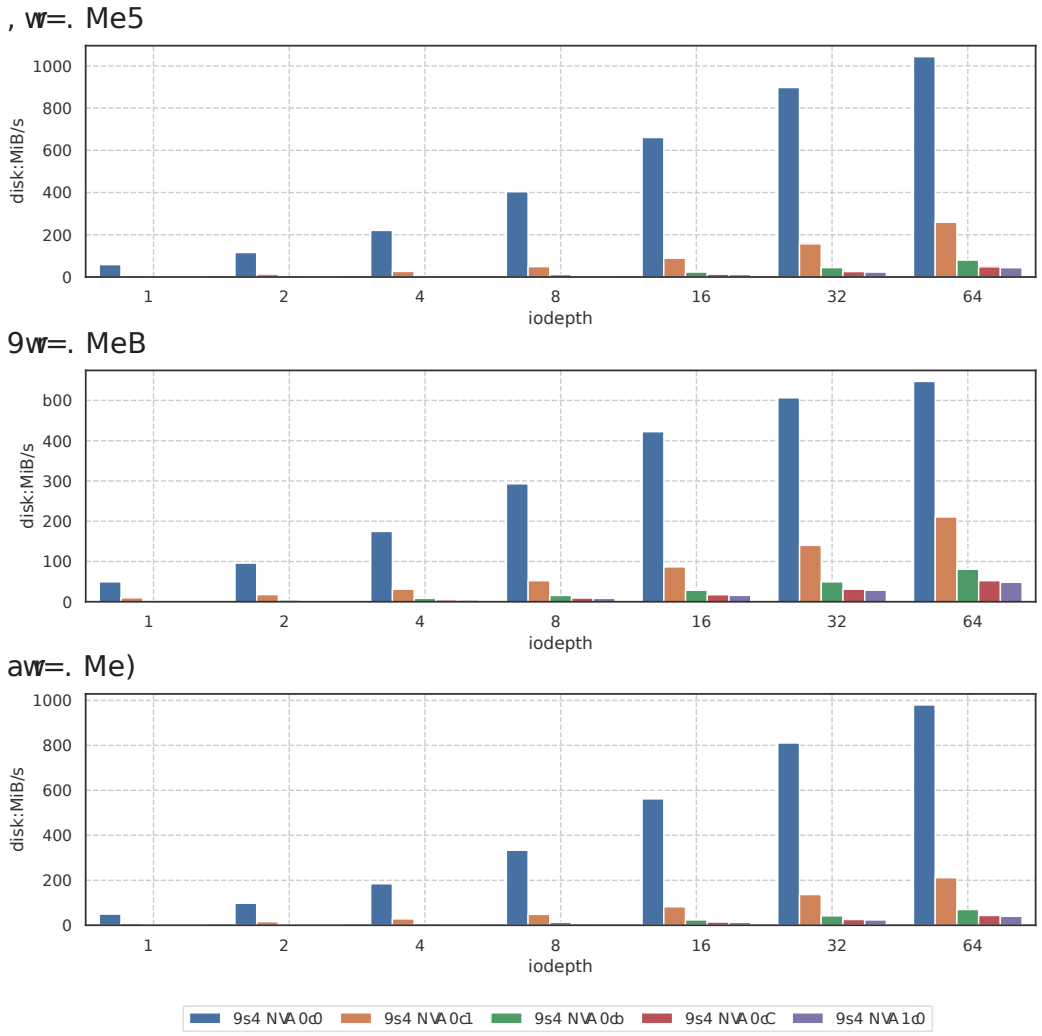


Figura 4.10: Desempenho dos dispositivos de armazenamento com 4 KiB de tamanho de bloco, acessos uniformemente aleatórios e sincronização de dados ativada.

4.5.3 Interferência de Desempenho Entre as Cargas de Trabalho

Esta seção analisa a interferência de desempenho sofrida pelo motor de armazenamento chave-valor quando exposto a diferentes tipos de cargas de trabalho concorrentes. Para produzir esta análise, utilizou-se o *framework* Storiks para combinar as cargas de trabalho previamente descritas em um conjunto de 20 experimentos para cada dispositivo de armazenamento e versão do kernel Linux avaliados. Uma vez que grande parte dos detalhes sobre essas cargas de trabalho já foram apresentadas nas Seções 4.5.1 e 4.5.2, esta seção apenas revisa algumas de suas principais

características, apresentando os aspectos considerados relevantes para a combinação dessas cargas.

A partir da Definição 1, o critério de desempenho principal considerado neste estudo é o valor médio de transações por segundo (tx/s) alcançado pelo motor de armazenamento chave-valor em cada carga de trabalho do YCSB avaliada na Seção 4.5.1. Em vez de simplesmente utilizar a letra C para representar este critério, esta seção irá referenciá-lo como C^{kv} para que um critério secundário de desempenho também possa ser analisado mais adiante. O conjunto de cargas de trabalho concorrentes, representado por D, corresponde às cargas produzidas pelo `access_time3`, conforme Seção 4.5.2.

Os experimentos de interferência de desempenho apresentados aqui correspondem a um produto cartesiano entre as duas cargas de trabalho do YCSB e os dez grupos de cargas de trabalho produzidas pelo `access_time3`. Em cada um desses experimentos, a carga de trabalho do YCSB foi configurada para executar ininterruptamente. Durante os 25 primeiros minutos, esta carga de trabalho foi executada sozinha, sendo os 15 primeiros minutos utilizados como aquecimento (*warm-up*), seguidos de 10 minutos de medições de desempenho. Para atender a Definição 3, este intervalo de 10 minutos corresponde a d_0, um caso especial de D representando a ausência de carga de trabalho concorrente.

Após o período representado por d_0, cada experimento executou sequencialmente as respectivas cargas de trabalho concorrentes produzidas pelo `access_time3`, do `iodepth` 1 ao 64, utilizando 10 minutos de intervalo para cada uma delas. Para facilitar a identificação, essas cargas de trabalho são referenciadas aqui como $d_1, d_2, \ldots, d_{64}$, conforme seus respectivos valores do parâmetro `iodepth`. O desempenho médio em MiB/s alcançado por cada cargas de trabalho produzida pelo `access_time3` compõe o segundo critério de desempenho, referenciado por C^{at3}. Embora este critério não esteja obrigatoriamente relacionado à hipótese deste estudo, ele serve para realizar algumas análises complementares sobre a relação de desempenho entre as cargas de trabalho primárias e concorrentes.

A partir da Definição 1, a função de pressão $\rho^{kv}(d_i)$ refere-se ao desempenho obtido pelo motor de armazenamento chave-valor para cada $d_i \in D$. Similarmente, o desempenho alcançado por cada carga de trabalho produzida pelo `access_time3` será referenciada como $\rho^{at3}(d_i)$. Considerando a Definição 2, uma vez que valores menores de tx/s, obtidos pelo motor de armazenamento, e valores menores de MiB/s, obtidos pelo `access_time3`, representam desempenhos piores, os conjuntos ordenados $(C^{kv}, \preceq_C^{kv})$ e $(C^{at3}, \preceq_C^{at3})$ podem ser trivialmente definidos como $(C^{kv}, \leq)$ e $(C^{at3}, \leq)$.

Para simplificar a representação de dados, a Definição 3 estabelece a normalização da pressão exercida por uma carga de trabalho d_i em relação a d_0. Ou seja:

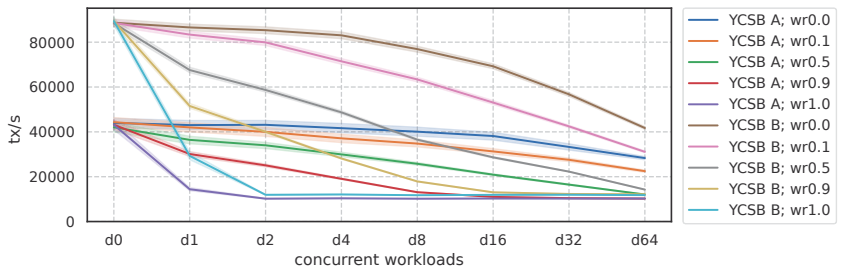
$$\overline{\rho}^{kv}(d_i) = (\rho^{kv}(d_0) - \rho^{kv}(d_i))/\rho^{kv}(d_0) \tag{4.1}$$

Esta pressão normalizada foi calculada por meio da biblioteca Storiks utilizando os valores de $\rho^{kv}(d_0)$ e $\rho^{kv}(d_i)$ de cada experimento individualmente.

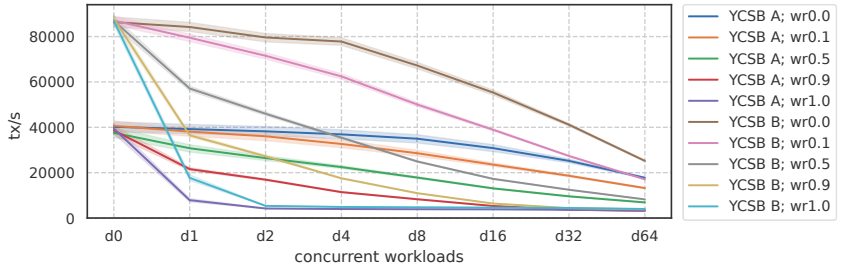
Para a normalização de $\rho^{at3}(d_i)$, uma vez que os experimentos apresentados nesta seção não possuem algo equivalente a $\rho^{at3}(d_0)$, utilizou-se como base de referência os experimentos apresentados na Seção 4.5.2. Uma vez que cada carga de trabalho do `access_time3` avaliada aqui (d_i) possui um correspondente (d_i^*) naquela seção, a pressão normalizada $\overline{\rho}^{at3}(d_i)$ foi calculada da seguinte forma:

$$\overline{\rho}^{at3}(d_i) = (\rho^{at3}(d_i^*) - \rho^{at3}(d_i))/\rho^{at3}(d_i^*) \tag{4.2}$$

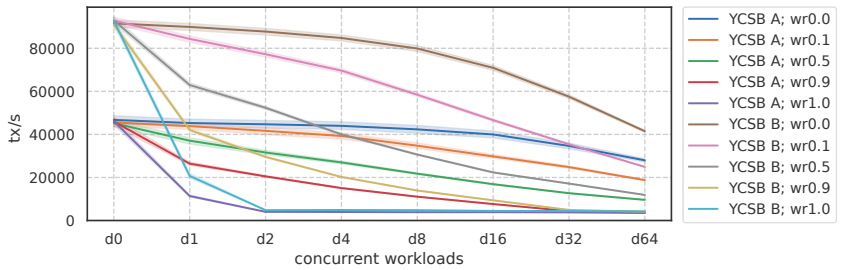
As Figuras 4.11, 4.12, 4.13 e 4.14 mostram a interferência de desempenho obtida pelos experimentos descritos no início desta subseção para cada dispositivo de armazenamento utilizando a versão 5.11 do kernel Linux. Os gráficos das Figuras 4.11 e 4.12 referem-se aos experimentos com a sincronização de dados desativada nas cargas de trabalho geradas pelo `access_time3`, enquanto que as Figuras 4.13 e 4.14 apresentam os resultados com esta opção ativada.



(a) NVMeA



(b) NVMeB



(c) NVMeC

Figura 4.11: Desempenho médio do RocksDB em relação a cada uma das cargas de trabalho concorrentes (o_dsync = false, kernel Linux versão 5.11)

As Figuras 4.11 e 4.13 apresentam os resultados dos experimentos na forma $\rho^{kv}(d_i)$, ou seja, desempenho médio do motor de armazenamento chave-valor em tx/s, sendo que cada linha desses gráficos corresponde a um experimento. Já as Figuras 4.12 e 4.14 trazem os valores obtidos em sua forma normalizada. Cada grupo de barras representa um experimento, sendo que os gráficos do lado esquerdo correspondem à pressão normalizada sofrida pelo motor de armazenamento chave-valor ($\overline{\rho}^{kv}$), enquanto que os gráficos do lado direito representam a pressão normalizada sofrida pelas cargas de trabalho concorrentes geradas pelo `access_time3` ($\overline{\rho}^{at3}$).

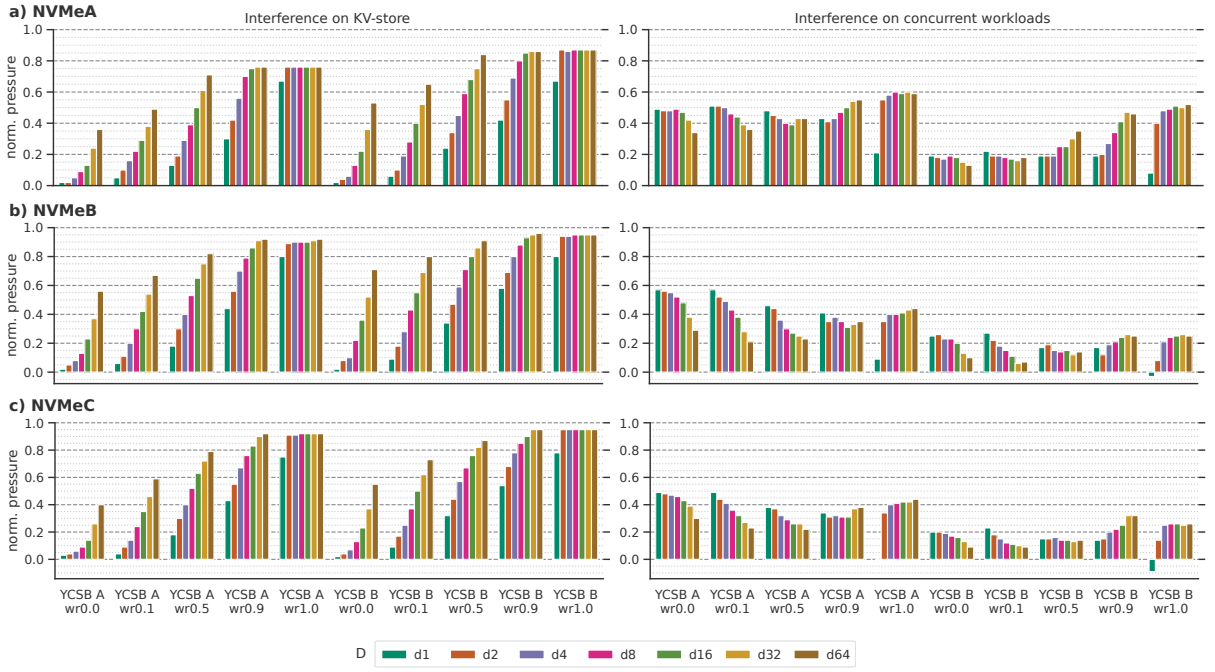


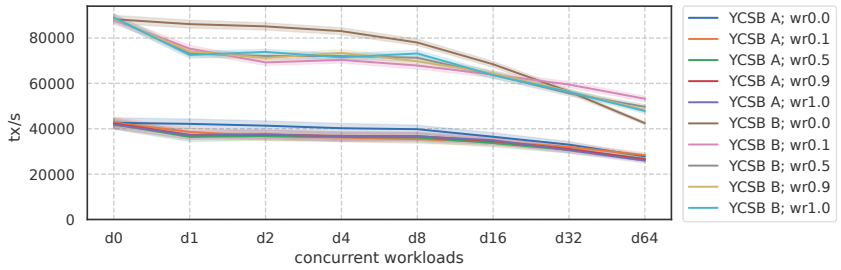
Figura 4.12: Interferência de desempenho sofrida pelas cargas de trabalho primárias e concorrentes (o_dsync = false, kernel Linux versão 5.11).

4.5.3.1 Sincronização de Dados

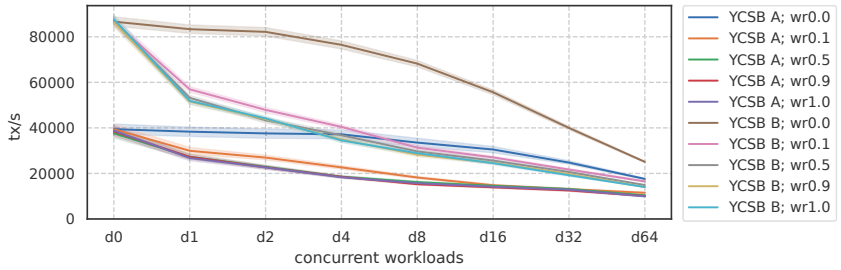
Os gráficos das Figuras 4.12 e 4.14 demonstram uma clara diferença entre as pressões normalizadas $\overline{\rho}^{kv}$ dos experimentos com e sem sincronização de dados nas cargas de trabalho concorrentes. Conforme apresentado na Figura 4.12, a ausência de sincronização de dados produziu uma degradação progressiva de desempenho em relação tanto ao aumento da proporção de escritas (`wr`) como de requisições simultâneas (`iodepth`) nas cargas de trabalho geradas pelo `access_time3`. Este mesmo efeito foi observado em todos os dispositivos de armazenamento e cargas de trabalho do YCSB.

Os experimentos da Figura 4.12 também demonstram que as cargas de trabalho concorrentes de somente leitura (`wr` = 0) produziram as menores pressões sobre o motor de armazenamento chave-valor em relação aos demais valores de `write_ratio` dentro do mesmo dispositivo de armazenamento, `iodepth` e carga de trabalho do YCSB. Neste caso, o dispositivo de armazenamento com menores valores de $\overline{\rho}^{kv}$ foi o NVMeA, com $\overline{\rho}^{kv}(d_{64}) = 0,36$, seguido pelo NVMeC $(0,4)$ e finalmente pelo NVMeB $(0,55)$. Esses gráficos também mostram uma diferença pequena entre os valores de $\overline{\rho}^{kv}$ produzidos por essas cargas de trabalho concorrentes cujo `iodepth` se manteve baixo. Na maioria dos casos, a degradação de desempenho exercida sobre o motor de armazenamento foi menor que 10% em d_4.

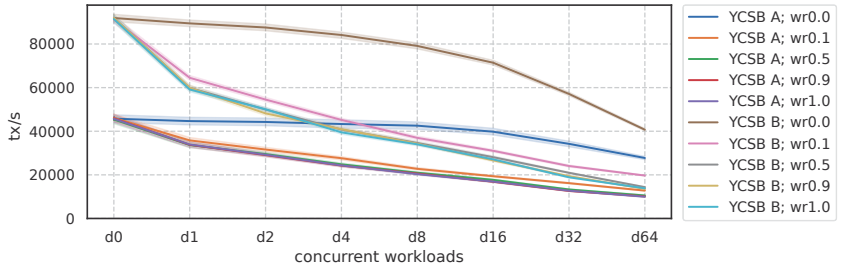
Em cada carga de trabalho primária (YCSB A e B) e dispositivo de armazenamento, os experimentos da Figura 4.12 também demonstram um progressivo aumento nos respectivos valores de $\overline{\rho}^{kv}$ em função de `wr`. Embora $\overline{\rho}^{kv}(d_1)$ tenha aumentado significativamente em função do aumento de `write_ratio`, a diferença entre $\overline{\rho}^{kv}(d_2)$ e $\overline{\rho}^{kv}(d_{64})$ diminuiu até colapsar em aproximadamente os mesmos valores em `wr` = 1. Nesses experimentos, os dispositivos de armazenamento mais suscetíveis a cargas de trabalho de concorrentes de escrita foram o NVMeC e NVMeB. Contudo, devido aos valores elevados de $\overline{\rho}^{kv}$ mesmo em d_1, o uso de cargas de trabalho concorrentes de somente escrita sem sincronização de dados deve ser evitado em qualquer um dos dispositivos avaliados.



(a) NVMeA



(b) NVMeB



(c) NVMeC

Figura 4.13: Desempenho médio do RocksDB em relação a cada uma das cargas de trabalho concorrentes (o_dsync = true, kernel Linux versão 5.11)

Conforme demonstrado na Figura 4.14, a sincronização de dados em cargas de trabalho concorrentes produziu um cenário diferente de pressão sobre o motor de armazenamento chave-valor. Exceto para `wr` = 0, o qual corresponde aproximadamente aos mesmos valores da Figura 4.12 devido à ausência de operações de escrita concorrente, os demais experimentos produziram pequenas variações em seus respectivos resultados dentro do mesmo valor de `write_ratio`, dispositivo de armazenamento e carga de trabalho do YCSB. Na maioria dos casos, os valores de $\overline{\rho}^{kv}$ com `wr` = 0.1 foram ligeiramente melhores, enquanto que os experimentos com valores maiores de `wr` apresentaram variações desprezíveis entre si.

O NVMeA foi claramente o dispositivo de menor suscetividade às cargas de trabalho concorrentes nos experimentos da Figura 4.14. Com o YCSB A e `write_ratio` entre 0.1 e 1, a interferência de desempenho sofrida pelo motor de armazenamento foi similar a `wr` = 0 para os valores de `iodepth` entre 16 e 64. Com o YCSB B, o valor obtido em $\overline{\rho}^{kv}(d_{64})$ foi maior em `wr` = 0 do que para os demais valores de `wr`. Já para cargas de trabalho concorrentes com valores baixos de `iodepth`, os experimentos com `wr` = 0 produziram valores melhores de $\overline{\rho}^{kv}$ em relação às demais opções de `write_ratio` analisadas.

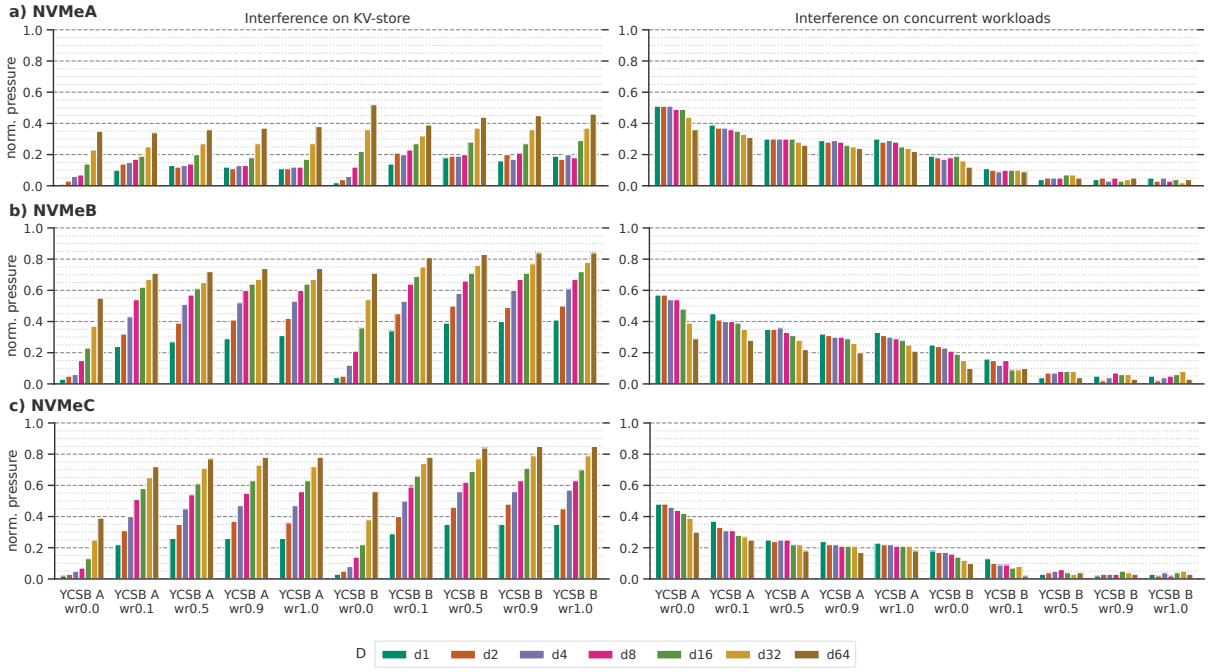


Figura 4.14: Interferência de desempenho sofrida pelas cargas de trabalho primárias e concorrentes (o_dsync = true, kernel Linux versão 5.11).

Para os dispositivos NVMeB e NVMeC, os experimentos da Figura 4.14 demonstram que mesmo uma pequena fração de requisições concorrentes de escrita com sincronização de dados ativa pode afetar negativamente o desempenho do motor de armazenamento chave-valor em pelo menos 20% para o YCSB A e aproximadamente 30% para o YCSB B. Devido à baixa suscetividade ao aumento do parâmetro `write_ratio`, esses dispositivos apresentam valores mais elevados de $\overline{\rho}^{kv}$ nos experimentos com `wr` = 0.1 se comparados aos da Figura 4.12. À medida que aumentou-se o `write_ratio`, as cargas de trabalho concorrentes sem sincronização de dados produziram maiores pressões sobre o motor de armazenamento chave-valor (Fig. 4.12) que suas cargas de trabalho correspondentes com sincronização de dados ativa (Fig. 4.14).

4.5.3.2 Diferenças entre YCSB A e B

Comparando os valores de $\overline{\rho}^{kv}$ entre os experimentos com o YCSB A e YCSB B, é possível constatar a partir das Figuras 4.12 e 4.14 que o YCSB B sofreu uma interferência maior de desempenho em relação ao YCSB A na maioria dos cenários avaliados. Apesar do desempenho superior do YCSB B, conforme demonstrado na Figura 4.8, esta carga de trabalho predominantemente de leitura tende a ser mais suscetível a cargas de trabalho concorrentes do que o YCSB A, sendo esta última de escrita intensiva.

A Figura 4.15 apresenta a diferença entre os valores de $\overline{\rho}^{kv}$ obtidos pelos experimentos com o YCSB B e os respectivos valores de $\overline{\rho}^{kv}$ com o YCSB A para cada dispositivo de armazenamento, `write_ratio`, `o_dsync` e `iodepth`. Para cargas de trabalho concorrentes sem sincronização de dados, as maiores diferenças são observadas em `wr` = 0.5 e `iodepth` entre 2 e 16. Nesses casos, os valores de $\overline{\rho}^{kv}$ com o YCSB B foram entre 13% e 20% superiores que em relação ao YCSB A. Já para os experimentos com `wr` = 0 e 0.1, as maiores diferenças de pressão normalizada são observadas entre d_{16} e d_{64}, com valores entre 11% e 16% superiores para o YCSB B. O NVMeA foi o dispositivo de armazenamento com maiores diferenças de pressão normalizada entre as duas cargas de trabalho primárias, seguido pelo NVMeB e NVMeC.

Para cargas de trabalho concorrentes com sincronização de dados (exceto `wr` = 0), a Figura 4.15 mostra uma diferença menor entre os valores de $\overline{\rho}^{kv}$ para o YCSB B e o YCSB A. Nesses casos, as maiores diferenças obtidas foram entre 10% e 13%, embora não seja possível observar um padrão definido entre os dispositivos de armazenamento. O NVMeB foi o dispositivo maior número de valores acima de 10%, seguido pelo NVMeC e NVMeA.

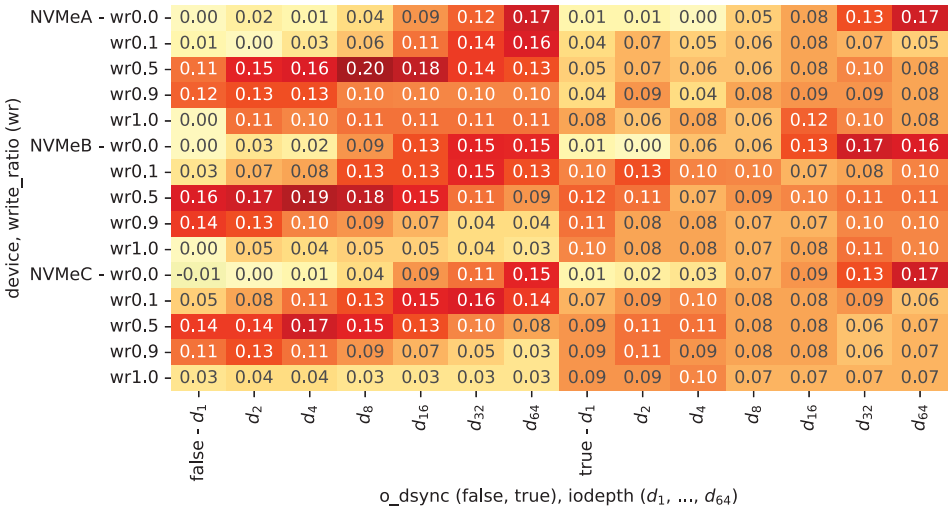


Figura 4.15: Diferença entre os valores de $\overline{\rho}^{kv}$ obtidos pelos experimentos com o YCSB B e o YCSB A para cada dispositivo de armazenamento (*device*), `write_ratio`, `o_dsync` e `iodepth` (kernel Linux versão 5.11).

4.5.3.3 Pressão sobre as Cargas de Trabalho do Access_time3 ($\overline{\rho}^{at3}$)

Além da pressão sofrida pelo motor de armazenamento chave-valor, também é possível analisar a pressão que este motor exerceu sobre as cargas de trabalho geradas pelo `access_time3`. Conforme descrito na Equação 4.2, a função $\overline{\rho}^{at3}$ representa esta pressão normalizada a partir dos valores de referência obtidos pelos experimentos das Figuras 4.9 e 4.10. Esta pressão normalizada é apresentada nos gráficos do lado direito das Figuras 4.12 e 4.14. É possível constatar a partir desses experimentos que a degradação de desempenho sofrida pelas cargas de trabalho do `access_time3` não seguiu o mesmo padrão de seus respectivos valores obtidos pela função $\overline{\rho}^{kv}$ (lado esquerdo dos gráficos). Na maioria dos experimentos, observa-se que a ordem produzida pelos valores da função $\overline{\rho}^{at3}$ é inversa e em intervalos menores que na função $\overline{\rho}^{kv}$. Nesses casos, as cargas de trabalho do `access_time3` que apresentaram degradações de desempenho menores produziram maiores impactos negativos sobre o desempenho do motor de armazenamento chave-valor.

Algumas exceções para esta inversão de valores obtidos pelas funções $\overline{\rho}^{at3}$ e $\overline{\rho}^{kv}$ são mais evidentes nos experimentos da Figura 4.12 com valores altos de `write_ratio`, especialmente com `wr` = 1 para todas as cargas de trabalho do YCSB e com `wr` = 0.9 para o YCSB B. Nestes casos, a degradação de desempenho sofrida pelas cargas de trabalho do `access_time3` foi muito menor que seus respectivos efeitos sobre o desempenho do motor de armazenamento chave-valor. Além destas, outras exceções são observadas nos experimentos onde a ordem produzida pela função $\overline{\rho}^{at3}$ não é bem definida e em intervalos pequenos. A maioria

dos experimentos que se enquadram nesta situação são os da Figura 4.14 que usam o YCSB B e valores de `wr` maiores que 0.1. Na Figura 4.12, esta ausência de ordem definida pode ser observada nos dispositivos NVMeB e NVMeC combinados com YCSB A e `wr` = 0.9 e com o YCSB B e `wr` = 0.5. Apesar de apresentarem uma distância pequena entre os valores de $\overline{\rho}^{at3}$ nesses casos, os valores correspondentes de $\overline{\rho}^{kv}$ são bem ordenados e em intervalos maiores.

Outra observação importante a ser feita a partir da Figura 4.12 é a presença de dois valores negativos de $\overline{\rho}^{at3}(d_1)$ para os dispositivos NVMeB $(-0,03)$ e NVMeC $(-0,09)$ usando o YCSB B e cargas de trabalho concorrentes de somente escrita (`wr` = 1). Uma suposição inicial para este fenômeno foi que as cargas de trabalho utilizadas para o aquecimento do sistema e que antecedem d_1 e d_1^* não eram as mesmas e estariam influenciando $\rho^{at3}(d_1)$ e $\rho^{at3}(d_1^*)$. Esta possibilidade foi descartada por meio de experimentos adicionais.

A explicação para os valores negativos de $\overline{\rho}^{at3}(d_1)$ está relacionada com as operações de leitura produzidas pela carga de trabalho B do YCSB. Para confirmar esta hipótese, foram executados os mesmos experimentos da Figura 4.9 com `wr` = 1, mas adicionando uma segunda instância do `access_time3` configurada para executar leituras aleatórias (`wr` = 0 e `rr` = 1) de tamanho de bloco igual a 4 KiB e `iodepth` = 1 sobre um arquivo de 20 GiB. Para os dispositivos NVMeB e NVMeC, esta carga de trabalho adicional melhorou o desempenho da primeira instância do `access_time3` em 42% e 11% para `iodepth` = 1, respectivamente. Para os demais valores de `iodepth`, a degradação de desempenho sofrida por esta instância foi positiva e ficou abaixo de 10%. Para o NVMeA, observou-se uma degradação de desempenho de 1% em `iodepth` = 1 e entre 31% e 36% para os demais valores de `iodepth`.

4.5.3.4 Diferentes Versões do Kernel

Para avaliar o efeito de diferentes versões do kernel Linux neste cenário de interferência de desempenho, foram executados os mesmos experimentos das Figuras 4.3 a 4.14 também com a versão 5.4 do kernel. Para os experimentos com cargas de trabalho produzidas pelo `access_time3` sem sincronização de dados e com o YCSB A e B executados separadamente, os resultados com esta versão de kernel são relativamente próximos às Figuras 4.9 e 4.8 na maior parte dos casos, conforme demonstrado nas Figuras 4.16 e 4.17. Na comparação entre as Figuras 4.9 e 4.16, o desempenho das cargas de trabalho produzidas pelo `access_time3` sem sincronização de dados foi superior com o kernel 5.4 para os dispositivos NVMeB e NVMeC e valores de `write_ratio` iguais a 0.5 e 0.9. Nesses casos, o aumento de desempenho em relação ao kernel 5.11 ficou entre 0 e 10%.

Já para o NVMeA, o desempenho do `access_time3` sem sincronização de dados com o kernel 5.4 foi menor em relação à versão 5.11 do Linux, especialmente para valores de `wr` iguais a 0.9 e 1 e `iodepth` maiores que 1. Nesses casos, a o desempenho com o kernel 5.4 ficou entre 7% e 28% menor em relação ao kernel 5.11. Já para `wr` igual a 0.5 e `iodepth` igual a 32 e 64, o desempenho com o kernel 5.4 foi 16% e 17% menor, respectivamente. Nos demais casos, a perda de desempenho com o kernel 5.4 foi, em sua maioria, inferior a 6%.

Na comparação entre as Figuras 4.8 e 4.17, o desempenho das cargas de trabalho A e B do YCSB executadas separadamente foi menor com kernel 5.4 em relação ao kernel 5.11. Nesses casos, a carga de trabalho B do YCSB foi a que apresentou maior diferença de desempenho, ficando entre 8,2% e 10%. Já para o YCSB A, a diferença de desempenho entre as versões do kernel foi de aproximadamente 2,6% para o NVMeC, 6,3% para o NVMeB e 8,37% para o NVMeA.

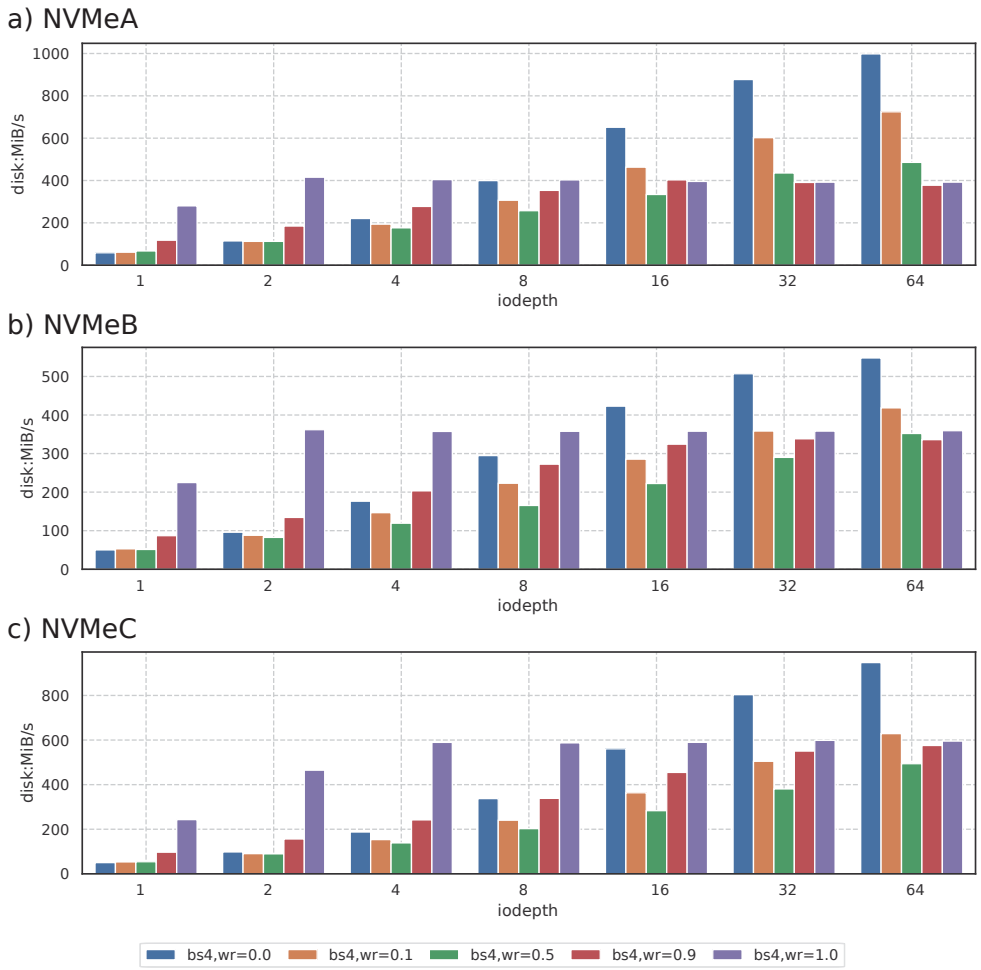


Figura 4.16: Desempenho dos dispositivos de armazenamento com 4 KiB de tamanho de bloco, acessos uniformemente aleatórios e sincronização de dados desativada (kernel Linux 5.4).

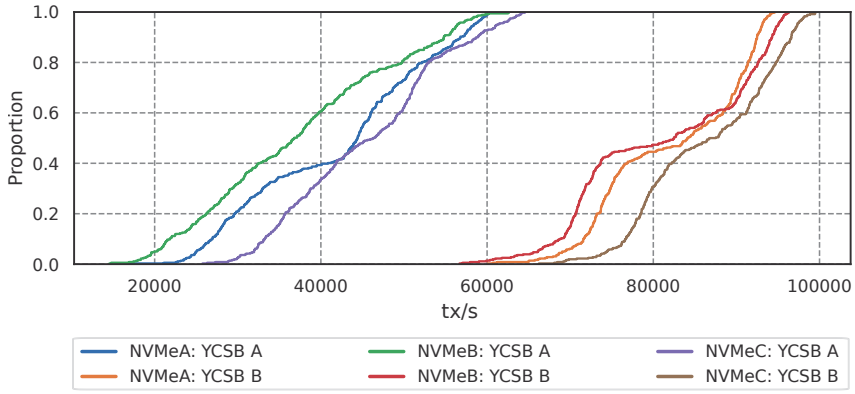


Figura 4.17: Distribuição cumulativa de desempenho de todas as cargas de trabalho YCSB e dispositivos avaliados (kernel Linux 5.4).

De modo semelhante à Figura 4.12, a Figura 4.18 apresenta os experimentos realizados com o kernel 5.4 para cargas de trabalho concorrentes sem sincronização de dados. Já a Figura 4.19 apresenta a diferença entre os valores de pressão normalizada representados por essas duas figuras, de modo que valores negativos correspondem a uma pressão maior para os experimentos com o kernel 5.11. Ao comparar os valores de $\overline{\rho}^{kv}$, observa-se a partir dessas figuras que o motor de armazenamento sofreu uma pressão ligeiramente menor com o kernel 5.4 (entre 0% e 5%) na maioria dos casos avaliados. Diferenças maiores que estas (entre 6% e 8%) em favor do kernel 5.4 foram encontradas especialmente nos casos envolvendo os dispositivos NVMeA e NVMeC, o YCSB B e cargas de trabalho concorrentes com `write_ratio` = 0 e `iodepth` entre 32 e 64. No NVMeC, também observa-se uma diferença de 6% em dois casos específicos: (YCSB A, `write_ratio` = 0, `iodepth` = 64) e (YCSB B, `write_ratio` = 0.1, `iodepth` = 16).

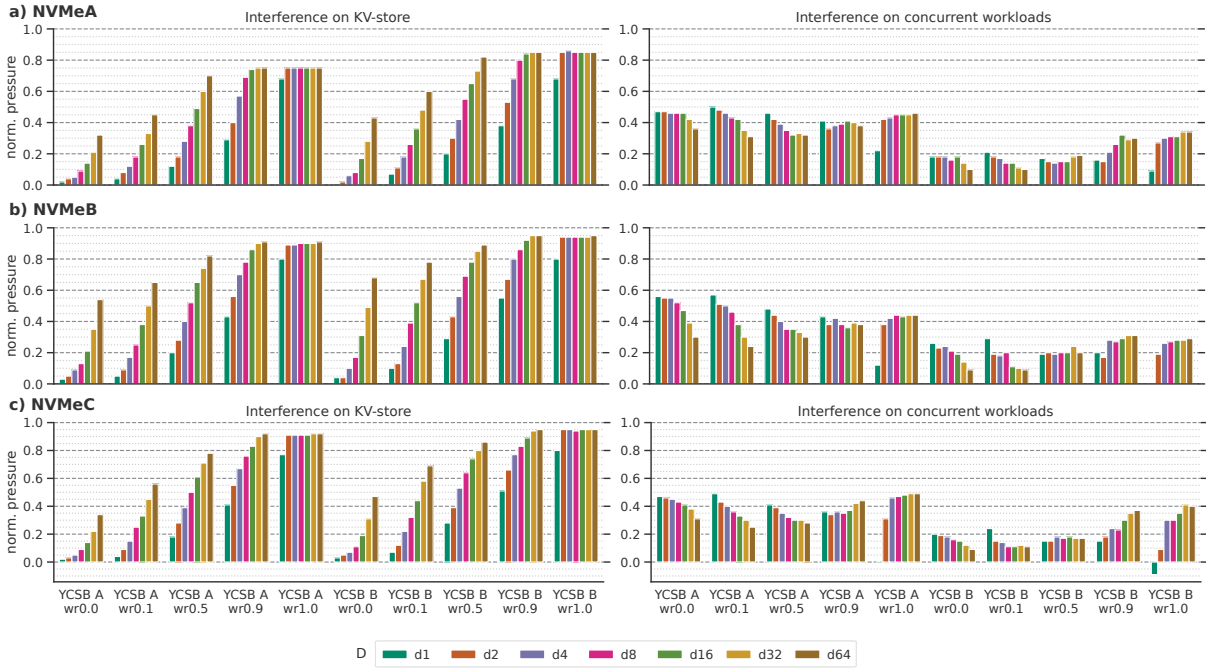


Figura 4.18: Interferência de desempenho sofrida pelas cargas de trabalho primárias e concorrentes (o_dsync = false, kernel Linux versão 5.4).

Na comparação entre os valores de $\overline{\rho}^{at3}$, os resultados apresentados nas Figuras 4.12 e 4.18 são semelhantes na maioria dos casos, com um aumento mais evidente de pressão exercida sobre o `access_time3` nos experimentos envolvendo o kernel 5.11, o dispositivo NVMeA e valores de `wr` maiores ou iguais a 0.9. Já no caso do dispositivo NVMeC, os valores de $\overline{\rho}^{at3}$ foram maiores com o kernel 5.4, especialmente com o YCSB A e `wr` menor e igual a 0.5.

As maiores divergências encontradas entre as versões 5.11 e 5.4 do kernel Linux ocorreram nos experimentos em que as cargas de trabalho produzidas pelo `access_time3` utilizaram sincronização de dados, em especial com os dispositivos NVMeB e NVMeC. Comparando com a Figura 4.10, a Figura 4.20 demonstra que o desempenho alcançado com a kernel 5.4 foi entre 4 e 44% inferior em relação kernel 5.11 para `iodepth` = 64 usando o dispositivo NVMeA. Para `iodepth` = 32, a diferença relativa de desempenho neste dispositivo ficou entre -1% e 42%.

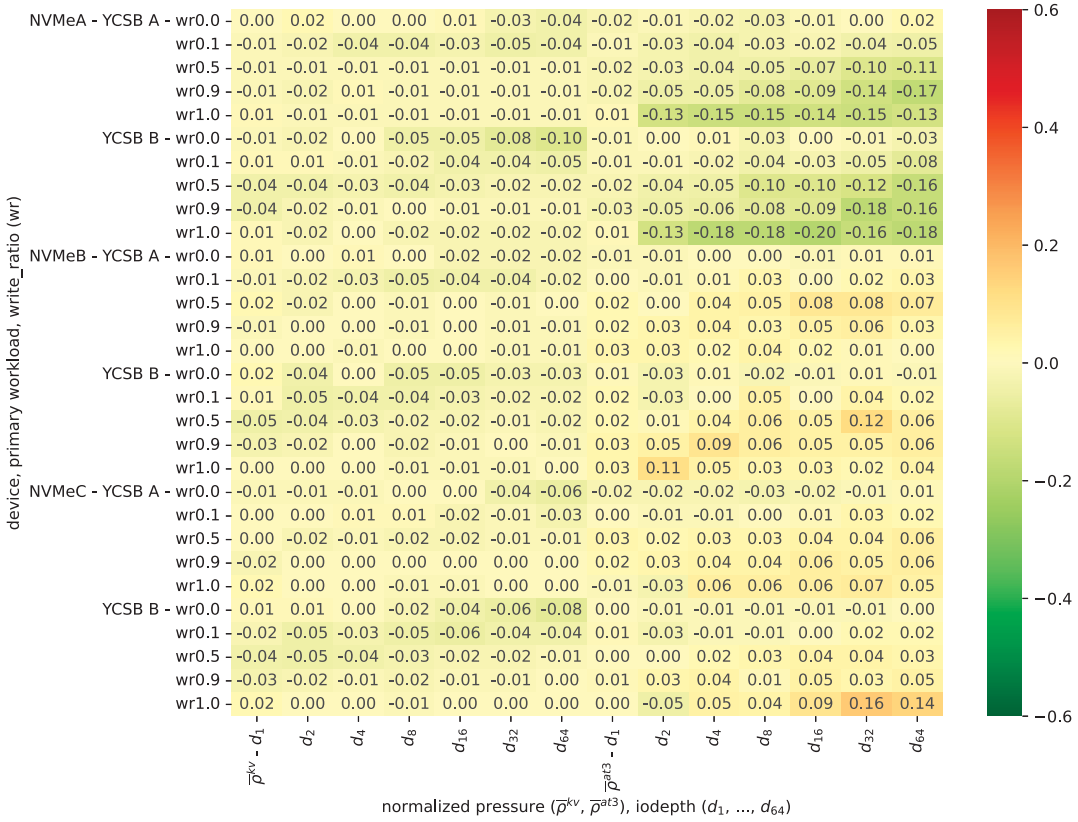


Figura 4.19: Diferença entre os valores de pressão normalizada entre as versões 5.4 e 5.11 do kernel Linux (o_dsync = false).

Já para os dispositivos NVMeB e NVMeC, o desempenho alcançado pelas cargas de trabalho produzidas pelo `access_time3` (Figura 4.20) foi significativamente menor com o kernel 5.4 em relação ao kernel 5.11, principalmente nos casos envolvendo valores de `iodepth` maiores que 2 e `write_ratio` maior que 0. Utilizando o NVMeB, os resultados foram entre 34% e 73% piores em relação ao kernel 5.11, enquanto que o NVMeC apresentou valores entre 39% e 84% inferiores. Em contrapartida, todos os dispositivos avaliados obtiveram ganhos de desempenho com o kernel 5.4 nos casos envolvendo `iodepth` = 1 e `wr` maior que 1. O NVMeA obteve um desempenho entre 83% e 111% maior em relação ao kernel 5.11, enquanto que o NVMeB ficou entre 29% e 44% e o NVMeC entre 22% e 34%.

A Figura 4.21 apresenta o resultado dos mesmos experimentos de interferência de desempenho da Figura 4.14 utilizando a versão 5.4 do kernel Linux. A diferença entre os valores de pressão normalizada obtidos pelos experimentos dessas duas figuras é apresentada na Figura 4.22 (valores de $\overline{\rho}^{kv}$ e $\overline{\rho}^{at3}$ com o kernel 5.4 menos os respectivos valores com o kernel 5.11). Comparando os valores de $\overline{\rho}^{kv}$ entre as Figuras 4.14 e 4.21, pode-se observar que as cargas de trabalho concorrentes de escrita com sincronização de dados ativa representam um fator de risco significativamente maior para o desempenho do motor de armazenamento chave-valor utilizando os dispositivos NVMeB e NVMeC e a versão 5.4 do kernel. Nesses casos, a degradação de desempenho observada em d_1 e `wr` = 0.1 foi de pelo menos 59% para o NVMeB e 67% para o NVMeC, chegando a valores superiores a 90% de degradação de desempenho em d_{64} para ambos os dispositivos. Considerando a diferença entre os valores de $\overline{\rho}^{kv}$ para essas duas versões de kernel Linux (Figura 4.22), os resultados obtidos em d_1 com a versão 5.4 foram entre 35% e 49% maiores utilizando o dispositivo NVMeB e entre 45% e 62% utilizando o NVMeC.

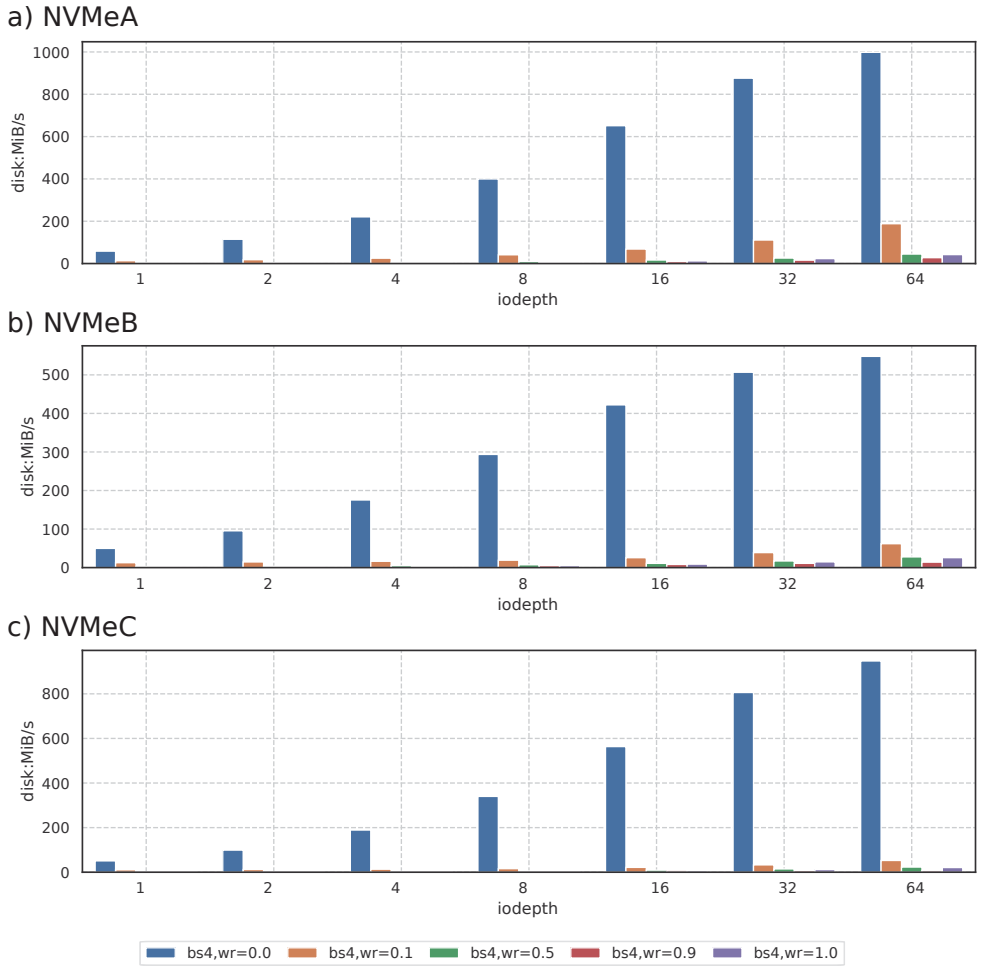


Figura 4.20: Desempenho dos dispositivos de armazenamento com 4 KiB de tamanho de bloco, acessos uniformemente aleatórios e sincronização de dados ativada (kernel Linux 5.4).

No caso do dispositivo NVMeA, a Figura 4.22 demonstra uma diferença menor de degradação de desempenho sobre o motor de armazenamento chave-valor. Para valores de `iodepth` entre 1 e 16 (d_1 a d_{16}), os valores obtidos de $\overline{\rho}^{kv}$ com o kernel 5.4 foram entre 5% e 11% maiores. Em d_{64}, esta diferença ficou entre -5% e 1% para os experimentos com o YCSB A e entre -4% e 7% para o YCSB B, demonstrando casos onde a degradação de desempenho com o kernel 5.4 foi menor em relação ao kernel 5.11.

A diferença significativa nos valores de interferência de desempenho entre as duas versões de kernel para conjuntos específicos de cargas de trabalho e dispositivos de armazenamento demonstram um relacionamento complexo entre os elementos avaliados. Uma vez que tanto o tipo do dispositivo de armazenamento persistente como a versão do sistema operacional tendem a não ser homogêneos em uma infraestrutura de computação em nuvem de larga escala, variações significativas de desempenho podem ocorrer quando motores de armazenamento chave-valor são migrados de um servidor de virtualização para outro mesmo que as cargas de trabalho concorrentes entre os servidores sejam semelhantes. Além disso, tais problemas de degradação de desempenho podem ocorrer inesperadamente quando alguma dessas aplicações ou serviços compartilhando o mesmo *hardware* começa a submeter algum tipo específico de requisição de E/S, como requisições de gravação com sincronização de dados.

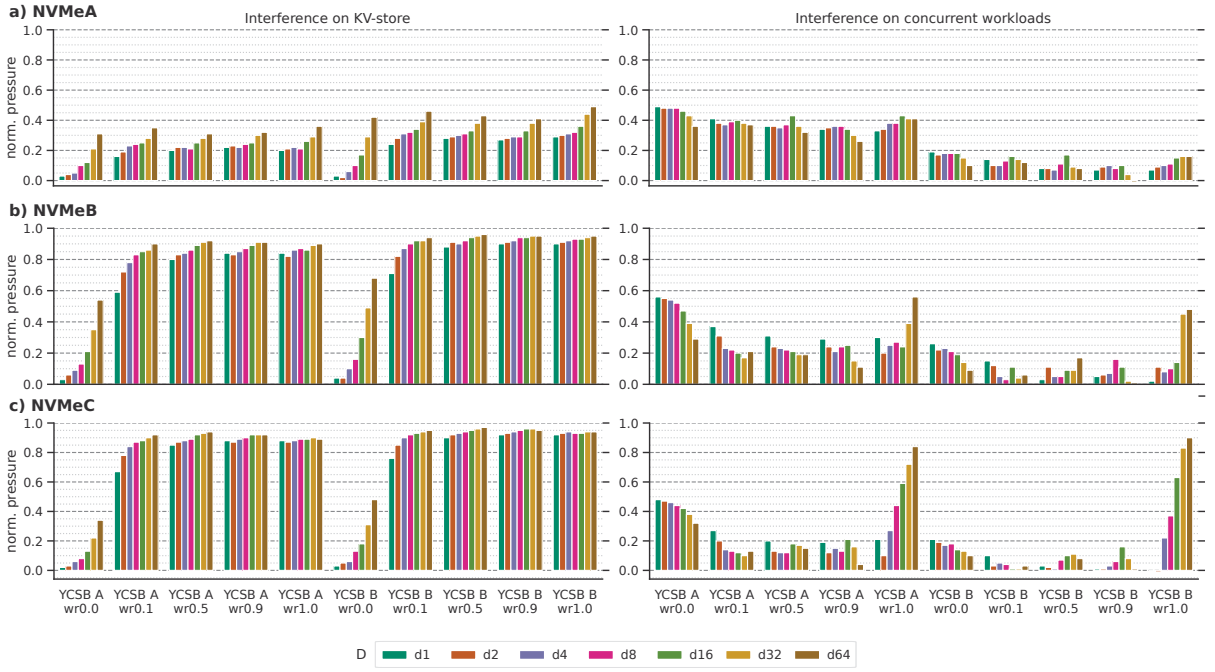


Figura 4.21: Interferência de desempenho sofrida pelas cargas de trabalho primárias e concorrentes (o_dsync = true, kernel Linux versão 5.4).

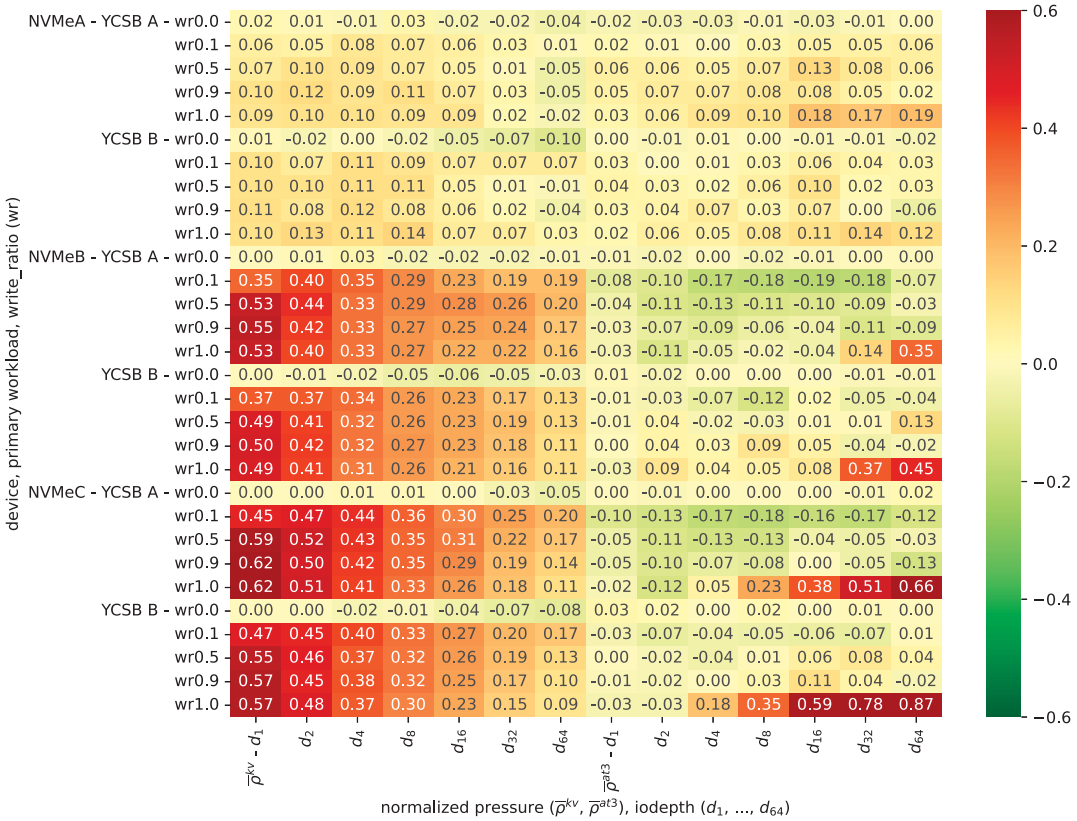


Figura 4.22: Diferença entre os valores de pressão normalizada entre as versões 5.4 e 5.11 do kernel Linux (o_dsync = true).

4.6 ANÁLISE

Os experimentos descritos neste capítulo avaliaram a interferência de desempenho exercida sobre um motor de armazenamento chave-valor em diversos cenários resultantes da combinação de diferentes dispositivos de armazenamento, cargas de trabalho primárias e concorrentes e versões do sistema operacional. Esta seção avalia como os dados obtidos nos experimentos se relacionam com a pergunta de pesquisa e hipótese descritas no Capítulo 1.

A pergunta de pesquisa Q1 questiona qual o impacto no desempenho de um motor de armazenamento chave-valor quando outras cargas de trabalho compartilham o mesmo dispositivo de armazenamento flash. As Figuras 4.12, 4.14, 4.18 e 4.21 respondem esta pergunta para os cenários de interferência de desempenho avaliados utilizando a função normalizada $\overline{\rho}^{kv}$ apresentada pela Definição 3. Embora as opções de dispositivos de armazenamento, versões do kernel Linux e cargas de trabalho que compuseram tais cenários tenham obedecido critérios de disponibilidade, capacidade e relevância, sua abrangência ainda é limitada, tanto pela grande variedade de cada uma dessas opções como pelo caráter combinatório deste problema. Neste sentido, a flexibilidade do *framework* Storiks é uma importante contribuição para que futuros trabalhos explorem outros possíveis cenários de interferência de desempenho envolvendo motores de armazenamento chave-valor.

A hipótese apresentada no Capítulo 1 considera que o impacto sobre o desempenho do motor de armazenamento chave-valor depende das cargas de trabalho, do dispositivo de armazenamento flash e do sistema operacional. Uma vez que todas essas dimensões foram cobertas pelos experimentos descritos ao longo deste capítulo, faz-se necessário avaliar se a variação de cada uma delas produziu impactos distintos sobre o desempenho do motor de armazenamento chave-valor. Do ponto de vista numérico, esta análise se concentrará nos valores normalizados de pressão ($\overline{\rho}^{kv}$) dispostos no lado esquerdo de cada um dos gráficos de interferência de desempenho apresentados nas Figuras 4.12, 4.14, 4.18 e 4.21, e na diferença de pressão normalizada apresentada nos gráficos das Figuras 4.15, 4.19 e 4.22.

4.6.1 Cargas de Trabalho Concorrentes.

As cargas de trabalho produzidas pelo `access_time3` variaram de acordo com três critérios: proporção de escritas, sincronização de dados e requisições simultâneas. Para cada um desses parâmetros, as Figuras 4.12, 4.14, 4.18 e 4.21 demonstram que houve variação na pressão normalizada produzida por essas cargas de trabalho. Na proporção de escrita (`wr`), os valores de $\overline{\rho}^{kv}$ tenderam a aumentar gradativamente nos experimentos sem sincronização de dados (Figuras 4.12 e 4.18). Nos experimentos com esta sincronização ativa (Figuras 4.14 e 4.21), este aumento progressivo de $\overline{\rho}^{kv}$ foi mais evidente entre cargas somente leitura (`wr` = 0) e as cargas de trabalho com `wr` igual a 0.1, com pequenas variações crescentes e decrescentes de $\overline{\rho}^{kv}$ entre `wr` = 0.1 e os demais valores de `wr` acima deste. As Figuras 4.23 e 4.24 apresentam esta diferença entre os valores de $\overline{\rho}^{kv}$ em função da variação de `wr` para os experimentos com as versões 5.4 e 5.11 do kernel Linux, respectivamente. Os valores positivos (em vermelho) representam um aumento da pressão normalizada em relação ao `write_ratio` de valor subsequentemente menor.

A quantidade de requisições simultâneas nas cargas de trabalho concorrentes também contribuiu para variações predominantemente crescentes de $\overline{\rho}^{kv}$. A Figura 4.25 apresenta a diferença entre os valores obtidos desta pressão em função do aumento de `iodepth`. De modo semelhante às Figuras 4.23 e 4.24, os valores positivos (em vermelho) representam um aumento de $\overline{\rho}^{kv}$ em relação à carga de trabalho concorrente com valor de `iodepth` subsequentemente menor.

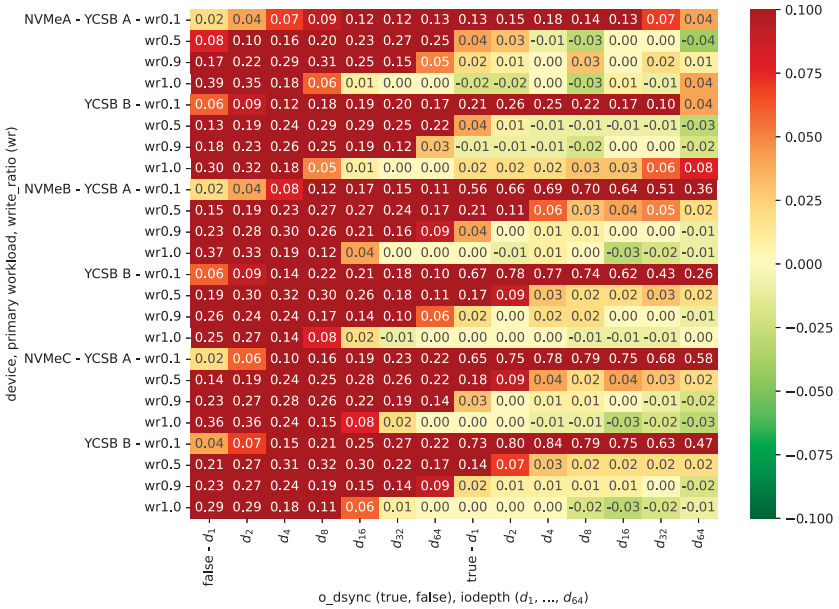


Figura 4.23: Diferença entre os valores de pressão normalizada ($\overline{\rho}^{kv}$) em função da variação de write_ratio (kernel Linux 5.4).

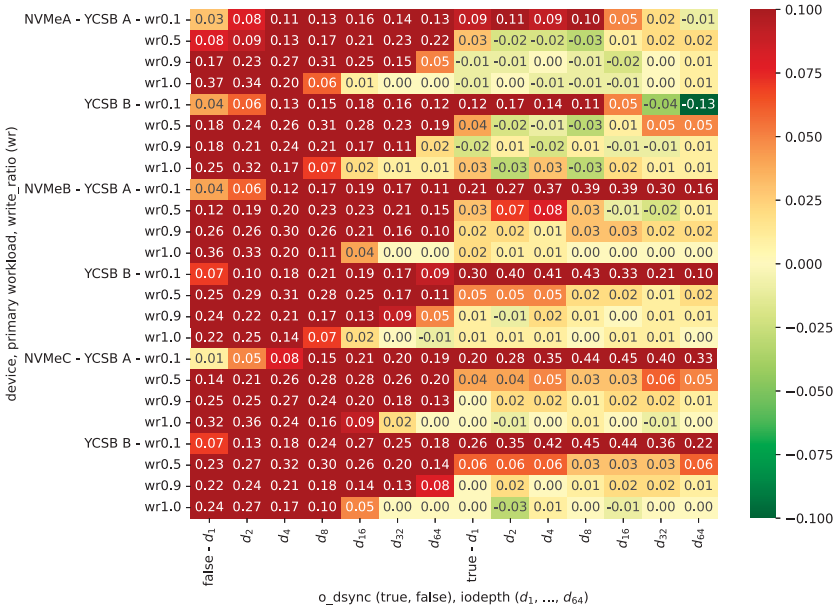
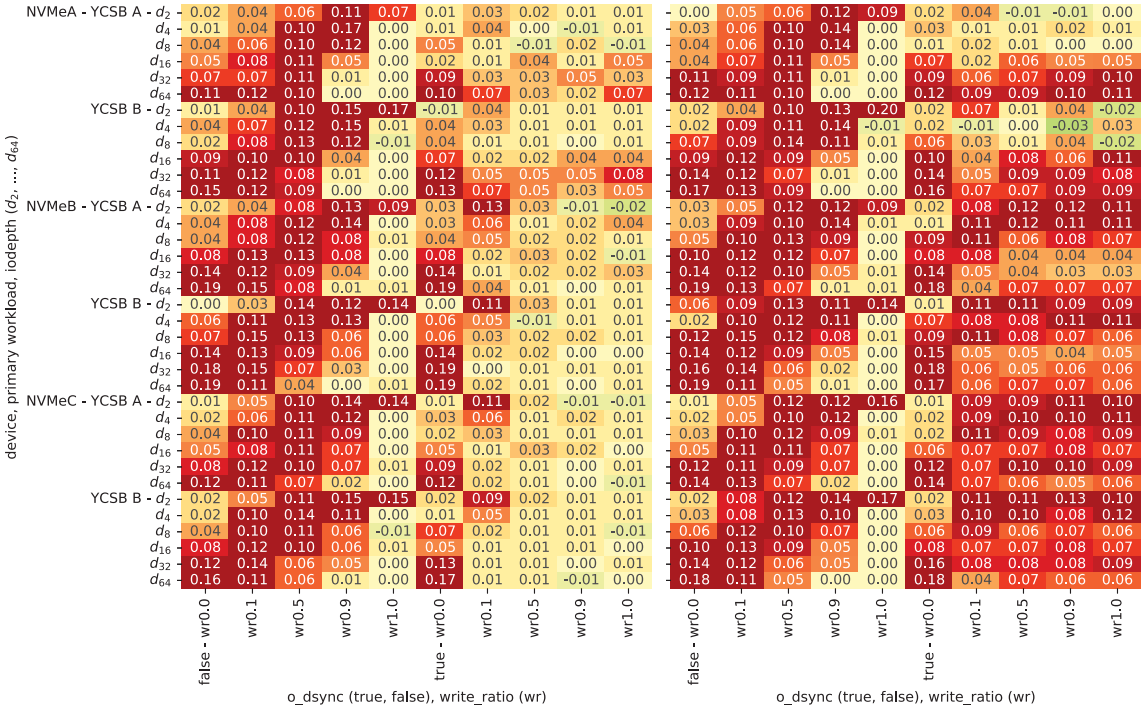


Figura 4.24: Diferença entre os valores de pressão normalizada ($\overline{\rho}^{kv}$) em função da variação de write_ratio (kernel Linux 5.11).

Para cargas de trabalho concorrentes sem sincronização de escrita (`o_dsync` = false), a Figura 4.25 destaca uma variação maior de $\overline{\rho}^{kv}$ em função do aumento de `iodepth` para valores de `wr` menores que 1. Para `wr` igual a 1 e `o_dsync` = false, destaca-se um aumento entre 9% e 20% de pressão normalizada entre d_1 e d_2, sem tendência de aumento ou diminuição de $\overline{\rho}^{kv}$ para valores de `iodepth` superiores a estes em ambas as versões de kernel avaliadas.

Já para cargas de trabalho concorrentes com sincronização de escrita, as Figuras 4.14 e 4.25b demonstram um aumento bem definido de $\overline{\rho}^{kv}$ em função do `iodepth` para os dispositivos NVMeB e NVMeC utilizando o kernel 5.11. Já com o NVMeA, este aumento de pressão normalizada é mais nítido entre d_8 e d_{64}, com variações positivas entre 2% e 12%

por acréscimo de `iodepth`. Entre d_1 e d_8, a variação de pressão normalizada não apresenta tendência clara de aumento ou redução neste dispositivo, especialmente em `wr` = 0.5, 0.9 e 1. No caso dos experimentos utilizando o kernel 5.4, embora os valores iniciais de $\overline{\rho}^{kv}$ sejam de pelo menos 59% para os dispositivos NVMeB e NVMeC e valores de `wr` maiores que 0 (Figura 4.21), a variação de pressão normalizada em função do aumento de `iodepth` ficou predominantemente abaixo de 4%, conforme demonstrado na Figura 4.25a. Algumas exceções para estes casos são observadas especialmente em `wr` = 0.1, com variações de $\overline{\rho}^{kv}$ entre 5% e 13% em d_2 e d_4.



(a) Kernel 5.4

(b) Kernel 5.11

Figura 4.25: Diferença entre os valores de pressão normalizada ($\overline{\rho}^{kv}$) em função da variação de iodepth.

4.6.2 Cargas de Trabalho Primárias.

A diferença entre os valores de pressão normalizada produzidas pelas cargas de trabalho A e B do YCSB foram menores, porém mais regulares, se comparadas com as variações de parâmetros do `access_time3`. Uma vez que a carga de trabalho B do YCSB possui uma proporção maior de transações de leitura que a carga de trabalho A, o desempenho desta primeira também tendeu a ser maior que a segunda, conforme demonstrado nas Figuras 4.8 e 4.17. Por outro lado, as Figuras 4.12, 4.14, 4.18 e 4.21 demonstram que o YCSB B foi predominantemente mais suscetível à interferência de desempenho que o YCSB A. A diferença entre os valores de $\overline{\rho}^{kv}$ obtidos com os experimentos usando o YCSB B e YCSB A é apresentada nas Figuras 4.15 e 4.26 para as versões 5.11 e 5.4 do kernel Linux, respectivamente.

Para cargas de trabalho concorrentes sem sincronização de escrita, as Figuras 4.15 e 4.26 demonstram uma maior diferença de pressão normalizada em `wr` = 0.5 e `iodepth` entre 4 e 16, com valores entre 13% e 20% superiores com o YCSB B. No caso dos experimentos com sincronização de dados e alguma proporção de escrita (`wr` > 0), observa-se uma menor diferença entre as duas cargas de trabalho primárias, sendo de até 13% com o kernel 5.4 e 15% com o kernel 5.11. Embora a presença de sincronização de escrita tenha apresentado valores elevados

de $\overline{\rho}^{kv}$ com o kernel 5.4 e dispositivos NVMeB e NVMeC (Figura 4.21), tal diferença entre o YCSB B e o YCSB A é menor se comparada ao kernel 5.11.

Conforme descrito na Seção 4.3, além das cargas de trabalho A e B do YCSB, o *framework* Storiks suporta outras cargas de trabalho primárias pertencentes ao YCSB e db_bench. Embora todas elas não tenham sido avaliadas neste estudo, futuros trabalhos sobre interferência de desempenho podem utilizar este *framework* para estender as análises apresentadas aqui a outras cargas de trabalho primárias.

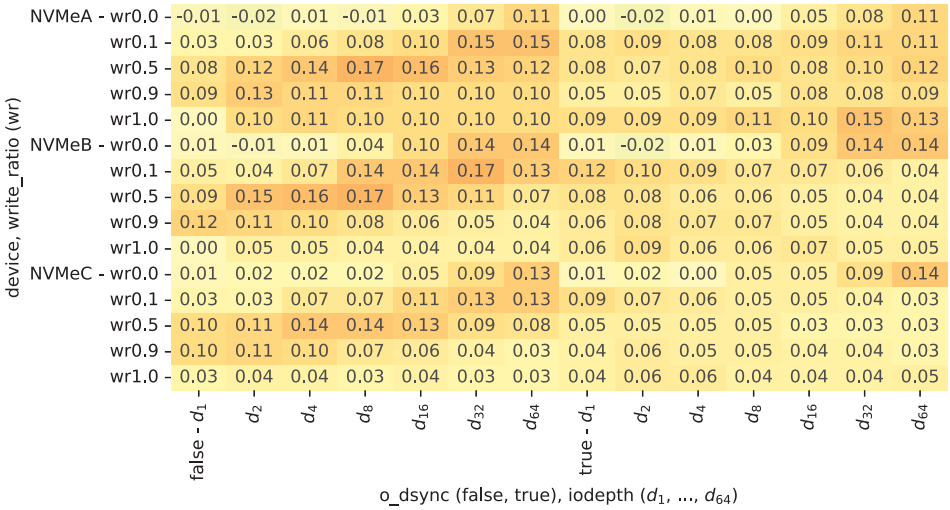


Figura 4.26: Diferença entre os valores de $\overline{\rho}^{kv}$ obtidos pelos experimentos com o YCSB B e o YCSB A para cada dispositivo de armazenamento (*device*), `write_ratio`, `o_dsync` e `iodepth` (kernel Linux versão 5.4).

4.6.3 Dispositivos de Armazenamento Flash.

Os experimentos das Figuras 4.12, 4.14, 4.18 e 4.21 também apresentam variações na pressão normalizada de acordo com os dispositivos de armazenamento avaliados. Apresentando um menor grau de variação entre eles, as Figuras 4.12 e 4.18 demonstram que o NVMeA foi o dispositivo de menor suscetividade, seguido pelo NVMeC e finalmente pelo NVMeB. Na Figura 4.14 (`o_dsync` ativo e kernel 5.11), a relação de ordem entre as pressões normalizadas dos dispositivos NVMeB e NVMeC dependeu da quantidade de requisições simultâneas na carga de trabalho concorrente. Para d_1, os valores de $\overline{\rho}^{kv}$ foram menores no dispositivo NVMeC. Já para d_{64}, esta ordem se inverteu, sendo a pressão normalizada no dispositivo NVMeB ligeiramente menor que no NVMeC para os experimentos com valores de `wr` maiores que 0.

Na Figura 4.21 (sincronização de dados ativa e kernel 5.4), os valores de $\overline{\rho}^{kv}(d_1)$ em `wr` > 0 foram menores no dispositivo NVMeB em relação ao NVMeC. Ao comparar os valores de $\overline{\rho}^{kv}(d_{64})$, esta mesma ordem se manteve em `wr` = 0.1, mas se inverteu ligeiramente em `wr` = 1. A proximidade nos valores de $\overline{\rho}^{kv}$ entre esses dois dispositivos pode estar relacionada com o uso do mesmo controlador Samsung Phoenix, conforme descrito na Tabela 4.4. Futuros trabalhos nesta área de interferência de desempenho podem se valer do *framework* Storiks para estender esta análise a outros dispositivos, inclusive de outros fabricantes.

4.6.4 Versões do Kernel Linux.

As diferenças entre os valores de pressão normalizada obtidos pelos experimentos utilizando as versões 5.4 e 5.11 do kernel Linux foram apresentas nas Figuras 4.19 e 4.22. Conforme descrito nas análises anteriores, a pressão normalizada sobre o motor de armazenamento chave-valor tem apresentado comportamentos distintos para as duas versões de kernel avaliadas, especialmente para cargas de trabalho concorrentes com sincronização de dados ativa (Figura 4.22). Para os experimentos com `wr` acima de 0, os valores de $\overline{\rho}^{kv}$ foram muito maiores no kernel 5.4 com os dispositivos NVMeB e NVMeC. No caso do dispositivo NVMeA, a pressão normalizada se mostrou maior no kernel 5.4 para valores de `iodepth` menores ou iguais a 16. Já os valores de $\overline{\rho}^{kv}(d_{64})$ variaram de ordem neste dispositivo entre as duas versões de kernel avaliadas.

A quantidade de código implementado no kernel do sistema operacional para gerenciar as requisições de E/S é relativamente alta e complexa. Isso inclui toda a parte de abstração do VFS, as implementações específicas de cada sistema de arquivos, além de questões relacionadas ao barramento e à interface física com o dispositivo de armazenamento (ex.: SATA ou NVMe). Por este motivo, os experimentos apresentados sugerem que tais variações na interferência de desempenho sofrida pelo motor de armazenamento chave-valor podem ocorrer com a simples atualização do sistema operacional ou a migração do motor de armazenamento chave-valor para outro servidor de virtualização dentro de uma mesma infraestutura de computação em nuvem. Além disso, possíveis degradações inesperadas de desempenho podem ocorrer quando alguma carga de trabalho concorrente alocada no mesmo servidor iniciar algum padrão específico de E/S. Utilizando o *framework* Storiks, futuros trabalhos neste campo podem analisar outras versões ou questões específicas de implementação do kernel do sistema operacional.

5 CONCLUSÃO

Esta tese analisou a interferência de desempenho exercida sobre um motor de armazenamento chave-valor ao compartilhar dispositivos de armazenamento persistente baseados em memória flash com outras cargas de trabalho concorrentes. O compartilhamento de recursos computacionais entre diversas aplicações ou serviços é uma prática essencial para as atuais infraestruturas de computação em nuvem, permitindo o aumento da utilização desses recursos e reduzindo os custos de aquisição e operação dos mesmos. Considerado como um efeito colateral desta prática, a interferência de desempenho precisa ser cuidadosamente analisada em cada contexto, de modo que os benefícios trazidos pela computação em nuvem não sejam invalidados mediante potenciais reduções na qualidade final dos serviços hospedados neste ambiente. A alta complexidade dos dispositivos de armazenamento baseados em memória flash impõe um desafio adicional a esta busca por equilíbrio entre compartilhamento de recursos e interferência de desempenho.

Desenvolvimento do *framework* Storiks. Visando entender melhor o problema de interferência de desempenho, este estudo propôs o desenvolvimento de um *framework* de teste e análise denominado Storiks, cuja abordagem principal se baseia em experimentos compostos por cargas de trabalho submetidas a um motor de armazenamento chave-valor e cargas de trabalho intensivas em E/S sobre um mesmo dispositivo de armazenamento. Observando diversos requisitos, como fácil utilização, portabilidade, reprodutibilidade, autonomia, significância e extensibilidade, este *framework* foi desenvolvido utilizando *containers* como principal infraestrutura de virtualização, a qual serviu tanto para o encapsulamento dos programas e serviços que o compõem como para a submissão de cargas de trabalho.

Pergunta de pesquisa. Definida como pressão, o *framework* Storiks permite responder a pergunta de pesquisa Q1, relacionando o desempenho obtido por uma carga de trabalho submetida ao motor de armazenamento chave-valor com cada carga de trabalho concorrente definida em um ou mais experimentos. Devido à grande quantidade de possíveis experimentos e o tempo elevado para a conclusão dos mesmos, este *framework* permite ainda automatizar este processo de execução, instanciando as cargas de trabalho definidas em cada experimento, verificando seu estado de conclusão e armazenando os dados obtidos. Além de medir o desempenho do motor de armazenamento, este *framework* coleta ainda uma enorme variedade de contadores e informações estatísticas provenientes deste motor, do sistema operacional e do *hardware*. Tais informações podem ser posteriormente analisadas pelo usuário por meio de um ambiente Jupyter Notebook interno.

Para a geração de cargas de trabalho submetidas ao motor de armazenamento chave-valor, o *framework* Storiks incorporou duas ferramentas de *benchmark* conhecidas, YCSB e db_bench, ambas utilizando o RocksDB como motor de armazenamento. Como forma de melhorar a extração de informações sobre o estado interno da árvore LSM utilizada para armazenar o banco de dados chave-valor, o RocksDB foi modificado ao longo do desenvolvimento deste *framework* para incluir um canal de comunicação entre este motor e o mecanismo de controle e coleta de estatísticas dos experimentos.

A geração de cargas de trabalho concorrentes foi contemplada com o desenvolvimento de um *microbenchmark* flexível chamado `access_time3`. Projetado para apresentar um baixo consumo de memória e CPU, o desenvolvimento deste *microbenchmark* considerou os diversos níveis de abstração e APIs de sistema operacional que integram uma arquitetura típica de armazenamento persistente local, incluindo desde a forma com que os dados são endereçados no

dispositivo de armazenamento até a organização do sistema de arquivos e as chamadas de sistema disponibilizadas pelo VFS. Como resultado final deste desenvolvimento, o `access_time3` proporciona uma grande variedade de padrões de acesso sobre o dispositivo de armazenamento, podendo alterar esses padrões ao longo de um experimento conforme especificado pelo usuário. As informações de desempenho coletadas pelo `access_time3` são ainda armazenadas pelo *framework* Storiks junto com as demais informações e estatísticas de desempenho de cada experimento.

Além da escolha das cargas de trabalho, a metodologia de teste apresentada neste estudo também obedeceu diversos critérios relacionados com as características de *hardware* e *software* do ambiente computacional avaliado e com a estabilidade de desempenho deste ambiente durante a realização dos experimentos. Esta estabilidade foi considerada fundamental para a obtenção de valores comparáveis e reprodutíveis de interferência de desempenho entre diferentes cargas de trabalho concorrentes. Os critérios observados para proporcionar tal estabilidade incluíram a preparação e o pré-condicionamento do dispositivo de armazenamento, do banco de dados chave-valor e dos arquivos utilizados para receberem as cargas de trabalho. Fornecidas pelo *framework* Storiks, informações estatísticas importantes sobre os estados internos do dispositivo de armazenamento e da árvore LSM que armazena o banco de dados chave-valor permitem ao usuário verificar se o ambiente experimental apresenta tais condições de estabilidade.

Hipótese. Utilizando o *framework* Storiks, este estudo avaliou diversas condições de interferência de desempenho, abrangendo três dispositivos de armazenamento baseados em memória flash, duas versões do kernel Linux, duas cargas de trabalho submetidas ao motor de armazenamento chave-valor e setenta cargas de trabalho concorrentes intensivas em E/S. Resultando em um conjunto de 840 casos de observação, este *framework* foi capaz de expor quantitativa e comparativamente a influência de cada uma das dimensões avaliadas. A partir destes experimentos, foi possível demonstrar que tal influência sobre o desempenho do motor de armazenamento chave-valor pode apresentar padrões distintos dependendo do tipo de carga de trabalho, dispositivo de armazenamento e versão do sistema operacional.

Considerando as setenta cargas de trabalho concorrentes avaliadas ao longo deste estudo, observou-se que a interferência de desempenho gerada por elas foi predominantemente crescente em função do aumento da proporção de escritas e do número de requisições concorrentes, com exceção de alguns casos de estabilidade ou de inversões desta ordem. Contudo, tal comportamento se mostrou divergente tanto em proporcionalidade como nos casos de exceção à medida que os demais fatores foram avaliados, como o uso ou ausência de sincronização de dados nessas cargas de trabalho concorrentes, o tipo de dispositivo de armazenamento, a versão do kernel Linux e a carga de trabalho submetida ao motor de armazenamento.

Os dispositivos de armazenamento flash avaliados representam três gerações diferentes de um mesmo fabricante, os quais variaram também em capacidade e tipo de controlador interno. Os experimentos realizados com o motor de armazenamento chave-valor executando isoladamente demonstraram desempenhos relativamente próximos entre esses dispositivos, havendo uma pequena margem de vantagem para o dispositivo de maior capacidade e para o de tecnologia mais recente. Em contrapartida, a absorção de cargas de trabalho concorrentes não seguiu esta mesma proporcionalidade, sendo o dispositivo mais recente predominantemente menos vulnerável neste contexto de interferência de desempenho.

Os diferentes padrões e intensidades de interferência de desempenho observados neste estudo reforçam a importância desta avaliação ao considerar o uso de motores de armazenamento chave-valor em ambientes onde os recursos de armazenamento persistente são compartilhados. Embora as cargas de trabalho, dispositivos de armazenamento e versões de sistema operacional avaliados aqui tenham sido escolhidos de acordo com critérios de representatividade e relevância,

tais fatores representam apenas uma região muito limitada deste vasto espaço de possíveis opções de infraestrutura. Contudo, o desenvolvimento do *framework* Storiks é uma contribuição relevante para que outros cenários possam ser explorados futuramente, simplificando a execução e análise de novos experimentos.

5.1 LIMITAÇÕES E PERSPECTIVAS FUTURAS

Esta tese integra a concepção de que ambientes de recursos computacionais compartilhados são pouco considerados durante o ciclo de vida de motores de armazenamento chave-valor, incluindo desde o projeto inicial até sua implantação em ambientes de nuvem. Do ponto de vista de projeto, embora diversos estudos tenham sido apresentados para melhorar a adaptabilidade de árvores LSM para diferentes composições de cargas de trabalho, quantidade de memória disponível e volume de dados armazenados, tais propostas ignoram as variações de desempenho provenientes de cargas de trabalho concorrentes. Uma oportunidade de pesquisa que merece destaque neste sentido é o desenvolvimento de árvores LSM mais resilientes a essas variações de desempenho, definindo regras adaptáveis de compactação entre níveis e de alocação de memória conforme a disponibilidade de recursos. O *framework* Storiks visa ser uma importante contribuição para este desenvolvimento, permitindo mapear esta relação entre cargas de trabalho concorrentes e desempenho.

Do ponto de vista de implantação em ambientes compartilhados, a estratégia comumente aplicada nos dias atuais é reservar recursos para impedir que cargas de trabalho concorrentes afetem o desempenho do motor de armazenamento chave-valor. Uma vez que esta estratégia deriva de um projeto que ignora o uso compartilhado de recursos, o desenvolvimento de abordagens mais resilientes às variações de desempenho abre margem para estratégias mais flexíveis de reserva de recursos. Como contribuição, o *framework* Storiks pode ser usado para definir os limites desta flexibilização para cada tipo de dispositivo de armazenamento e versão de sistema operacional utilizados, além de permitir a avaliação de outras opções de infraestrutura, como a utilização de diferentes tipos de RAID e gerenciadores de volumes lógicos (LVMs).

Em sua versão atual, o *framework* Storiks utiliza *containers* como tecnologia de virtualização e apenas o RocksDB como motor de armazenamento chave-valor. Embora tais opções sejam relevantes e amplamente utilizadas nos dias atuais, a incorporação de outras técnicas de virtualização e motores de armazenamento chave-valor, incluindo diferentes estruturas de armazenamento internas, fica a cargo de trabalhos futuros. Avaliar como cada uma dessas opções é afetada por cargas de trabalho concorrentes também representa um direcionamento de pesquisa bastante interessante.

A metodologia adotada neste estudo para a realização dos experimentos buscou priorizar a estabilidade do ambiente computacional e evitar que medições de desempenho sejam realizadas em situações com tendências de aumento ou diminuição. Neste sentido, tanto a escolha das cargas de trabalho como a preparação do ambiente experimental seguiram critérios rigorosos para garantir esta estabilidade. Embora os resultados apresentados sejam importantes para traçar um panorama geral de interferência de desempenho, os ambientes de produção são mais dinâmicos e menos sujeitos ao controle de cada uma das partes envolvidas. A extensão do *framework* Storiks para ambientes mais dinâmicos é uma importante oportunidade de trabalhos futuros. Além disso, outras técnicas estatísticas e de aprendizagem de máquina podem ser incorporadas a este *framework* com o objetivo de priorizar ou reduzir a quantidade de experimentos, diminuindo assim o desgaste excessivo dos dispositivos de armazenamento avaliados.

REFERÊNCIAS

[1] Apache HBase. http://hbase.apache.org. Acessado Fev. 2022.

[2] Apache Cassandra. https://cassandra.apache.org. Acessado Fev. 2022.

[3] Aio(7) - Linux man page. https://linux.die.net/man/7/aio. Acessado em Fev. 2022.

[4] Apache Hadoop. http://hadoop.apache.org. Acessado Fev. 2022.

[5] Debian Wiki – SSD Optimization. https://wiki.debian.org/SSDOptimization. Acessado Fev. 2022.

[6] Matplotlib. https://matplotlib.org. Acessado Fev. 2022.

[7] NumPy. https://numpy.org. Acessado Fev. 2022.

[8] Pandas. https://pandas.pydata.org. Acessado Fev. 2022.

[9] Seaborn. https://seaborn.pydata.org. Acessado Fev. 2022.

[10] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis Tsotras, Rares Vernica, Jian Wen, and Till Westmann. AsterixDB: A Scalable, Open Source BDMS. *Proc. VLDB Endow.*, 7(14):1905–1916, oct 2014.

[11] Amazon. Choosing the Right EC2 Instance Type for Your Application. https://aws.amazon.com/pt/blogs/aws/choosing-the-right-ec2-instance-type-for-your-application/. Acessado Jan. 2023.

[12] AMD. Ryzen™ 5 3600. https://www.amd.com/en/products/cpu/amd-ryzen-5-3600. Acessado Fev. 2022.

[13] AnandTech. The Samsung 970 EVO Plus (250GB, 1TB) NVMe SSD Review: 92-Layer 3D NAND. https://www.anandtech.com/show/13761/the-samsung-970-evo-plus-ssd-review. Acessado Fev. 2022.

[14] ASRock. B550M Phantom Gaming 4. https://www.asrock.com/mb/AMD/B550M%20Phantom%20Gaming%204. Acessado Fev. 2022.

[15] Manos Athanassoulis, Michael S Kester, Lukas M Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. Designing Access Methods: The RUM Conjecture. In *Proceedings of the 19th International Conference on Extending Database Technology (EDBT)*, pages 461–466, 2016.

[16] Jens Axboe. Fio: Flexible I/O Tester. https://github.com/axboe/fio. Acessado Fev. 2022.

[17] Burton H Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, 1970.

[18] John S. Bucy, Jiri Schindler, Steven W. Schlosser, and Gregory R. Ganger. The DiskSim Simulation Environment Version 4.0 Reference Manual, 2008. https://citeseer.ist.psu.edu/viewdoc/download?doi=10.1.1.218.6649&rep=rep1&type=pdf.

[19] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H C Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. pages 209–223. USENIX Association, 2020.

[20] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26, 2008.

[21] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, pages 273–286. USENIX Association, 2005.

[22] Cockroach Labs. CockroachDB. https://www.cockroachlabs.com. Acessado Fev. 2022.

[23] Cockroach Labs. Pepple. https://github.com/cockroachdb/pebble. Acessado Fev. 2022.

[24] Brian F Cooper. YCSB repository. https://github.com/brianfrankcooper/YCSB. Acessado Fev. 2022.

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

[26] Corsair. VENGEANCE® LPX 16GB (2 x 8GB) DDR4 DRAM 3200MHz C16 Memory. https://www.corsair.com/us/en/Categories/Products/Memory/VENGEANCE-LPX/p/CMK16GX4M2B3200C16. Acessado Fev. 2022.

[27] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 79–94, New York, NY, USA, 2017. Association for Computing Machinery.

[28] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *ACM Trans. Database Syst.*, 43(4), 2018.

[29] Niv Dayan and Stratos Idreos. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 505–520, New York, NY, USA, 2018. Association for Computing Machinery.

[30] Niv Dayan and Stratos Idreos. The Log-Structured Merge-Bush & the Wacky Continuum. In *2019 International Conference on Management of Data (SIGMOD '19)*, pages 449–466, New York, NY, USA, jun 2019. ACM.

[31] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, page 77–88. ACM, 2013.

[32] Diego Didona, Nikolas Ioannou, Radu Stoica, and Kornilios Kourtis. Toward a Better Understanding and Evaluation of Tree Structures on Flash SSDs. *Proceedings of the VLDB Endowment*, 14:364–377, 2021.

[33] Docker. Docker: Accelerated, Containerized Application Development. `https://www.docker.com`. Acessado Fev. 2022.

[34] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Stumm. Optimizing space amplification in rocksDB. *CIDR 2017 - 8th Biennial Conference on Innovative Data Systems Research*, 2017.

[35] Facebook. MyRocks: A RocksDB storage engine with MySQL. `http://myrocks.io`. Acessado Fev. 2022.

[36] Facebook. RocksDB – Basic Operations. `https://github.com/facebook/rocksdb/wiki/Basic-Operations`. Acessado Fev. 2022.

[37] Facebook. RocksDB – BlockBasedTable Format. `https://github.com/facebook/rocksdb/wiki/Rocksdb-BlockBasedTable-Format`. Acessado Fev. 2022.

[38] Facebook. RocksDB – Compaction. `https://github.com/facebook/rocksdb/wiki/Compaction`. Acessado Fev. 2022.

[39] Facebook. RocksDB – Leveled Compaction. `https://github.com/facebook/rocksdb/wiki/Leveled-Compaction`. Acessado Fev. 2022.

[40] Facebook. RocksDB – Use Cases. `https://github.com/facebook/rocksdb/wiki/RocksDB-Users-and-Use-Cases`. Acessado Fev. 2022.

[41] Facebook. RocksDB repository. `http://facebook.github.io/zstd`. Acessado Fev. 2022.

[42] Facebook. ZippyDB. `https://engineering.fb.com/2021/08/06/core-data/zippydb`. Acessado Fev. 2022.

[43] Facebook. Zstandard. `https://github.com/facebook/rocksdb`. Acessado Fev. 2022.

[44] Sanjay Ghemawat and Jeff Dean. LevelDB repository. `https://github.com/google/leveldb`. Acessado Fev. 2022.

[45] Google. How To Use Gflags. `https://gflags.github.io/gflags`. Acessado Fev. 2022.

[46] Google. Machine families resource and comparison guide. https://cloud.google.com/compute/docs/machine-resource#recommendations_for_machine_types. Acessado Jul. 2023.

[47] Donghyun Gouk, Miryeong Kwon, Jie Zhang, Sungjoon Koh, Wonil Choi, Nam Sung Kim, Mahmut Kandemir, and Myoungsoo Jung. Amber*: Enabling precise full-system simulation with detailed modeling of all SSD resources. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, 2018-Octob:469–481, 2018.

[48] Stratos Idreos and Mark Callaghan. Key-Value Storage Engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2667–2672, New York, NY, USA, jun 2020. ACM.

[49] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, and Zichen Zhu. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn. In *9th Biennial Conference on Innovative Data Systems Research (CIDR '19)*, 2019.

[50] Stratos Idreos, Kostas Zoumpatianos, Manos Athanassoulis, Niv Dayan, Brian Hentschel, Michael S Kester, Demi Guo, Lukas M Maas, Wilson Qin, Abdul Wasay, and Yiyou Sun. The Periodic Table of Data Structures. *IEEE Data Eng. Bull.*, 41(3):64–75, 2018.

[51] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S Kester, and Demi Guo. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 535–550, New York, NY, USA, 2018. Association for Computing Machinery.

[52] Intel. HiBench: The bigdata micro benchmark suite.

[53] Jupyter. Project Jupyter. https://jupyter.org. Acessado Fev. 2022.

[54] Ram Srivatsa Kannan, Michael Laurenzano, Jeongseob Ahn, Jason Mars, and Lingjia Tang. Caliper: Interference Estimator for Multi-Tenant Environments Sharing Architectural Resources. *ACM Trans. Archit. Code Optim.*, 16, 2019.

[55] Jaeho Kim, Donghee Lee, and Sam H. Noh. Towards SLO Complying SSDs Through OPS Isolation. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 183–189. USENIX Association, 2015.

[56] Miryeong Kwon, Donghyun Gouk, Changrim Lee, Byounggeun Kim, Jooyoung Hwang, and Myoungsoo Jung. DC-Store: Eliminating Noisy Neighbor Containers using Deterministic I/O Performance and Resource Isolation. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 183–191. USENIX Association, 2020.

[57] Adriano Lange. Imagem de container do framework Storiks, 2022. https://hub.docker.com/r/alange0001/storiks.

[58] Adriano Lange. Performancemonitor repository, 2022. https://github.com/alange0001/performancemonitor.

[59] Adriano Lange. Storiks Project, 2022. https://github.com/alange0001/storiks.

[60] Adriano Lange, Tiago Rodrigo Kepe, Eduardo Cunha de Almeida, and Marcos Sfair Sunye. Uncovering performance interference of multi-tenants in big data environments. In *2020 IEEE International Conference on Big Data (IEEE BigData 2020), Atlanta, GA, USA, December 10-13, 2020*, pages 2773–2778. IEEE, 2020.

[61] Adriano Lange, Tiago Rodrigo Kepe, and Marcos Sfair Sunye. Performance interference on key-value stores in multi-tenant environments: When block size and write requests matter. In *ICPE'21: ACM/SPEC International Conference on Performance Engineering, Virtual Event, France, April 19-21, 2021, Companion Volume*, pages 89–92. ACM, 2021.

[62] David B. Lomet. Cost/performance in modern data stores: how data caching systems succeed. In *Proceedings of the 14th International Workshop on Data Management on New Hardware, Houston, TX, USA, June 11, 2018*, pages 9:1—-9:10. ACM, 2018.

[63] Chen Luo and Michael J. Carey. On performance stability in LSM-based storage systems. *Proceedings ofthe VLDB Endowment*, 13(4):449–462, 2019.

[64] Chen Luo and Michael J. Carey. LSM-based storage techniques: a survey. In *VLDB Journal*, volume 29, pages 393–418. Springer, jan 2020.

[65] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-Locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, page 248–259. Association for Computing Machinery, 2011.

[66] Microsoft. Storage: Performance best practices for SQL Server on Azure VMs. `https://learn.microsoft.com/en-us/azure/azure-sql/virtual-machines/windows/performance-guidelines-best-practices-storage`. Acessado Jun. 2023.

[67] Microsoft. VM size: Performance best practices for SQL Server on Azure VMs. `https://learn.microsoft.com/en-us/azure/azure-sql/virtual-machines/windows/performance-guidelines-best-practices-vm-size`. Acessado Abr. 2023.

[68] MongoDB. MongoDB. `http://www.mongodb.com`. Acessado Fev. 2022.

[69] MongoDB. WiredTiger Storage Engine. `https://www.mongodb.com/docs/manual/core/wiredtiger`. Acessado Fev. 2022.

[70] Jeff Moyer. Ensuring data reaches disk, 2011. `https://lwn.net/Articles/457667`. Acessado Fev. 2022.

[71] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 219–230. USENIX Association, 2013.

[72] NVM Express Inc. NVM Express Base Specification. Technical report, 2019. `https://nvmexpress.org/specification/nvm-express-base-specification`.

[73] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33:351–385, 1996.

[74] Tarikul Islam Papon and Manos Athanassoulis. A Parametric I/O Model for Modern Storage Devices. In *Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN 2021)*. ACM, 2021.

[75] Mark Rogov, Chris Conniff, and Tim Lustig. Storage Performance Benchmarking: Part 5 - Workloads. https://www.snia.org/sites/default/files/ESF/Storage%20Performance%20Benchmarking%20Workloads.pdf. Acessado Jan. 2023.

[76] Mohammad Sadrosadati and Onur Mutlu. Fine-Grained Mapping & Multi-Plane Operation-Aware Block Management. https://safari.ethz.ch/projects_and_seminars/fall2022/lib/exe/fetch.php?media=pns_modern_ssds_fall2022_9th_after_meeting.pdf, http://www.youtube.com/watch?v=MwQKHXfIWfA. Acessado dez. 2022.

[77] Samsung. Over-Provisioning Benefits for Samsung Data Center SSDs. https://semiconductor.samsung.com/resources/white-paper/S190311-SAMSUNG-Memory-Over-Provisioning-White-paper.pdf. Acessado Fev. 2022.

[78] Samsung. Samsung 970 EVO. https://semiconductor.samsung.com/consumer-storage/internal-ssd/970evo. Acessado Fev. 2022.

[79] Samsung. Samsung 970 EVO Plus. https://semiconductor.samsung.com/consumer-storage/internal-ssd/970evoplus. Acessado Fev. 2022.

[80] Samsung. Samsung 980 PRO. https://semiconductor.samsung.com/consumer-storage/internal-ssd/980pro. Acessado Fev. 2022.

[81] Daniel Seybold, Moritz Keppler, Daniel Gründler, and Jörg Domaschka. Mowgli: Finding your way in the dbms jungle. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering - ICPE '19*, pages 321–332. ACM Press, 2019.

[82] SNIA. Reference Real World Workloads. https://www.snia.org/technology-focus-areas/physical-storage/real-world-workloads/reference-real-world-workloads. Acessado Jan. 2023.

[83] SNIA. Solid State Storage (SSS) Performance Test Specification (PTS) Version 2.0.2, 2020. https://www.snia.org/sites/default/files/technical_work/PTS/SSS_PTS_2.0.2.pdf.

[84] SPEC. SPEC Cloud IaaS 2018. https://www.spec.org/cloud_iaas2018, 2018.

[85] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 1493–1509, New York, NY, USA, 2020. Association for Computing Machinery.

[86] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. MQSIM: A framework for enabling realistic studies of modern multi-queue SSD devices. *Proceedings of the 16th USENIX Conference on File and Storage Technologies, FAST 2018*, pages 49–65, 2018.

[87] The kernel development community. Ext4 Data Structures and Algorithms. `https://www.kernel.org/doc/html/latest/filesystems/ext4/dynamic.html`. Acessado em Fev. 2022.

[88] The kernel development community. Ext4: Dynamic Structures. `https://www.kernel.org/doc/html/latest/filesystems/ext4/dynamic.html`. Acessado em Fev. 2022.

[89] The kernel development community. Overview of the Linux Virtual File System. `https://www.kernel.org/doc/html/latest/filesystems/vfs.html`. Acessado Fev. 2022.

[90] The kernel development community. The SGI XFS Filesystem. `https://docs.kernel.org/admin-guide/xfs.html`. Acessado Jan. 2023.

[91] Transaction Processing Performance Council. TPC-C: On-Line Transaction Processing Benchmark. Technical report. `http://www.tpc.org/tpcc/default.asp`.

[92] Transaction Processing Performance Council. TPC-H: Decision Support Benchmark. Technical report. `http://www.tpc.org/tpch/default.asp`.

[93] Daniel Wagner, Keith Busch, Jeff Lien, Hannes Reinecke, and Minwoo Im. NVMe management command line interface. `https://github.com/linux-nvme/nvme-cli`. Acessado Fev. 2022.

[94] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, Chen Zheng, Gang Lu, Kent Zhan, Xiaona Li, and Bizhu Qiu. BigDataBench: A big data benchmark suite from internet services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–499. IEEE, feb 2014.

[95] Ran Xu, Subrata Mitra, Jason Rahman, Peter Bai, Bowen Zhou, Greg Bronevetsky, and Saurabh Bagchi. Pythia: Improving datacenter utilization via precise contention prediction for multiple co-located workloads. In *Proceedings of the 19th International Middleware Conference*, page 146–160. ACM, 2018.

[96] Gala Yadgar, Moshe Gabel, Shehbaz Jaffer, and Bianca Schroeder. SSD-Based Workload Characteristics and Their Performance Implications. *ACM Trans. Storage*, 17, 2021.

[97] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, page 607–618. ACM, 2013.

[98] Hobin Yoon, Juncheng Yang, Sveinn Fannar Kristjansson, Steinn E Sigurdarson, Ymir Vigfusson, and Ada Gavrilovska. Mutant: Balancing Storage Cost and Latency in LSM-Tree Data Stores. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*, pages 162–173. ACM, 2018.