Headless Flash Disk Tree: Uma Estrutura Dinâmica em SSD para Dados Unidimensionais

Diogo Sobral Forasteiro[1], Thiago F. M. Silva[1], Rodrigo Duarte Seabra[1], Lúcio F. D. Santos[2], Enzo Seraphim[1], Luiz Olmes Carvalho[1]

[1]Universidade Federal de Itajubá (UNIFEI) - Itajubá (MG), Brasil

[2]Instituto Federal do Norte de Minas Gerais (IFNMG) - Montes Claros (MG), Brasil

{diogosforasteiro, thiagofernandes1568}@gmail.com

lucio.santos@ifnmg.edu.br, {rodrigo, seraphim, olmes}@unifei.edu.br

Abstract. *The shift from hard disks to solid-state drives led to the development of new data structures that deal with their specific characteristics. To mitigate the effects of constraints such as "erase-before-write", data structures were designed to reduce their impact. In this context, this paper introduces the Headless Flash Disk Tree, a dynamic structure for unidimensional data following a total order relation, which improves insert and query performance. The evaluation used a real-world dataset, and the results show that the proposal is up to 16% faster than competitors for inserts and up to 8% faster for queries.*

Resumo. *Com a transição do uso de discos rígidos para unidades de estado sólido, tornou-se necessário desenvolver novas estruturas de dados que considerem as suas características. Para mitigar os efeitos de restrições como o "apagar antes de escrever", foram projetadas estruturas de dados capazes de reduzir seu impacto. Nesse contexto, este artigo apresenta a Headless Flash Disk Tree, uma estrutura dinâmica para dados unidimensionais que seguem relação de ordem total, e que melhora o desempenho de operações de inserção e consulta. A avaliação foi realizada usando um conjunto de dados real, e os resultados mostraram que a proposta é até 16% mais rápida que as concorrentes para inserir dados e até 8% mais rápida para realizar consultas.*

1. Introdução

Com a evolução dos dispositivos de armazenamento, os tradicionais discos rígidos estão sendo gradativamente substituídos pelas unidades de estado sólido (*Solid-State Drive* - SSD), devido às suas principais vantagens: ausência de partes mecânicas, menor tamanho, menor consumo de energia e, principalmente, maiores velocidades de leitura e escrita de dados [Haas and Leis 2023, Zhang et al. 2025].

Entretanto, as memórias *flash*, que são a base da construção dos SSDs, impõem restrições que devem ser consideradas ao se desenvolver soluções que os utilizam. Em um sistema de Banco de Dados, uma destas principais restrições é denotada por "apagar antes de escrever" (*erase-before-write*): uma página só pode ser atualizada (reescrita) após todo o seu bloco (conjunto de páginas) ser apagado [Hassani et al. 2026]. Isto faz com que a atualização de um dado já escrito em uma página dispare sucessivas operações de leitura e escrita em novas páginas [Boukhobza et al. 2025].

O "*erase-before-write*" compromete, principalmente, o desempenho das estruturas de indexação voltadas para discos rígidos como a B-Tree e a B^+-Tree, visto que elas realizam atualizações frequentes nos nós. Portanto, diversas estruturas de índice foram desenvolvidas para operar sobre SSDs [Fevgas et al. 2019]. Entretanto, a literatura não define qual é a melhor estrutura, mas sim, qual estrutura é a mais indicada para tipos de aplicação e de dados específicos. Por exemplo, para dados unidimensionais (números, datas e *strings*), o melhor índice para aplicações que priorizam a operação de leitura ainda é a tradicional B^+-Tree [Comer 1979]; para sistemas onde há restrição de memória RAM, a WPCB-Tree [Ho and Park 2018] se sobressai; em sistemas *multi-thread*, onde a concorrência é um aspecto essencial, a Bw-Tree é a estrutura mais adequada [Levandoski et al. 2013]; enquanto que a FD-Tree [Li et al. 2010] é indicada para situações onde a operação de escrita e atualização de nós é realizada com frequência.

Em um ambiente OLTP (*Online Transaction Processing*), onde o volume de transações e de escritas em disco é alto, a FD-Tree alcança alto desempenho. Parte da eficiência desta estrutura se deve ao seu princípio de funcionamento: reunir várias operações de escrita aleatória de pequenas quantidades de dados e executar a gravação em lote (*batch*), evitando atualizações daquela mesma página. Para isso, a estrutura mantém uma primeira camada (*head-tree*) composta por uma B^+-Tree em memória primária, enquanto as camadas seguintes formam nós potencialmente grandes em tamanho, armazenados no SSD, e que mantêm a relação de ordem total entre os elementos [Li et al. 2010].

Neste contexto, este trabalho investiga a hipótese de que "a head-tree impacta no desempenho da estrutura", visto que é necessário gerenciar as operações de inserção e divisão de nós (*split*) de uma B^+-Tree. Além disso, uma outra restrição da FD-Tree é que os seus nós possuem uma taxa de ocupação máxima, para que a estrutura seja capaz de realizar a escrita em lote quando esta taxa for atingida. Esta segunda restrição é problemática para tipos de dados que não possuem tamanho fixo predefinido (e.g. *strings*).

Tendo como foco uma estrutura que prioriza a operação de escrita, o objetivo deste trabalho é introduzir a *Headless Flash Disk Tree* (HFD-Tree), uma estrutura dinâmica destinada à indexação de dados unidimensionais sobre SSDs. A HFD-Tree segue os mesmos princípios de construção da FD-Tree, porém, a otimiza nos seguintes aspectos:

- *Ausência da head-tree*: não há uma B^+-Tree em memória RAM apontando para a camada seguinte no SSD, mas sim nós de HFD-Tree, gerenciados por uma sessão.
- *Controle de sessão*: os nós iniciais da árvore são mantidos em RAM enquanto estão ativos na *sessão* atual. Ao término da sessão, eles são persistidos no SSD.
- *Modelagem dos nós*: os nós mantêm referências para os objetos (chaves) de modo a permitir o armazenamento de tipos de dados com tamanho variável (e.g. *strings*).
- *Ausência de taxa máxima de ocupação*: dados são persistidos em cada nó enquanto houver espaço livre disponível.

Os experimentos foram conduzidos utilizando um conjunto de dados de palavras e, portanto, de tamanho variável em *bytes*. Os resultados encontrados evidenciam que a HFD-Tree melhora o desempenho da operação de escrita em até 16% e a operação de consulta em até 8% em relação à árvore original sem otimizações.

Este trabalho está organizado da seguinte forma: a Seção 2 apresenta o referencial literário. A Seção 3 introduz a HFD-Tree, sua modelagem e otimizações. A Seção 4 mostra os experimentos e os resultados. A Seção 5 traz a conclusão e as propostas futuras.

2. Referencial Teórico

Uma das formas de classificação dos índices desenvolvidos para SSDs é pela dimensionalidade dos dados que eles armazenam: uni ou multidimensionais [Fevgas et al. 2019].

Para dados unidimensionais, as estruturas mais comuns são derivações da B^+-Tree, com modificações em seus nós. Para cenários com alta taxa de escrita, foco deste trabalho, destacam-se a AB-Tree [Jiang et al. 2014], a Bw-Tree [Levandoski et al. 2013], a LSM-Tree [Teng et al. 2018] e a FD-Tree [Li et al. 2010]. Enquanto a AB-Tree implementa escrita em *buckets* e gravações em lote, a Bw-Tree e a LSM-Tree oferecem desempenho satisfatório, aliado à capacidade de gerenciar operações concorrentes. Já a FD-Tree é a que atinge melhor desempenho entre todas elas. Neste trabalho, a FD-Tree é considerada como base para a geração de um novo índice otimizado para escrita.

Ainda na categoria de dados unidimensionais, também existem estruturas de tabelas de espalhamento para SSDs, como a *h-hash* [Kim et al. 2015], mas o foco destas abordagens é a operação de busca, e não de escrita de dados, o que difere desta proposta.

Para dados multidimensionais, existem estruturas destinadas à indexação de pontos e de objetos geométricos no espaço. Destas, a xBR^+-Tree [Roumelis et al. 2018] e a LB-Grid [Fevgas and Bozanis 2019] destacam-se como estruturas que realizam o processamento em lote, de modo a otimizar as operações de entrada e saída. Já as propostas e-Find [Carniel et al. 2019] e FAST [Sarwat et al. 2011] são *frameworks* genéricos para otimizar índices espaciais em memória *flash*. A principal diferença destas propostas para a HFD-Tree é a dimensionalidade do dado. Na presente proposta, os dados são unidimensionais e seguem uma relação de ordem total, o que auxilia na busca pelas chaves.

3. A Árvore HFD

A *Headless Flash Disk Tree* é uma estrutura dinâmica, isto é, permite inserções a qualquer momento, e balanceada, com todos os nós do último nível na mesma altura. As chaves de busca são mantidas nos vários níveis da estrutura, sem a necessidade de replicação de chaves nos nós do nível mais inferior.

Os nós da HFD-Tree possuem o mesmo tamanho, em *bytes*, da página do sistema operacional e são armazenados sequencialmente no SSD, de modo que campos de referências (ponteiros) mantidos em cada nó indicam as páginas dos níveis inferiores, analogamente à representação de uma árvore. Os objetos são mantidos ordenados dentro dos nós. Os nós são modelados de acordo com a Fig. 1, sendo divididos em três seções:

- *Cabeçalho*: contém o tipo de página (nó da árvore ou descritor de arquivo indicando o nó raiz), o identificador (ponteiro) da próxima página no mesmo nível e a quantidade de objetos presentes no nó.

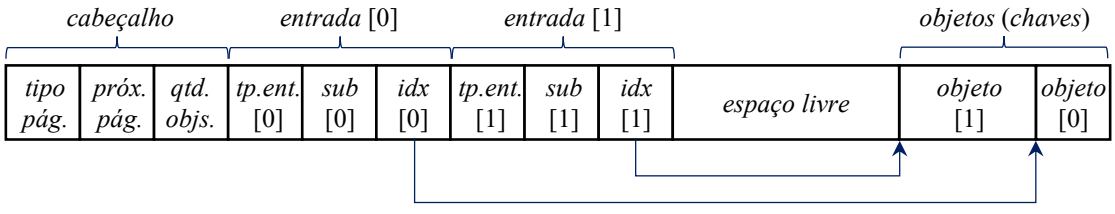


Figura 1. Modelagem de um nó da HFD-Tree

- *Entradas*: triplas de tamanho fixo contendo o tipo da entrada (comum ou referencial), a referência para uma entrada no mesmo nível ou para o nível inferior (se não for entrada comum), e o índice para o primeiro *byte* de cada chave.
- *Objetos*: são as chaves, armazenadas no final da página, de forma a permitir objetos de tamanhos distintos. A primeira entrada referencia a última chave do nó. A segunda entrada referencia a penúltima, e assim por diante. Entre as *entradas* e os *objetos*, têm-se o espaço livre do nó, que é utilizado durante novas inserções.

A busca por uma entrada e_x na HFD-Tree percorre, no máximo, L páginas: uma página para cada nível. A partir do nó raiz, cada entrada e_a armazenada na página é analisada de acordo as seguintes regras:

1. Se e_a for uma entrada comum e $e_a = e_x$, então a entrada foi encontrada.
2. Se e_a for uma entrada referencial e $e_a < e_x$, ela é salva como a última entrada referencial visitada.
3. Se $e_a \geq e_x$, a página apontada pela última entrada referencial salva é carregada e a busca recomeça.

Caso todas as entradas de uma página sejam analisadas e não for encontrada nenhuma entrada maior ou igual a e_x, duas situações podem ocorrer: (i) se houver alguma entrada referencial salva, a página apontada por ela é carregada e o processo de busca recomeça por esta página; (ii) se não houver nenhuma entrada referencial salva, isso significa que a página atual é uma página do último nível da árvore (pois esse é o único nível que não possui nenhuma referência) e, portanto, a entrada não está presente na árvore.

Uma vez que o primeiro nível da HFD-Tree é um nível comum da árvore, e não uma B^+-Tree como na FD-Tree, o algoritmo de inserção é modificado. Para realizar a inserção de um conjunto de chaves e_x na HFD-Tree, inicialmente elas são inseridas em uma página temporária, mantida em memória. Esta página temporária é mesclada com o nível da raiz, de modo que cada entrada é colocada em sua posição correta. Caso o nó raiz exceda sua capacidade, o processo de mesclagem se reinicia de forma recursiva: a raiz é mesclada com o nível seguinte (inferior) até que nenhuma página exceda sua capacidade.

A operação de mesclagem se difere da FD-Tree pelo fato de que não há níveis de uma *head-tree* para serem processados. A mesclagem acontece quando uma página está cheia, sendo necessário repassar seus dados para o nível inferior. Dados dois níveis L_i (cheio) e L_{i+1}, a operação de mesclagem ocorre de acordo com os seguintes passos:

1. Abrir L_i e L_{i+1} para leitura sequencial.
2. Criar uma página temporária (*buffer*) em memória, para a mesclagem.
3. Enquanto um dos níveis (L_i ou L_{i+1}) não estiver vazio:
   - 3.1 Ler um bloco de cada nível.
   - 3.2 Para cada chave nos blocos lidos:
   - 3.3 Se a mesma chave aparece em ambos os blocos, manter apenas a do nível mais inferior (mais atualizada).
   - 3.4 Senão, inserir a menor delas, de forma ordenada, no *buffer* temporário.
   - 3.5 Se o *buffer* estiver cheio, escrevê-lo no nível L_{i+1}.
4. Atualizar estatísticas de ocupação dos níveis.
5. Caso ocorra *overflow* em L_{i+1}, mesclá-lo, recursivamente, com L_{i+2}.

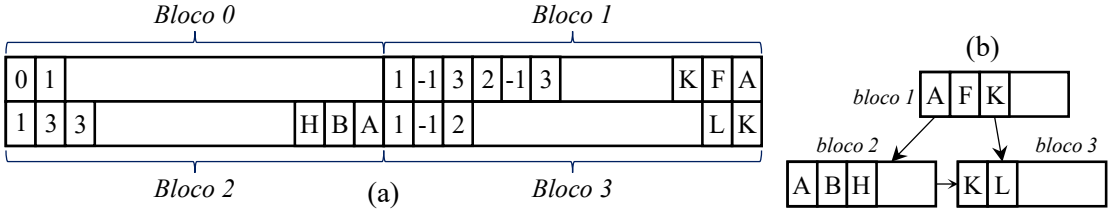


Figura 2. Blocos do arquivo da HFD-Tree e sua representação conceitual

Após a operação de mesclagem ser concluída, os blocos envolvidos no nível L_i não permanecem vazios. Para que haja o correto apontamento para os blocos do nível inferior, as primeiras chaves de cada bloco do nível L_{i+1} são inseridas, na forma de entradas referenciais, no bloco do nível L_i, apontando para os seus respectivos blocos.

Para realizar o processo de inserção, a HFD-Tree implementa uma otimização denotada por *sessão*. A *sessão* é um mecanismo que, fazendo uso de uma tabela de espalhamento (*hash*), mantém todas as páginas que foram acessadas em memória primária, onde as atualizações são realizadas. Ao final de todas as operações de atualização, as páginas que estão mantidas em memória na *sessão* atual são gravadas em lote no SSD.

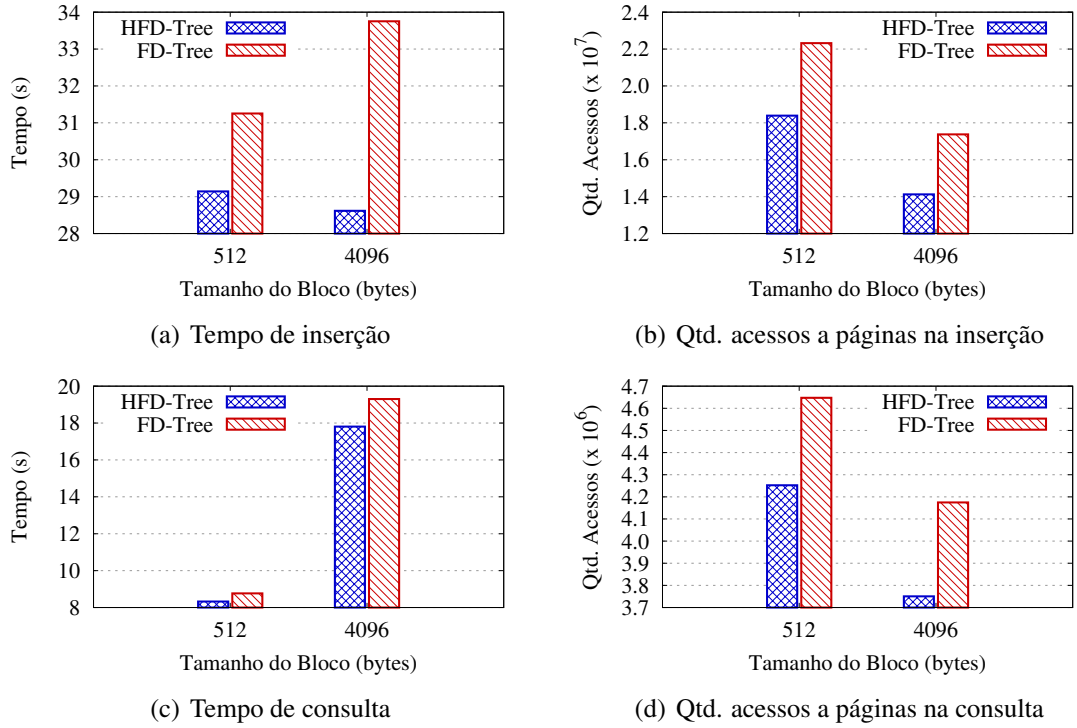
A Fig. 2(a) apresenta o esquema de blocos do arquivo que representa a HFD-Tree persistida no SSD (*tp.ent.* e *idx* das entradas omitidos nos blocos). O Bloco 0 é o descritor, ou cabeçalho (tipo de página = 0), e indica qual bloco é a raiz (no caso, o Bloco 1). O Bloco 1 é um nó da árvore, onde seu tipo de página é 1. Ele não aponta para nenhum bloco no mesmo nível (referência = -1), e contém 3 elementos: A, F e K. A chave 'A' aponta para o Bloco 2, 'F' não aponta para nenhum bloco (-1) e 'K' aponta para o Bloco 3. Este esquema corresponde à representação conceitual da estrutura, ilustrada pela Fig. 2(b).

4. Experimentos

O objetivo dos experimentos foi avaliar o desempenho da estrutura HFD-Tree nas operações de inserção e consulta de dados, onde foram aferidos os valores de tempo e a quantidade de acessos a páginas de disco para escrita/leitura. Os experimentos variaram o tamanho do bloco de disco entre 512 *bytes* (padrão do sistema operacional Linux) e 4096 *bytes* (padrão do sistema operacional Windows) e os valores apresentados correspondem à média de 10 execuções. O conjunto de dados empregado é o *LibreOffice Dictionary* (`pt_BR`)[1], composto por 312368 palavras da língua portuguesa, em ordem aleatória. A HFD-Tree foi comparada com a FD-Tree (FDT) e com a lista sequencial em disco. As implementações foram realizadas em C++, no mesmo *framework*, para permitir comparações, e os testes foram executados em uma máquina com processador Intel Core i5-14500T, 16 GB RAM, SSD NVMe 256 GB - Leitura: 4000 MB/s, Escrita: 2000 MB/s.

A Fig. 3(a) apresenta o tempo total de inserção de dados, onde cada inserção abriu uma nova sessão. A HFD-Tree foi 6% e 15% mais rápida que a FD-Tree para blocos de 512 e 4096 *bytes*, respectivamente. A Fig. 3(b) mostra a quantidade total de acessos a páginas durante a operação de inserção. A HFD-Tree realizou 17% e 18% menos acessos do que a FD-Tree para blocos de 512 e 4096 *bytes*, respectivamente. A otimização de remoção da camada zero faz com que a HFD-Tree não necessite de gerenciar a *head-tree*, reduzindo o tempo e o total de acessos a páginas. A Fig. 3(c) apresenta o tempo total para

[1]LibreOffice: `https://github.com/libreoffice/dictionaries`. Acesso: 23/04/2026



(a) Tempo de inserção (b) Qtd. acessos a páginas na inserção

(c) Tempo de consulta (d) Qtd. acessos a páginas na consulta

Figura 3. Resultados: operações de inserção e consulta

consultar todo o conjunto de dados. A HFD-Tree foi 5% e 8% mais rápida que a FD-Tree para blocos de 512 e 4096 *bytes*, respectivamente. Mesmo que a disposição dos elementos nos nós não seja idêntica para ambas as estruturas, este resultado mostra que estas árvores são otimizadas para a busca. Destaca-se o ganho da HFD-Tree em relação à concorrente. Já em relação à quantidade de acessos, mostrada na Fig. 3(d), a HFD-Tree realizou 22% e 10% menos acessos que a FD-Tree, para blocos de 512 e 4096 *bytes*, respectivamente.

A lista sequencial foi a estrutura que apresentou o menor tempo de inserção, proporcional a $\mathcal{O}(1)$. A lista obteve 0,55 e 0,47 segundos para inserir os dados, para blocos de 512 e 4096 *bytes*, respectivamente. Porém, em relação à consulta ($\mathcal{O}(n)$), o teste foi interrompido após cinco minutos, o que mostra seu baixo desempenho na busca de dados. Estes valores não constam nos gráficos para não distorcer a visualização dos resultados.

5. Conclusão

A evolução dos meios físicos de armazenamento de dados levou à transição dos discos rígidos para as unidades em estado sólido. Dessa forma, surgiram diversas estruturas de índices voltadas para este *hardware*. Neste contexto, este trabalho apresentou a *Headless Flash Disk Tree*, que é um índice dinâmico para dados unidimensionais em SSDs. Esta estrutura segue os princípios de construção da FD-Tree, porém, introduziu quatro otimizações em relação à modelagem e ao funcionamento da estrutura original. Os experimentos, sobre um conjunto de dados real, mostraram um ganho de desempenho temporal de até 16% sobre os concorrentes avaliados. Como passos subsequentes, destacam-se a implementação da operação de remoção de chaves; a análise de desempenho em cenários com concorrência, comuns em Bancos de Dados; a avaliação sobre outros conjuntos de dados (e.g. numéricos), de maior volume, e uma análise estatística dos resultados.

Referências

Boukhobza, J., Olivier, P., Lim, W. S., Chen, L. C., Hsieh, Y. S., Wu, S. T., Ho, C. C., Huang, P. C., and Chang, Y. H. (2025). A survey on flash-memory storage systems: A host-side perspective. *ACM Transactions on Storage*, 21(3):1–59.

Carniel, A. C., Ciferri, R. R., and Aguiar, C. D. (2019). A generic and efficient framework for flash-aware spatial indexing. *Information Systems*, 82:102–120.

Comer, D. (1979). Ubiquitous b-tree. *ACM Computing Surveys*, 11(2):121–137.

Fevgas, A., Akritidis, L., Bozanis, P., and Manolopoulos, Y. (2019). Indexing in flash storage devices: a survey on challenges, current approaches, and future trends. *The VLDB Journal*, 29(1):273–311.

Fevgas, A. and Bozanis, P. (2019). Lb-grid: An ssd efficient grid file. *Data & Knowledge Engineering*, 121:18–41.

Haas, G. and Leis, V. (2023). What modern nvme storage can do, and how to exploit it: High-performance i/o for high-performance storage engines. *Proceedings of the VLDB Endowment*, 16(9):2090–2102.

Hassani, F. S., Fetrat, A. G., Vanestan, S. B., Gholipoor, M., Zoufan, S., Lee, J. A., and Azad, H. S. (2026). A survey on ssd wear leveling techniques. *Computer Science Review*, 60:1–14.

Ho, V. P. and Park, D. J. (2018). Wpcb-tree: A novel flash-aware b-tree index using a write pattern converter. *Symmetry*, 10(1):1–18.

Jiang, Z., Wu, Y., Zhang, Y., Li, C., and Xing, C. (2014). Ab-tree: A write-optimized adaptive index structure on solid state disk. In *11th Web Information System and Application Conference*, pages 188–193.

Kim, B. K., Lee, S. W., and Lee, D. H. (2015). h-hash: A hash index structure for flash-based solid state drives. *Journal of Circuits, Systems and Computers*, 24(09):1–32.

Levandoski, J. J., Lomet, D. B., and Sengupta, S. (2013). The bw-tree: A b-tree for new hardware platforms. In *29th Int. Conference on Data Engineering*, pages 302–313.

Li, Y., He, B., Yang, R. J., Luo, Q., and Yi, K. (2010). Tree indexing on solid state drives. *Proceedings of the VLDB Endowment*, 3(1–2):1195–1206.

Roumelis, G., Velentzas, P., Vassilakopoulos, M., Corral, A., and Manolopoulos, Y. (2018). Spatial batch-queries processing using xBR+-trees in solid-state drives. In *22nd Int. Conference on Database and Expert Systems Applications*, pages 301–317.

Sarwat, M., Mokbel, M. F., Zhou, X., and Nath, S. (2011). Fast: a generic framework for flash-aware spatial trees. In *12th International Conference on Advances in Spatial and Temporal Databases*, page 149–167.

Teng, D., Guo, L., Lee, R., Chen, F., Zhang, Y., Ma, S., and Zhang, X. (2018). A low-cost disk solution enabling lsm-tree to achieve high performance for mixed read/write workloads. *ACM Trans. Storage*, 14(2).

Zhang, X., Bhimani, J., Pei, S., Lee, E., Lee, S., Seong, Y. J., Kim, E. J., Choi, C., Nam, E. H., Choi, J., and Kim, B. S. (2025). Storage abstractions for ssds: The past, present, and future. *ACM Transactions on Storage*, 21(1):1–44.