# External-memory Dictionaries in the Affine and PDAM Models

MICHAEL A. BENDER, Stony Brook University
ALEX CONWAY, VMware Research
MARTÍN FARACH-COLTON, Rutgers University
WILLIAM JANNEN, Williams College
YIZHENG JIAO, The University of North Carolina at Chapel Hill
ROB JOHNSON, VMware Research
ERIC KNORR, Harvard University
SARA MCALLISTER, Carnegie Mellon University
NIRJHAR MUKHERJEE, The University of North Carolina at Chapel Hill
PRASHANT PANDEY, Lawrence Berkeley National Laboratory and University of California Berkeley
DONALD E. PORTER, The University of North Carolina at Chapel Hill
JUN YUAN, Pace University
YANG ZHAN, The University of North Carolina at Chapel Hill

Storage devices have complex performance profiles, including costs to initiate IOs (e.g., seek times in hard drives), parallelism and bank conflicts (in SSDs), costs to transfer data, and firmware-internal operations.

An earlier version of this article appeared in the *Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* [12].

We gratefully acknowledge support from NSF Grants No. CCF 805476, No. CCF 822388, No. CNS 1408695, No. CNS 1755615, No. CCF 2106827, No. CCF 2118832, No. CCF 1439084, No. CCF 1725543, No. CSR 1763680, No. CCF 1716252, No. CCF 1617618, No. IIS 1247726, No. CNS 1938709, No. CNS 1938180, and from Sandia National Laboratories, VMware, and a Netapp Faculty Fellowship.

Authors' addresses: M. A. Bender, Department of Computer Science, Stony Brook University, Stony Brook, NY 11794-2424 USA; email: bender@cs.stonybrook.edu; A. Conway and R. Johnson, VMware research, 3425 Hillview Avenue, Palo Alto CA 94304 USA; emails: {aconway, robj}@vmware.com; M. Farach-Colton, Department of Computer Science, Rutgers University, Piscataway, NJ 08855 USA; email: martin@farach-colton. com; W. Jannen, Department of Computer Science, 47 Lab Campus Drive, Williams College, Williamstown, MA 01276 USA; email: jannen@cs.williams.edu; Y. Jiao and D. E. Porter, Department of Computer Science, University of North Carolina at Chapel Hill, Campus Box 3175, Brooks Computer Science Building, Chapel Hill, NC 27599-3175 USA; emails: yizheng@cs.unc.edu, porter@cs.unc.edu; E. Knorr,School of Engineering and Applied Sciences, Harvard University, Science and Engineering Complex, 150 Western Avenue, Boston, MA 02134 USA; email: eric.r.knorr@gmail.com; S. McAllister, Computer Science Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213-3891 USA; email: sjmcalli@cs.cmu.edu; N. Mukherjee, Computer Science Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213-3891 USA; email: nirjharm@gmail.com; P. Pandey, Lawrence Berkeley National Lab and University of California, Berkeley, 1 Cyclotron Road, Berkeley, CA 94720 USA; email: ppandey@berkeley.edu; J. Yuan, Seidenberg School of Computer Science and Information Systems, Pace University, One Pace Plaza, New York, NY 10038-2613 USA; email: jyuan2@pace.edu; Y. Zhan, Brooks Computer Science Building, 201 S. Columbia Street, University of North Carolina at Chapel Hill, Chapel Hill, NC 27599-3175 USA; email: yzhan@cs.unc.edu.
Updated author affiliation: NIRJHAR MUKHERJEE, Carnegie Mellon University.

2329-4949/2021/09-ART15 $15.00
https://doi.org/10.1145/3470635

The Disk-access Machine (DAM) model simplifies reality by assuming that storage devices transfer data in blocks of size $B$ and that all transfers have unit cost. Despite its simplifications, the DAM model is reasonably accurate. In fact, if $B$ is set to the half-bandwidth point, where the latency and bandwidth of the hardware are equal, then the DAM approximates the IO cost on any hardware to within a factor of 2.

Furthermore, the DAM model explains the popularity of B-trees in the 1970s and the current popularity of B$^{\varepsilon}$-trees and log-structured merge trees. But it fails to explain why some B-trees use small nodes, whereas all B$^{\varepsilon}$-trees use large nodes. In a DAM, all IOs, and hence all nodes, are the same size.

In this article, we show that the affine and PDAM models, which are small refinements of the DAM model, yield a surprisingly large improvement in predictability without sacrificing ease of use. We present benchmarks on a large collection of storage devices showing that the affine and PDAM models give good approximations of the performance characteristics of hard drives and SSDs, respectively.

We show that the affine model explains node-size choices in B-trees and B$^{\varepsilon}$-trees. Furthermore, the models predict that B-trees are highly sensitive to variations in the node size, whereas B$^{\varepsilon}$-trees are much less sensitive. These predictions are born out empirically.

Finally, we show that in both the affine and PDAM models, it pays to organize data structures to exploit varying IO size. In the affine model, B$^{\varepsilon}$-trees can be optimized so that all operations are simultaneously optimal, even up to lower-order terms. In the PDAM model, B$^{\varepsilon}$-trees (or B-trees) can be organized so that both sequential and concurrent workloads are handled efficiently.

We conclude that the DAM model is useful as a first cut when designing or analyzing an algorithm or data structure but the affine and PDAM models enable the algorithm designer to optimize parameter choices and fill in design details.

CCS Concepts: • **Information systems** → **Record and block layout**; **Magnetic disks**; **Flash memory**; • **Theory of computation** → **Data structures design and analysis;**

Additional Key Words and Phrases: External memory, performance models, write optimization

**ACM Reference format:**
Michael A. Bender, Alex Conway, Martín Farach-Colton, William Jannen, Yizheng Jiao, Rob Johnson, Eric Knorr, Sara McAllister, Nirjhar Mukherjee, Prashant Pandey, Donald E. Porter, Jun Yuan, and Yang Zhan. 2021. External-memory Dictionaries in the Affine and PDAM Models. *ACM Trans. Parallel Comput.* 8, 3, Article 15 (September 2021), 20 pages.
https://doi.org/10.1145/3470635

## 1 INTRODUCTION

Storage devices have complex performance profiles, including costs to initiate IO (e.g., seek times in hard drives), parallelism and bank conflicts (in SSDs), costs to transfer data, and firmware-internal operations.

The **Disk-access Machine (DAM)** model [2] simplifies reality by assuming that storage devices transfer data in blocks of size $B$ and that all transfers have unit cost. Despite its simplifications, the DAM model has been a success [4, 71], in part, because it is easy to use.

The DAM model is also reasonably accurate. If $B$ is set to the hardware's ***half-bandwidth size***—that is, the IO size where a request's average initiation time and data-transfer time are equal—then the DAM model predicts the IO cost of any algorithm to within a factor of roughly 2 on that hardware. In other words, if an algorithm replaces all of its IOs with IOs of the half-bandwidth size, i.e., by padding smaller IOs and breaking larger IOs into half-bandwidth-sized requests, then that algorithm's IO cost increases by a factor of roughly 2.

The DAM model explains some choices that software architects have made. For example, the DAM model gives an analytical explanation for why B-trees [8, 33] took over in the 1970s and why B$^{\varepsilon}$-trees [17, 26], log-structured merge trees [15, 53], and external-memory skip lists [10, 19, 61] are taking over now.

But the DAM model has its limits. For example, the DAM model does not explain why B-trees in many databases and file systems use nodes of size 16 KiB [1, 50, 51, 55, 56], which is well below the half-bandwidth size on most storage devices, whereas B-trees optimized for range queries use larger node sizes, typically up to around 1 MB [57, 59]. Nor does it explain why TokuDB's [69] B$^{\varepsilon}$-tree uses 4MiB nodes and LevelDB's [42] LSM-tree uses 2MiB SSTables for all workloads. In a DAM, all IOs, and hence all nodes, are the same size.

How can an optimization parameter that can vary by over three orders of magnitude have escaped algorithmic recognition? The answer is that the DAM model is too blunt an instrument to capture these design issues.

In this article, we show that the ***affine*** [3, 63] and ***PDAM*** [2] models, which are small refinements of the DAM model, yield a surprisingly large improvement in predictivity without sacrificing ease of use.

The ***affine*** and ***PDAM*** models explicitly account for seeks (in spinning disks) and parallelism (in solid-state storage devices). In the affine model, the cost of an IO of $k$ words is $1 + \alpha k$, where $\alpha \ll 1$ is a hardware parameter.[1] In the PDAM model, an algorithm can perform up to $P$ IOs of size $B$ in parallel.

**Results.** We show that the affine and PDAM models improve upon the DAM model in three ways.

**The affine and PDAM models more accurately estimate IO costs.** In Section 4, we present microbenchmarks on a large collection of storage devices showing that the affine and PDAM models are good approximations of the performance characteristics of hard drives and SSDs, respectively. We show that, for example, the PDAM is able to correctly predict the run-time of a parallel random-read benchmark on SSDs to within an error of never more than 14% across a broad range of devices and numbers of threads. The DAM model, on the other hand, overestimates the completion time for large numbers of threads by roughly $P$, the parallelism of the device, which ranges from 2.9 to 5.5. On hard drives, the affine model predicts the time for IOs of varying sizes to within a 25% error, whereas, as described above, the DAM model is off by up to a factor of 2.

Researchers have long understood the underlying hardware effects motivating the affine and PDAM models. Nonetheless, it was a pleasant surprise to see how accurate these models turn out to be, even though they are simple tweaks of the DAM model.

**The affine and PDAM models explain software design choices.** In Sections 5 and 6, we reanalyze the B-tree and the B$^{\varepsilon}$-tree in the affine and PDAM models. The affine model explains why B-trees typically use nodes that are much smaller than the half-bandwidth size, whereas B$^{\varepsilon}$-trees have nodes that are larger than the half-bandwidth size. Furthermore, the models predict that the B-tree is highly sensitive to variations in the node size whereas B$^{\varepsilon}$-trees are much less sensitive. These predictions are borne out empirically.

**The affine and PDAM models enable better data-structure designs.** In a B$^{\varepsilon}$-tree, small nodes optimize point queries and large nodes optimize range queries and insertions. In Section 6, we show that in the affine model, nodes can be organized with internal structure (such that nodes have subnodes) so that all operations are simultaneously optimal, up to lower order terms. Since the DAM model loses a factor of 2, it is blind to such fine-grained optimizations.

The PDAM model allows us to organize nodes in a search tree so that the tree achieves optimal throughput when the number of concurrent read threads is both large and small. A small number of read threads favors large nodes, and a large number favors small nodes. In Section 8, we show how to organize nodes so that part or all of them can be read in parallel, which allows the data structure to handle both workloads obliviously and optimally.

[1] In reality, storage systems have a minimum write size, but we ignore this issue, because it rarely makes a difference in the analysis of data structures and it makes the model cleaner.

**Discussion.** Taking a step back, we believe that the affine and PDAM models are important complements to the DAM model. The DAM is useful as a first cut when designing or analyzing an algorithm or data structure but the affine and PDAM models enable the algorithm designer to optimize parameter choices and fill in design details.

## 2 THE PDAM AND AFFINE MODELS

In this section, we present the affine model (most predictive of hard disks) and the PDAM model (most predictive of SSDs). The DAM model assumes all IOs have the same size and cost, which is a reasonable approximation for IOs that are large enough. The affine and PDAM models capture what happens for small and medium IO sizes.

### 2.1 Disk Access Machine (DAM) Model

The DAM model [2] assumes a two-level cache hierarchy with a cache of $M$ words, where the cache is subdivided into $M/B$ blocks; a slower storage device transfers data to and from cache in these $B$-sized blocks. The model applies to any two adjacent levels of a cache hierarchy, such as RAM versus disk or L3 versus RAM. Performance in the DAM model is measured by counting the number of block transfers performed during an algorithm's or data structure's execution.

Note that $B$ is distinct from the block size of the underlying hardware. $B$ is a tunable parameter that determines the amount of contiguous data transferred per IO: a bigger $B$ means each IO transfers more data but takes more time.

The DAM model counts IOs but does not assign a cost to each IO. The DAM model's simplicity is a strength in terms of usability but a weakness in terms of predictability. On HDDs, it does not model the faster speeds of sequential IO versus random IO. On SSDs, it does not model internal device parallelism or the incremental cost of larger IOs.

These inaccuracies limit the effectiveness of the DAM model for optimizing data structures. As we will show, there are asymptotic consequences for these performance approximations.

For example, in the DAM model, the optimal node size for an index such as a B-tree, B$^{\varepsilon}$-tree, or buffered repository tree is $B$ [17, 27, 33]. There is no advantage to growing smaller than $B$, since $B$ is the smallest granularity at which data is transferred in the data structure. But using nodes larger than $B$ also does not help.

Could the DAM be right? Maybe the right solution is to pick the best $B$ as an extra-model optimization, and from then on use $B$ in all data-structure design. Alas no. The best IO size is workload and data-structure dependent [17, 20].

### 2.2 SSDs and the PDAM Model

The PDAM model improves SSD performance analysis over the DAM model by accounting for the IO parallelism. In its original presentation [2], the external-memory model included one more parameter, $P$, to represent the number of blocks that can be transferred concurrently. This parameter was originally proposed to model the parallelism available in RAID arrays (arrays of cooperating disks that masquerade as a single logical unit). Although there exist parallel and multicore IO models [5, 6, 24, 32, 72, 73], almost all theoretical work on external memory is parameterized by $M$, $N$, $B$, and not $P$, which means that implicitly $P = 1$.

We argue for reviving $P$ to model the parallelism of SSDs, including fast NVMe SSD devices. (NVMe, which stands for non-volatile memory express, is an interface specification for connecting non-volatile memory devices over the PCI express bus; NVMe SSDs typically have faster, higher-end components.)

Internally, SSDs store data on units of NAND flash, which the device controller may access in parallel. To manage the complexities of the internal hardware, SSD controllers implement a **flash**

**translation layer (FTL)** that (among its many responsibilities) receives IO requests from the OS and transparently maps those requests to physical locations on the SSD's NAND flash chips. The FTL performs IOs at page granularity, typically 512 B–16 KB. Multiple NAND flash pages form a single write-erase block, and multiple write-erase blocks are grouped together into banks, which can be independently accessed in parallel. The FTL typically manages multiple flash chips, granting even more parallelism [38, 43]. This internal parallelism is why applications must maintain deep IO request queues to get full bandwidth out of an SSD [30, 41].

*Definition 1 (PDAM Model).* In each time step, the device can serve up to $P$ IOs, each of size $B$. If the application does not present $P$ IOs to the device in a time step, then the unused slots are wasted. Within a time step, the device can serve any combination of reads and writes. Performance is measured in terms of time steps, not the total number of IOs.

Thus, in the PDAM model a sequential scan of $N$ items, which uses $O(N/B)$ IOs, can be performed in $O(N/(PB))$ time steps.

For the purposes of this article, IOs are **concurrent-read-exclusive-write (CREW)** [74] (i.e., if there is a write to location $x$ in a time step, then there are no other concurrent reads or writes to $x$.)

### 2.3 Hard Disks and the Affine Model

When a hard drive performs an IO, the read/write head first seeks, which has a ***setup time*** of $s$ seconds, and once the disk head is in place, it reads data locally off the platter at bandwidth of $t$ seconds/byte, which corresponds to a ***transfer rate***.

Parameters $s$ and $t$ are approximate, and vary based on disk geometry; the setup time can vary by an order of magnitude. For example, a track-to-track seek may be ~1 ms, while a full platter seek is ~10 ms. Nonetheless, it is remarkably predictive to view these as fixed [63].

*Definition 2 (Affine Model).* IOs can have any size. An IO of size $x$ costs $1 + \alpha x$, where the 1 represents the normalized setup cost and $\alpha \leq 1$ is the normalized transfer cost.

Thus, after normalizing, $\alpha = t/s$ for a hard disk.

The following lemma gives reductions between the affine and DAM models, even when IOs in the DAM model need to be block-aligned.

Lemma 1. *An affine algorithm with affine cost $C$ can be transformed into a DAM algorithm with DAM cost at most $4C$, where blocks have size $B = 1/\alpha$. If the affine algorithm has a RAM of size $M$, then the DAM algorithm has a RAM of size $M + 2/\alpha$.*

*A DAM algorithm with DAM cost $C$ and blocks of size $B = 1/\alpha$ can be transformed into an affine algorithm with affine cost $2C$. If the DAM algorithm has a RAM of size $M$, then the affine algorithm also has a RAM of size $M$.*

Proof. We first give the affine-to-DAM transformation. The idea is to round up any read or write IO of size $x$ into blocks of size $1/\alpha$. Whatever data is stored in memory in the affine algorithm is also stored in the DAM algorithm. Because IOs are rounded up to aligned blocks, an IO of size $x$ in the affine algorithm may be rounded up to a block-aligned chunk of memory of size $x + 2/\alpha$ in the DAM model. For an IO that is a read, this means that data of size nearly $2/\alpha$ may be read into memory and then immediately discarded. For an IO that is a write, this means that data of size nearly $2/\alpha$ may need to be read into memory before the aligned blocks can be written atomically.

We now consider IOs of size $x$. We separately address the cases of (1) $x \leq 1/\alpha$ and (2) $x > 1/\alpha$.

(1) When $x \leq 1/\alpha$, an IO costs at least 1 in the affine model, and requires at most two blocks to be transferred. However, if these two blocks are writes, then up to two blocks may first also need

to be read so that blocks can be written back atomically. Thus, an affine cost of between 1 and 2 may trigger a DAM cost of at most 4.

(2) When $x > 1/\alpha$, an IO of size $x$ transfers $1 + \lceil \alpha x \rceil \leq \alpha x + 2$ blocks in the DAM model. For IOs that are writes, there are an additional two blocks that may need to be read in. Thus, when the original affine cost is $1 + \alpha x$, the DAM cost is at most $\alpha x + 4$.

The DAM-to-affine transformation is more direct. Every IO in the DAM algorithm has size $1/\alpha$. The same IO of size $1/\alpha$ has an affine cost of 2. □

Thus, if losing a constant factor on all operations is satisfactory, then the DAM is good enough.

What may be surprising is how many asymptotic design effects show up when optimizing to avoid losing this last constant factor. A factor of 2 is a lot for an external-memory dictionary. For example, even smaller factors were pivotal for a subset of authors of this article when we were building and marketing TokuDB [69]. Additionally, losing a factor of 2 on large sequential write performance was a serious setback on making B*ε*trFS a general-purpose file system [34, 47, 76–78].

## 3 BACKGROUND ON B-TREES AND B$^\varepsilon$-TREES

A ***dictionary data structure*** maintains a set of key-value pairs and supports inserts, deletes, point queries, and range queries. Here we review some common external-memory dictionaries.

**B-trees.** The classic dictionary for external storage is the B-tree [8, 33]. A B-tree is a balanced search tree with fat nodes of size $B$, so that a node can have $\Theta(B)$ pivot keys and $\Theta(B)$ children. All leaves have the same depth, and key-value pairs are stored in the leaves. The height of a B-tree is $\Theta(\log_{B+1} N)$, where $N$ is the number of elements in the tree. There are many algorithmic choices for B-tree rebalancing. One simple mechanism is the following. Split any node that has greater than $B$ elements and merge any node that has fewer than $B/4$ elements with a sibling.

The following theorem gives the performance of a B-tree with a warm cache, that is, when the top part of the tree is stored not only on disk but also is also cached in RAM:

Lemma 2 (Folklore). *In a B-tree with size-$B$ nodes and $N$ elements, point queries, inserts, and deletes take $O(\log_{B+1}(N/M))$ IOs. A range query scanning $\ell$ elements takes $O(\lceil \ell/B \rceil)$ IOs plus the point-query cost. The amortized IOs spent modifying the tree per insert/delete is $O(1/B)$.*

Proof. The B-tree can cache up to $M/B$ of its nodes in RAM. Say that the top $\log_{B+1}(M/B) \pm o(1)$ levels of the tree are cached. Then accessing any node in this top part of the tree has zero IO cost. The result is that a point query, which follows a path from the root of the tree to a leaf, makes $\log_{B+1}(N) - \log_{B+1}(M) \pm o(1)$ IOs.

A range query is just a point query plus a scan of the leaves. Since $\ell$ elements are stored in $O(\lceil \ell/B \rceil)$ leaves, the IO cost for the leaf scan is an additional $O(\lceil \ell/B \rceil)$ IOs.

An insert or delete is a point query plus the cost to merge and split tree nodes. At most $O(1)$ nodes are merged/split per level, and so the bound follows.

In fact, the amortized rebalance cost is a low-order term. As we explain, in any sequence of $\Theta(N)$ inserts/deletes there are $O(N/B)$ node splits/merges. In particular, after a leaf triggers a split or merge, there are $\Omega(B)$ inserts/deletes before it can trigger another split/merge. A parent of a leaf therefore needs $\Omega(B^2)$ inserts/deletes into descendants between when it can trigger a modification, and in general a node at height $h$ needs $\Omega(B^{h+1})$ inserts or deletes. Thus, the total number of IOs is $\Theta(N/B)$ for an amortized modification cost of $\Theta(1/B)$ IOs. □

The systems community often evaluates data structures in terms of their ***write amplification***, which we define below [62].

*Definition 3.* The ***write amplification*** is the amortized amount of data written to disk per update divided by the amount of data actually modified by the update.

Write amplification is the traditional way of distinguishing between read IOs and write IOs. Distinguishing between reads and writes makes sense, because with some storage technologies (e.g., NVMe SSDs) writes are more expensive than reads, and this has algorithmic consequences [9, 22, 23, 46]. Moreover, even when reads and writes have about the same cost, other aspects of the system can make writes more expensive. For example, modifications to the data structure may be logged, and so write IOs in the B-tree may also trigger write IOs from logging and checkpointing.

In the DAM model, the write amplification of a dictionary is just $B$ times the amortized number of write IOs per insertion.

Lemma 3. *The worst-case write-amplification of a B-tree is* $\Theta(B)$.

Proof. We first give a B-tree workload that achieves a write amplification of $\Omega(B)$. Let the number of elements in the B-tree be $N = \Omega(BM)$. Suppose that random elements are inserted into the tree. Divide the computation into phases, where, each phase consists of $M$ random inserts. Thus, in expectation, $\Theta(M)$ leaves are modified. But only $\Theta(M/B)$ leaves can be cached in memory at any time. Thus, each phase takes an expected $\Theta(M)$ IOs, writing $\Theta(M)$ leaves to disk. Since each leaf has size $\Theta(B)$, each phase transfers an expected $\Theta(MB)$ data.

We now show that amortized write amplification of a B-tree is $O(B)$. Divide any workload into phases of $\Theta(N)$ inserts/deletes. By Lemma 2, in every $\Theta(N)$ inserts/deletes, there may be $O(N)$ IOs, but there are only $O(N/B)$ node splits and merges. Thus, the amortized IO cost for modifications to the tree per insert/delete is $O(1)$ for a write amplification of $O(B)$. □

**B$^\varepsilon$-trees.** The B$^\varepsilon$-tree [13, 17, 26, 27, 48] is a write-optimized generalization of the B-tree. (A ***write-optimized dictionary*** (***WOD***) is a searchable data structure that has (1) substantially better insertion performance than a B-tree and (2) query performance at or near that of a B-tree.)

The B$^\varepsilon$-tree is used in some write-optimized databases and file systems [34, 39, 47, 47, 49, 58, 67, 68, 76–78]. A more detailed description of the B$^\varepsilon$-tree is available in the prior literature [17].

As with a B-tree, the B$^\varepsilon$-tree is a balanced search tree with fat nodes of size $B$. A B$^\varepsilon$-tree leaf looks like a B-tree leaf, storing key-value pairs in key order. A B$^\varepsilon$-tree internal node has pivot keys and child pointers, like a B-tree, but it also has space for a ***buffer***. The buffer is part of the node and is written to disk with the rest of the node when the node is evicted from memory. Modifications to the dictionary are encoded as ***messages***, such as an insertion or a so-called tombstone message for deletion. These messages are stored in the buffers in internal nodes, and eventually applied to the key-value pairs in the leaves. A query must search the entire root-to-leaf path, and logically apply all relevant messages in all of the buffers.

Buffers are maintained using the ***flush*** operation. Whenever a node $u$'s buffer is full ("overflows"), then the tree selects (at least one) child $v$, and moves all relevant messages from $u$ to $v$. Typically $v$ is chosen to be the child with the most pending messages. Flushes may recurse, i.e., when a parent flushes, it may cause children and deeper decedents to overflow.

The B$^\varepsilon$-tree has a tuning parameter $\varepsilon$ ($0 \leq \varepsilon \leq 1$) that controls the fanout $F \approx B^\varepsilon + 1$. Setting $\varepsilon = 1$ optimizes for point queries and the B$^\varepsilon$-tree reduces to a B-tree. Setting $\varepsilon = 0$ optimizes for insertions/deletions, and the B$^\varepsilon$-tree reduces to a buffered repository tree [27]. Setting $\varepsilon$ to a constant in between leads to point-query performance that is within a constant factor of a B-tree, but insertions/deletions that are asymptotically faster. In practice, $F$ is chosen to be in the range $[10, 20]$; for example, in TokuDB, the target value of $F$ is 16.

Theorem 4 ([17, 26]). *In a B$^\varepsilon$-tree with N elements, size-B nodes, and fanout* $1 + B^\varepsilon$, *for* $\varepsilon \in [0, 1]$,

*(1) insertions and deletions take* $O(\frac{1}{B^{1-\varepsilon}} \log_{B^\varepsilon+1}(N/M))$ *amortized IOs,*

*(2) point queries take* $O(\log_{B^\varepsilon+1}(N/M))$ *IOs, and*

*(3) a range query returning $\ell$ elements takes $O(\lceil \ell/B \rceil)$ IOs plus the cost of a point query,*
*(4) The write amplification is $O(B^{\varepsilon} \log_{B^{\varepsilon}+1}(N/M))$.*

## 4 MICROBENCHMARKS TO VALIDATE THE AFFINE AND PDAM MODELS

We now experimentally validate the accuracy of the affine model for hard disks and the PDAM model for SSDs. We show that the models are remarkably accurate, even though they do not explicitly model most hardware effects.

One of the messages of this section is that, even though the affine and PDAM models are only tweaks to the DAM model, they have much more predictive power. We can even use them to make predictions and reason about constants. As we will see in the next section, optimizing the constants for various operations will cause some design parameters to change asymptotically.

Unless noted otherwise, all experiments in this article were collected on a Dell PowerEdge T130 with a 3.0 GHz Intel E3-1220 v6 processor, 32 GiB of DDR4 RAM, two 500 GiB TOSHIBA DT01ACA050 HDDs, and one 250 GiB Samsung 860 EVO SSD. Each experimental SSD was first preconditioned according to the SNIA **Solid State Storage (SSS) Performance Test Specification (PTS)**, version 2.0.1 [66].

### 4.1 Validating the PDAM Model

The PDAM model ignores some issues of real SSDs, such as bank conflicts, i.e., when independent IOs address the same bank and those IOs must be serialized, which can limit the parallelism available for some sets of IO requests. Despite its simplicity, we verify below that the PDAM model is consistent with real SSD and NVMe SSD performance.

To test the PDAM model, we ran many rounds of IO read experiments with different numbers of threads. In each round of the experiment, we spawned $p = \{1, 2, 3, 4, 5, 6, 7, 8, 16, 32, 64\}$ OS threads that each read 10 GiB of data. We selected 163,840 random block-aligned offsets and read 64KiB starting from each. Thus, there were up to $p$ outstanding IO requests at any time, and the total data read was $p \times 10$ GiB per round.

The PDAM model predicts that the time to complete the experiment should be the same for all $p \leq P$ and should increase linearly in $p$ for all $p > P$. Equivalently, the PDAM model predicts that the observed throughput should increase linearly with each additional thread for all $p \leq P$, and then remain flat for $p > P$. Figure 1 shows the observed throughput in MiB/second during each round of IO read experiments. Each SSD exhibits two distinct performance regimes: the throughput grows linearly until around $p = 4$–$6$, depending on the device, and it flattens sharply thereafter.

We used segmented linear regression to estimate the device parallelism ($P$) and the throughput ($\beta$) of each thread up until the device parallelism is saturated. Segmented linear regression is appropriate for fitting data that is known to follow different linear functions in different ranges. Segmented linear regression outputs the boundaries between the different regions and the parameters of the line of best fit within each region. Table 1 shows the experimentally derived parallelism, $P$, and the device saturation, $\propto P\beta$, for a variety of flash devices.

To verify the goodness of fit, we report the $R^2$ value. An $R^2$ value of 1 means that the regression coefficients perfectly predicted the observed data. Our $R^2$ values are all within 4% of 1.

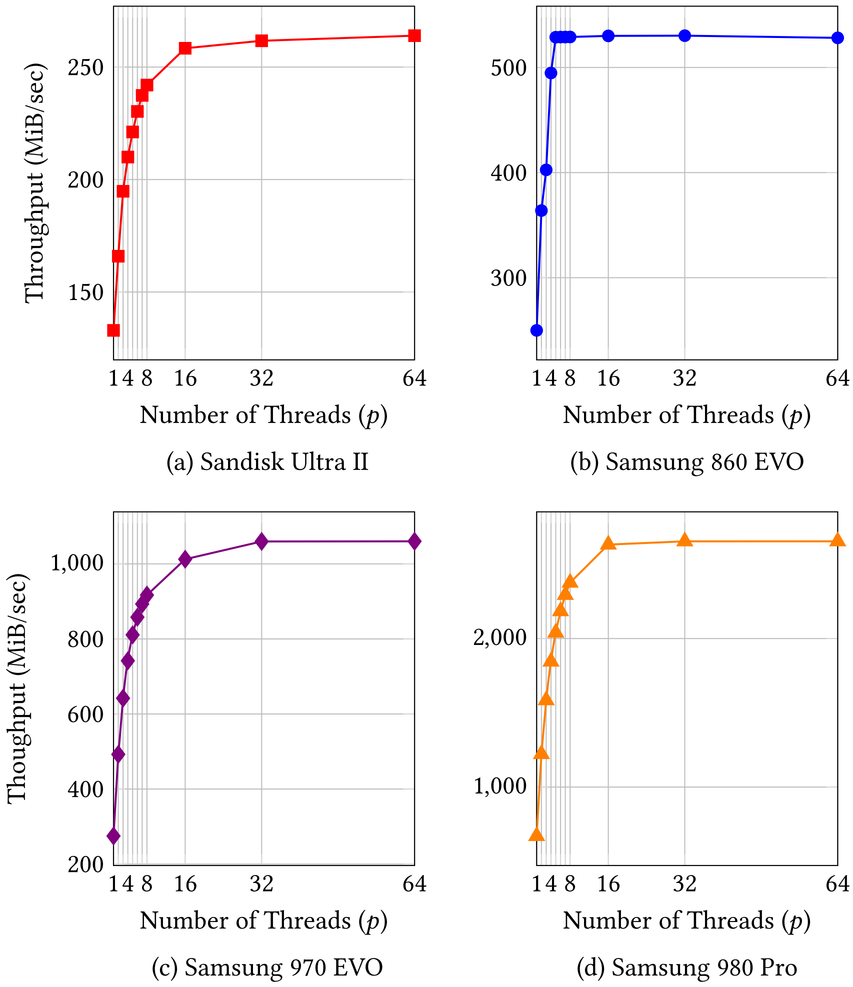
### 4.2 Validating the Affine Model

In this section, we empirically derive $\alpha = t/s$ for a series of commodity hard disks, and we confirm that the affine model is highly predictive of hard disk performance.

Table 1. Experimentally Derived PDAM Values for Real Hardware

| Device | $P$ | $\propto P\beta$ | $R^2$ |
|---|---|---|---|
| Sandisk Ultra II | 5.5 | 240 | 0.964 |
| Samsung 860 EVO | 4.5 | 530 | 0.989 |
| Samsung 970 EVO | 5.4 | 900 | 0.971 |
| Samsung 980 Pro | 5.6 | 2,300 | 0.963 |

We used segmented linear regression to calculate $P$. After $P$ threads, throughput remains nearly constant at $\propto P\beta$.



(a) Sandisk Ultra II

(b) Samsung 860 EVO

(c) Samsung 970 EVO

(d) Samsung 980 Pro

Fig. 1. Random-read throughput as a function of the number of threads ($p$).

For our experiments, we chose an IO size, $I$, and issued 64 $I$-sized reads to block-aligned offsets chosen randomly within the device's full LBA range. We repeated this experiment for a variety of IO sizes, with $I$ ranging from 1 disk block up to 16 MiB. Table 2 shows the experimentally derived values for each HDD. To verify the goodness of fit, we report the $R^2$ value. $R^2$ values are all within 0.1% of 1, and we conclude that the affine model is an excellent fit for hard disks.

Table 2. Experimentally Derived $\alpha$ Values for Commodity HDDs

| Disk | Year | $s$ (s) | $t$ (s/4K) | $\alpha$ | $R^2$ |
|---|---|---|---|---|---|
| 250 GB Seagate | 2006 | 0.015 | 0.000033 | 0.0022 | 0.9997 |
| 1 TB Hitachi | 2009 | 0.013 | 0.000041 | 0.0031 | 0.9999 |
| 1 TB WD Black | 2011 | 0.012 | 0.000035 | 0.0029 | 0.9997 |
| 2 TB Seagate | 2012 | 0.018 | 0.000021 | 0.0012 | 0.9994 |
| 6 TB WD Red | 2018 | 0.016 | 0.000026 | 0.0017 | 0.9972 |

We issued 64 random block-aligned reads with IO sizes ranging from 1 disk block to 16 MiB. We conducted linear regression to get the setup cost $s$ and bandwidth cost $t$. We calculated $\alpha$ by $t/s$.

Table 3. A Sensitivity Analysis of Node Sizes for B$^\varepsilon$-trees and B-trees

| | Insertion/Deletion | Query |
|---|---|---|
| B-trees | $\Theta\left(\frac{1+\alpha B}{\log B}\log\frac{N}{M}\right)$ | $\Theta\left(\frac{1+\alpha B}{\log B}\log\frac{N}{M}\right)$ |
| B$^\varepsilon$-tree ($F = \sqrt{B}$) | $\Theta\left(\frac{1+\alpha B}{\sqrt{B}\log B}\log\frac{N}{M}\right)$ | $\Theta\left(\frac{1+\alpha\sqrt{B}}{\log B}\log\frac{N}{M}\right)$ |
| B$^\varepsilon$-tree | $\Theta\left(\frac{F(1+\alpha B)}{B\log F}\log\frac{N}{M}\right)$ | $\Theta\left(\frac{F+\alpha F^2+\alpha B}{F\log F}\log\frac{N}{M}\right)$ |

The cost of B-tree update operations grows nearly linearly as a function of $B$—specifically $\frac{1+\alpha B}{\log B}$. B$^\varepsilon$-trees should optimize $\frac{F(1+\alpha B)}{B\log F}$ for inserts, deletes, and updates and $\frac{2F+\alpha F^2+\alpha B}{F\log F}$ for queries. The cost for inserts and queries increases more slowly in B$^\varepsilon$-trees than in B-trees as the node size increases.

## 5 B-TREE NODES IN THE AFFINE MODEL

In this section, we use the affine model to analyze the effect of changing the size of B-tree nodes. In the next section, we will perform the analysis for B$^\varepsilon$-trees.

### 5.1 Large Nodes Optimize Lookups, Updates, and Range Queries

The following lemma follows immediately from the definition of a B-tree and the definition of the affine model.

Lemma 5. *The affine IO cost of a lookup, insert, or delete in a B-tree with sized-B nodes is* $(1 + \alpha B)\log_{B+1}(N/M)(1+o(1))$. *The affine IO cost of a range query returning* $\ell$ *items is* $O(1+\ell/B)(1+\alpha B)$ *plus the cost of the query.*

Proof. A B-tree node has size $B$ and the cost to perform an IO of size $B$ is $1 + \alpha AsB$. The height of the B-tree is $\log_{B+1}(N)(1 + o(1))$, since the target fanout is $B$, and the fanout can vary by at most a constant factor. The top $\Theta(\log_{B+1} M)$ levels can be cached so that accesses to nodes within the levels are free. Thus, the search cost follows from the structure of the tree.

As described in Lemma 2, during the course of $N$ inserts/deletes, there are $O(N/B)$ node splits or merges. Thus, the tree-rebalance cost during inserts/deletes is a lower-order term, and so the insert/delete cost is the same as the search cost.

A range query returning $\ell$ items fits in $\Theta(\lceil\ell/B\rceil)$ leaves and each block access costs $1 + \alpha B$. □

Corollary 6. *In the affine IO model, search, insert/delete, and range queries are asymptotically optimized when* $B = \Theta(1/\alpha)$.

Proof. Setting the node size to $B = 1/\alpha$ matches the half-bandwidth size. □

Corollary 6 seems definitive, because it says that there is a parameter setting such that both point queries and range queries run within a constant factor of optimal. It is not.

It may be surprising that the half-bandwidth size is not what people usually use to optimize a B-tree. In particular, B-trees in many databases and file systems use nodes of size 16KiB [1, 50, 51, 55, 56], which is too small to amortize the setup cost. As a result, range queries run slowly, underutilizing disk bandwidth [34, 35, 65]. In contrast, B-trees in databases that are more focused on analytical workloads use larger block sizes, typically up to around 1 MB [57, 59], to optimize for range queries.

**B-tree nodes are often small.** The rest of this section gives analytical explanations for why B-tree nodes are generally smaller than the half-bandwidth size.

Our first explanation is simply that even small constant factors can matter.

The following corollary shows that in the affine model, when we optimize for point queries, inserts, and deletes, then the B-tree node size is smaller than indicated in Corollary 6—that is, $B = o(1/\alpha)$. For these smaller node sizes, range queries run asymptotically suboptimally. In contrast, if range queries must run at near disk bandwidth, then point queries, inserts, and deletes are necessarily suboptimal in the worst case.

Corollary 7. *Point queries, inserts, and deletes are optimized when the node size is* $\Theta(1/(\alpha \ln(1/\alpha)))$. *For this node size, range queries on the B-tree are asymptotically suboptimal.*

Proof. From Lemma 5, finding the optimal node size for point queries means finding the minimum of the function

$$f(x) = \frac{1 + \alpha x}{\ln(x + 1)}.$$

Taking the derivative, we obtain

$$f'(x) = \frac{\alpha}{\ln(x + 1)} - \frac{1}{\ln^2(x + 1)} \frac{1 + \alpha x}{1 + x}.$$

Setting $f'(x) = 0$, the previous equation simplifies to

$$1 + \alpha x = \alpha \ln(x + 1)(1 + x).$$

Given that $\alpha < x < 1$, we obtain $x \ln x = \Theta(1/\alpha)$, which means that $x = \Theta(1/(\alpha \ln(1/\alpha)))$. Second derivatives confirm that we have a minimum.

We now adapt the information-theoretic lower bound on the cost of searching to the affine model. Consider a successor/predecessor query for an element $x$ that is not stored in the B-tree. (The argument is essentially the same when $x$ is present.) There are $N + 1$ possible answers to the query. This means that it takes at least $\lg(N + 1)$ bits to specify the answer to a query.

An IO of size $B$ contains at most $B$ pivots, and therefore there are at most $B + 1$ possible outcomes from comparing the target element to the pivots. Therefore, we can learn at most $\lg(B + 1)$ bits from doing an IO of size $B$, so that the resulting search needs $\frac{\lg(N+1)}{\lg(B+1)} = \lg_{B+1}(N + 1)$ IOs.

Since the cost of such an IO is $1 + \alpha B$, the IO cost per bit learned is $\frac{1+\alpha B}{\lg(B+1)}$. Minimizing this cost means finding the maximum of the function $\frac{\lg(X+1)}{1+\alpha B}$, as before. □

As Corollary 7 indicates, the optimal node size $x$ is not large enough to amortize the setup cost. This means that as B-trees age, their nodes get spread out across the disk, and range-query performance degrades. This is borne out in practice [34, 35, 37, 65].

A second reason that B-trees often use small nodes has to do with write amplification, which is large in a B-tree; see Lemma 3. Since the B-tree write amplification is linear in the node size, there is downward pressure towards small B-tree nodes. A third reason is that big nodes pollute the cache, making the cache less effective.

As mentioned above, database practice has lead to a dichotomy in B-tree uses: Online Transaction Processing (OLTP) databases favor point queries and insertions; Online Analytical Processing (OLAP) databases favor range quieres. As predicted by the analysis in this section, OLTP databases use small leaves and OLAP databases use large leaves.

We believe that the distinction between OLAP and OLTP databases is not driven by user need but by the inability of B-trees to keep up with high insertion rates [36], despite optimizations [7, 11, 16, 18, 21, 28, 29, 44, 45, 52, 54, 64, 70, 75].

We next turn to the B$^\varepsilon$-tree, which can index data at rates that are orders of magnitude faster than the B-tree.

## 6 B$^\varepsilon$-TREE NODES IN THE AFFINE MODEL

In this section, we use the affine model to analyze B$^\varepsilon$-trees. We first perform a naïve analysis of the B$^\varepsilon$-tree [17, 26] in the affine model, assuming that IOs only read entire nodes—effectively the natural generalization of the DAM model analysis.

The analysis reveals that B$^\varepsilon$-trees are more robust to node-size choices than B-trees. In the affine model, once the node size $B$ becomes sufficiently large, transfer costs grow linearly in $B$. For a B$^\varepsilon$-tree with $\varepsilon = 1/2$, the transfer costs (and write amplification) of inserts grow proportionally to $\sqrt{B}$. This means B$^\varepsilon$-trees can use much larger nodes than B-trees, and that they are much less sensitive to the choice of node size.

However, the transfer costs of queries in a B$^\varepsilon$-tree still grow linearly in $B$, which means that, in the affine model and with a standard B$^\varepsilon$-tree, designers face a trade-off between optimizing for insertions versus optimizing for queries.

We describe three optimizations to the B$^\varepsilon$-tree that eliminate this trade-off.

This result is particularly exciting because, in the DAM model, there is a tight tradeoff between reads and writes [26]. In the DAM model, a B$^\varepsilon$-tree (for $0 < \varepsilon < 1$) performs inserts a factor of $\varepsilon B^{1-\varepsilon}$ faster than a B-tree, but point queries run a factor of $1/\varepsilon$ times slower. While this is already a good tradeoff, the DAM model actually underestimates the performance advantages of the B$^\varepsilon$-tree; the B$^\varepsilon$-tree has performance advantages that cannot be understood in the DAM model.

We first give the affine IO performance of the B$^\varepsilon$-tree:

Lemma 8. *Consider a B$^\varepsilon$-tree with nodes of size B, where the fanout at any nonroot node is within a constant factor of the target fanout F. Then the amortized insertion cost is*

$$O\left(\left(\tfrac{F}{B} + \alpha F\right)\log_F(N/M)\right).$$

*The affine IO cost of a query is*

$$O\left((1 + \alpha B)\log_F(N/M)\right).$$

*The affine IO cost of a range query returning $\ell$ items is $O(1 + \ell/B)(1 + \alpha B)$ plus the cost of the query.*

Proof. We first analyze the query cost. When we perform a query in a B$^\varepsilon$-tree, we follow a root-to-leaf path. We need to search for the element in each buffer along the path, as well as in the target leaf. The cost to read an entire node is $1 + \alpha B$.

We next analyze the amortized insertion/deletion cost. The affine IO cost to flush the $\Theta(B)$ messages in one node one level of the tree is $\Theta(F + \alpha FB)$. This is because there are $\Theta(F)$ IOs (for the node and all children). The total amount of data being flushed to the leaves is $\Theta(B)$, but the total amount of data being transferred from the IOs is $\Theta(FB)$, since nodes that are modified may need to be completely rewritten. Thus, the amortized affine IO cost to flush an element down one level of the tree is $\Theta(F/B + \alpha F)$. The amortized flushing cost follows from the height of the tree.

The impact of tree rebalancing turns out to be a lower-order effect. If the leaves are maintained between half full and full, then in total, there are only $O(N/B)$ modifications to the B$^\varepsilon$-tree's pointer structure in $\Theta(N)$ updates. Thus, the amortized affine IO contribution due to rebalances is $O(\alpha + 1/B)$, which is a low-order term. □

We now describe three affine-model optimizations of the B$^\varepsilon$-tree. These optimizations use variable-sized IOs to improve the query cost of the B$^\varepsilon$-tree without harming its insertion cost, and will enable us to get our robustness and B-tree dominance theorems.

Theorem 9. *There exists a $B^\varepsilon$-tree with nodes of size $B$ and target fanout $F$ with the following bounds. The amortized insertion cost is*

$$O\left(\left(\tfrac{F}{B} + \alpha F\right) \log_F(N/M)\right).$$

*The affine IO cost of a query is at most*

$$\left(1 + \alpha \tfrac{B}{F} + \alpha F\right) \log_F (N/M) \left(1 + 1/\log F\right).$$

*The affine IO cost of a range query returning $\ell$ items is $O((1 + \ell/B)(1 + \alpha B))$ plus the cost of the query.*

Proof. We make three algorithmic decisions to obtain the target performance bounds.

(1) We specify an internal organization of the nodes, and in particular, how the buffers of the nodes are organized.
(2) We store the pivots of a node outside of that node—specifically in the node's parent.
(3) We use a rebalancing scheme in which the nonroot fanouts stay within $(1 \pm o(1))F$.

Our objective is to have a node organization that enables large IOs for insertions/deletions and small IOs for queries—and only one per level.

We first describe the internal node/buffer structure. We organize the nodes/buffer so that all of the elements destined for a particular child are stored contiguously. We maintain the invariant that no more than $B/F$ elements in a node can be destined for a particular child, so the cost to read all these elements is only $1 + \alpha B/F$.

Each node $u$ has a set of pivots. However, we do not store node $u$'s pivots in $u$, but rather in $u$'s parent. The pivots for $u$ are stored next to the buffer that stores elements destined for $u$. When $F = O(\sqrt{B})$, storing a nodes pivots in its parent increases node sizes by at most a constant factor.

Finally, we describe the rebalancing scheme. Define the ***weight*** of a node to be the number of leaves in the node's subtree. We maintain the following weight-balanced invariant. Each nonroot node $u$ at height $h$ satisfies

$$F^h \cdot (1 - 1/\log F) \leq \text{weight}(u) \leq F^h \cdot (1 + 1/\log F).$$

The root just maintains the upper bound on the weight, but not the lower bound.

Whenever a node $u$ gets out of balance, e.g., $u$'s weight grows too large or small, then we rebuild the subtree rooted at $u$'s parent from scratch, thus maintaining the balancing invariant.

We next bound the minimum and maximum fanout that a node can have. Consider a node $u$ and parent node $v$ of height $h$ and $h + 1$, respectively. Then, since weight$(v) \leq F^{h+1}(1 + 1/\log F)$ and weight$(u) \geq F^h(1 - 1/\log F)$, the maximum number of children that $v$ can have is

$$F \cdot \left(\tfrac{1+1/\log F}{1-1/\log F}\right) = F + O\left(\tfrac{F}{\log F}\right).$$

By a similar reasoning, if $v$ is a nonroot node, then the minimum fanout that $v$ can have is $F - O(\frac{F}{\log F})$.

As in Lemma 8, the amortized affine IO cost to flush an element down one level of the tree is $O(F/B + \alpha F)$, and so the amortized insert/delete cost follows from the height of the tree.

The amortized rebalance cost is determined as follows. The IO cost to rebuild the subtree rooted at $u$'s parent $v$ is $O(\text{weight}(v)) = O(F \cdot \text{weight}(u))$, since nodes have size $\Theta(1/\alpha)$ and the cost to access any node is $O(1)$. The number of leaves that are added or removed before $v$ needs to be rebuilt again is $\Omega(\text{weight}(u)/\log F)$. There are $\Omega(1/\alpha)$ inserts or deletes into a leaf before a new leaf gets split or merged. Thus, the number of inserts/deletes into $u$'s subtree between inserts/deletes is $\Omega(\alpha\, \text{weight}(u)/\log F)$. Consequently, the amortized cost to rebuild, per element insert/delete is $O(\alpha \log F)$, which is a low-order cost.

The search bounds are determined as follows. Because the pivot keys of a node $u$ are stored in $u$'s parent, we only need to perform one IO per node, and each IO only needs to read one set of pivots followed by one buffer—not the entire node. Thus, the IO cost per node for searching is $1 + \alpha B/F + \alpha F$, and the search cost follows directly. □

Theorem 9 can be viewed as a sensitivity analysis for the node size $B$, establishing that B$^{\varepsilon}$-trees are less sensitive to variations in the node size than B-trees. This is particularly easy to see when we take $F = \sqrt{B}$.

Corollary 10. *When $B > 1/\alpha$, the B-tree query cost increases nearly linearly in $B$, whereas the $B^{1/2}$-tree ($F = \Theta(\sqrt{B})$) increases nearly linearly in $\sqrt{B}$.*

We now give a more refined sensitivity analysis, optimizing $B$, given $F$ and $\alpha$.

Corollary 11. *When $B = \Omega(F^2)$ and $B = o(F/\alpha)$, there exists $B^{\varepsilon}$-trees where the affine IO cost to read each node is $1 + o(1)$, and the search cost is $(1 + o(1)) \log_F(N/M)$.*

Proof. For a search, the IO cost per node is

$$1 + \alpha B/F + \alpha F = 1 + o(1).$$

This means that the search cost is $(1 + o(1)) \log_F(N/M)$. □

We can now optimize the fanout and node size in Corollary 11. In particular, say that $F = \sqrt{B}$. Then it is sufficient that $B < o(1/\alpha^2)$.

What is interesting about this analysis is that an optimized B$^{\varepsilon}$-tree node size can be nearly the square of the optimal node size for a B-tree for reasonable parameter choices. In contrast, in the DAM model, B-trees and B$^{\varepsilon}$-trees always have the same size, which is the block-transfer size. Small subconstant changes in IO cost can have asymptotic consequences in the data structure design.

This analysis helps explain why the TokuDB B$^{\varepsilon}$-tree has a relatively large node size (~4 MB), but it also has sub-nodes (“basement nodes”), which can be paged in and out independently on searches. It explains the contrast with B-trees, which have much smaller nodes. It is appealing how much asymptotic structure you see just from optimizing the constants and how predictive it is of real data-structure designs.

Finally, we show that in the affine model, we can make a B$^{\varepsilon}$-tree whose search cost is optimal up to low-order terms—see Corollary 7 for an affine optimized B-tree—but whose insert cost is $\Theta(\log(1/\alpha))$ times faster than the insert cost of the optimal B-tree.

Corollary 12. *There exists a $B^{\varepsilon}$-tree with fanout $F = \Theta(1/\alpha \log(1/\alpha))$ and node size $B = F^2$ whose query cost is optimal in the affine model up to low-order terms over all comparison-based external-memory dictionaries. The $B^{\varepsilon}$-tree's query cost matches the B-tree up to low-order terms, but its amortized insert cost is a factor of $\Theta(\log(1/\alpha))$ times faster.*

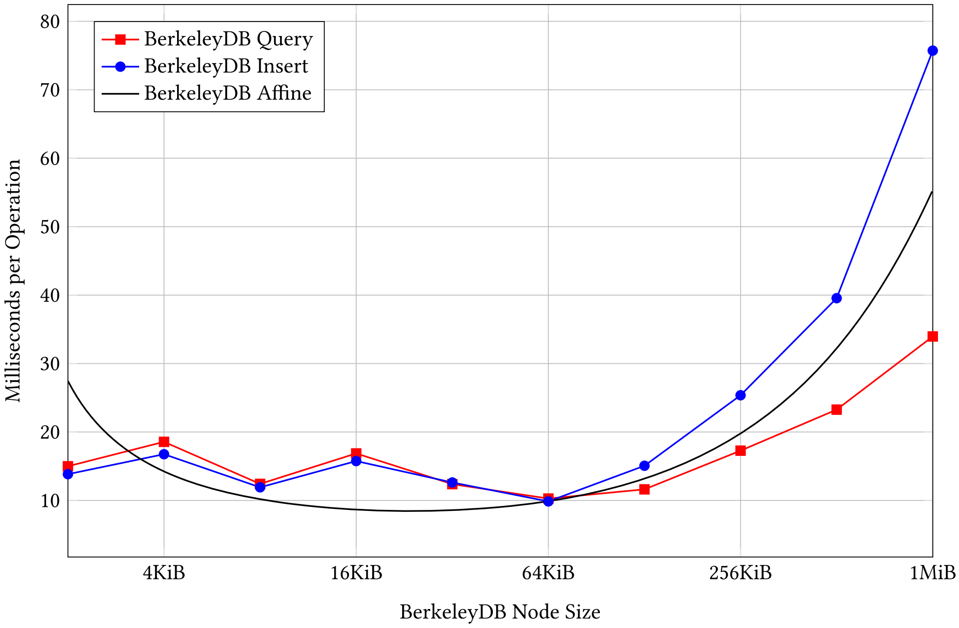


Fig. 2. Microseconds per query/insert with increasing node size for BerkeleyDB. The fitted line (black) has an alpha of $\mathbf{1.58357435 \times 10^{-04}}$ and a root-mean square (RMS) of **8.4**.

## 7 EMPIRICAL VALIDATION OF B-TREE AND B$^{\varepsilon}$-TREE NODE SIZE

We measured the impact of node size on the average run time for random queries and random inserts on HDDs. We used BerkeleyDB [56] as a typical B-tree and TokuDB [67] as a typical B$^{\varepsilon}$-tree. We first inserted 16 GiB of key-value pairs into the database. Then, we performed random inserts and random queries to about a thousandth of the total number of keys in the database. We turned off compression in TokuDB to obtain a fairer comparison, and we limited the memory to 4 GiB to ensure that most of the database contents were outside of RAM.

Figure 2 presents the BerkeleyDB query and insert performance on HDDs. We see that insert and query costs grow once nodes exceed 64 KiB, which is larger than the default node size. After the optimal node size for inserts—64 KiB—the insert and query costs start increasing roughly linearly with the node size, as predicted.

Figure 3 gives performance numbers for TokuDB, which are consistent with Table 3 where $F = \sqrt{B}$. The optimal node size is around 512 KiB for queries and 4 MiB for inserts. In both cases, the next few larger node sizes decrease performance, but only slightly compared to the BerkeleyDB results.

## 8 PDAM DICTIONARIES

We now give some observations on how the PDAM model can inform external-memory dictionary design.

Consider the problem of designing a B-tree for a database that serves a dynamically varying number of clients. We want to exploit the storage device's parallelism, regardless of how many clients are currently performing queries.

If we have $P$ clients, then the optimal strategy is to build our B-tree with nodes of size $B$ and let each client perform one IO per time step. If the database contains $N$ items, then each client requires $\Theta(\log_B N)$ time steps to complete a query (Technically $\Theta(\log_{B+1} N)$, but we use $\Theta(\log_B N)$

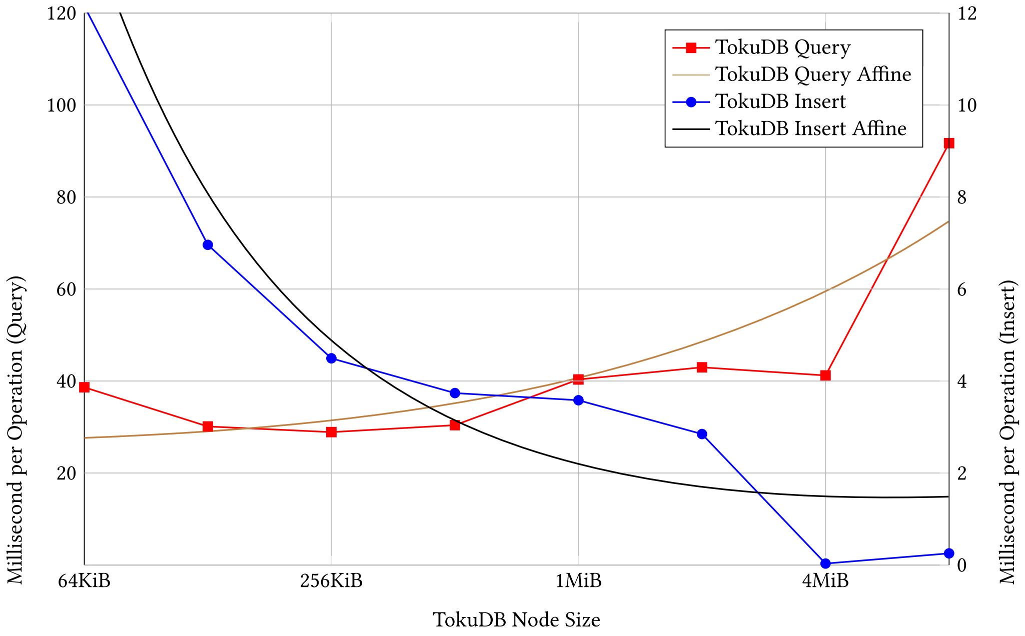


Fig. 3. Number of microseconds per query/insert with increasing node size for TokuDB. The fitted lines have an alpha value of $\mathbf{1.58357435 \times 10^{-03}}$ and the RMS is 18.7.

in this section to keep the math clean). We can answer $P$ queries every $\Theta(\log_B N)$ time-steps, for an amortized throughput of $\Theta(P/\log_B N)$ queries per time-step.

Now suppose that we have a single client performing queries. Since walking the tree from root to leaf is inherently sequential, a B-tree with nodes of size $B$ is unable to use device parallelism. The client completes one query each $\Theta(\log_B N)$ time steps, and all device parallelism goes to waste. Now the B-tree performs better with nodes of size $PB$. The client loads one node per time step, for a total of $\Theta(\log_{PB} N)$ time-steps per query, which is a significant speed-up when $P = \omega(B)$.

In summary, to optimize performance, we want nodes of size $B$ when we have many clients, and nodes of size $PB$ when we have only one client. But B-trees have a fixed node size.

The point is that the amount of IO that can be devoted to a query is not predictable. Dilemmas like this are common in external-memory dictionary design, e.g., when dealing with system prefetching [20, 31].

One way to resolve the dilemma uses ideas from cache-oblivious data-structures [14, 40]. Cache-oblivious data structures are universal data structures in that they are optimal for all memory-hierarchy parameterizations. Most cache-oblivious dictionaries are based on the van Emde Boas layout [14, 60].

In the B-tree example, we use nodes of size $PB$, but organize each node in a van Emde Boas layout. Now suppose there are $k \leq P$ clients. Each client is given $P/k$ IOs per time slot. Thus, a client can traverse a single node in $\Theta(\log_{PB/k} PB)$ time steps, and hence traverses the entire root-to-leaf path in $\Theta(\log_{PB/k} N)$ time steps. When $k = 1$, this matches the optimal single-threaded B-tree design described above. When $k = P$, this matches the optimal multi-threaded client throughput given above. Furthermore, this design gracefully adapts when the number of clients varies over time.

Lemma 13. *In the PDAM model, a B-tree with $N$ elements and nodes of size $PB$ has query throughput $\Omega(\frac{k}{\log_{PB/k} N})$ for any $k \leq P$ concurrent query threads.*

This strategy also fits well into current caching and prefetching system designs. In each time step, clients issue IOs for single blocks. Once the system has collected all the IO requests, if there are any unused IO slots in that time step, then it expands the requests to perform read-ahead. So, in our B-tree example with a single client, there is only one IO request (for the first block in a node), and the system expands that to $P$ blocks, effectively loading the entire node into cache. As the client accesses the additional blocks of the same B-tree node during its walk of the van Emde Boas layout, the blocks are in cache and incur no additional IO. If, however, there are two clients, then the system sees two one-block IO requests, which it will expand into two runs of $P/2$ blocks each.

This basic strategy extends to other cache-oblivious data structures; see e.g., [15, 25] for write-optimized examples. The PDAM explains how these structures can always make maximal use of device parallelism, even as the number of concurrent threads changes dynamically.

## 9 CONCLUSION

As the analyses in this article illustrate, seemingly small changes in the DAM model have substantial design and performance implications. The more accurate computational models that we have considered are more predictive of running times and software practice. We posit that these models are an essential tool for algorithmists seeking to design new algorithms for IO-bound workloads. The simplicity of the DAM model is a strength but also has let important design considerations slip through the cracks for decades.

## REFERENCES

[1] mkfs.btrfs. [n.d.]. Manual page. Retrieved from https://btrfs.wiki.kernel.org/index.php/Manpage/mkfs.btrfs.

[2] Alok Aggarwal and Jeffrey Scott Vitter. 1988. The input/output complexity of sorting and related problems. *Commun. ACM* 31, 9 (Sept. 1988), 1116–1127. https://doi.org/10.1145/48529.48535

[3] Matthew Andrews, Michael A. Bender, and Lisa Zhang. 2002. New algorithms for disk scheduling. *Algorithmica* 32, 2 (2002), 277–301. https://doi.org/10.1007/s00453-001-0071-1

[4] Lars Arge. 2002. External-memory geometric data structures. In *Proceedings of the EEF Summer School on Massive Data Sets*.

[5] Lars Arge, Michael T. Goodrich, Michael Nelson, and Nodari Sitchinava. 2008. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures (SPAA'08)*. 197–206. https://doi.org/10.1145/1378533.1378573

[6] Lars Arge, Michael T. Goodrich, and Nodari Sitchinava. 2010. Parallel external-memory graph algorithms. In *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS'10)*. IEEE, 1–11. https://doi.org/10.1109/IPDPS.2010.5470440

[7] Microsoft Azure. 2016. How to Use Batching to Improve SQL Database Application Performance. Retrieved from https://docs.microsoft.com/en-us/azure/sql-database/sql-database-use-batching-to-improve-performance.

[8] Rudolf Bayer and Edward M. McCreight. 1972. Organization and maintenance of large ordered indexes. *Acta Info.* 1, 3 (Feb. 1972), 173–189. https://doi.org/10.1145/1734663.1734671

[9] Naama Ben-David, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. 2016. Parallel algorithms for asymmetric read-write costs. In *Proceedings of the 28th ACM on Symposium on Parallelism in Algorithms and Architectures (SPAA'16)*. 145–156. https://doi.org/10.1145/2935764.2935767

[10] Michael A. Bender, Jon Berry, Rob Johnson, Thomas M. Kroeger, Samuel McCauley, Cynthia A. Phillips, Bertrand Simon, Shikha Singh, and David Zage. 2016. Anti-persistence on persistent storage: History-independent sparse tables and dictionaries. In *Proceedings of the 35th ACM Symposium on Principles of Database Systems (PODS'16)*. 289–302.

[11] Michael A. Bender, Jake Christensen, Alex Conway, Martin Farach-Colton, Rob Johnson, and Meng-Tsung Tsai. 2019. Optimal ball recycling. In *Proceedings of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'19)*. 2527–2546. https://doi.org/10.1137/1.9781611975482.155

[12] Michael A. Bender, Alex Conway, Martín Farach-Colton, William Jannen, Yizheng Jiao, Rob Johnson, Eric Knorr, Sara McAllister, Nirjhar Mukherjee, Prashant Pandey, Donald E. Porter, Jun Yuan, and Yang Zhan. 2019. Small refinements to the DAM can have big consequences for data-structure design. In *Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'19)*. 265–274.

[13] Michael A. Bender, Rathish Das, Rob Johnson, Martín Farach-Colton, and William Kuszmaul. 2020. Flushing without Cascades. In *Proceedings of the 31th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'20)*. 650–669.

[14] Michael A. Bender, Erik Demaine, and Martin Farach-Colton. 2005. Cache-Oblivious B-Trees. *Journal on Computing* 35, 2 (2005), 341–358.

[15] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. 2007. Cache-oblivious streaming B-trees. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'07)*. 81–92. https://doi.org/10.1145/1248377.1248393

[16] Michael A. Bender, Martin Farach-Colton, Mayank Goswami, Rob Johnson, Samuel McCauley, and Shikha Singh. 2018. Bloom filters, adaptivity, and the dictionary problem. In *Proceedings to the IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS'18)*. 182–193.

[17] Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. 2015. An introduction to B$^{\varepsilon}$-trees and write-optimization. *:login; Mag.* 40, 5 (Oct. 2015), 22–28.

[18] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. 2012. Don't thrash: How to cache your hash on flash. *Proc. Very Large Data Bases Endow.* 5, 11 (2012), 1627–1637.

[19] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Simon Mauras, Tyler Mayer, Cynthia A. Phillips, and Helen Xu. 2017. Write-optimized skip lists. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS'17)*. 69–78. https://doi.org/10.1145/3034786.3056117

[20] Michael A. Bender, Martin Farach-Colton, and Bradley Kuszmaul. 2006. Cache-oblivious string B-trees. In *Proceedings of the 25th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'06)*. 233–242.

[21] Michael A. Bender, Martín Farach-Colton, and William Kuszmaul. 2019. Achieving optimal backlog in multi-processor cup games. In *Proceedings of the 51st Annual ACM Symposium on the Theory of Computing (STOC'19)*.

[22] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, and Julian Shun. 2016. Efficient algorithms with asymmetric read and write costs. In *Proceedings of the 24th Annual European Symposium on Algorithms (ESA'16)*. 14:1–14:18. https://doi.org/10.4230/LIPIcs.ESA.2016.0

[23] Guy E. Blelloch, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. 2018. The parallel persistent memory model. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA'18)*. 247–258. https://doi.org/10.1145/3210377.3210381

[24] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. 2010. Low depth cache-oblivious algorithms. In *Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'10)*, Friedhelm Meyer auf der Heide and Cynthia A. Phillips (Eds.). ACM, 189–199. https://doi.org/10.1145/1810479.1810519

[25] Gerth S. Brodal, Erik D. Demaine, Jeremy T. Fineman, John Iacono, Stefan Langerman, and J. Ian Munro. 2010. Cache-oblivious dynamic dictionaries with update/query tradeoffs. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'10)*. 1448–1456.

[26] Gerth S. Brodal and Rolf Fagerberg. 2003. Lower bounds for external-memory dictionaries. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'03)*. 546–554.

[27] Adam L. Buchsbaum, Michael H. Goldwasser, Suresh Venkatasubramanian, and Jeffery Westbrook. 2000. On external-memory graph traversal. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'00)*. 859–860.

[28] Mark Callaghan. 2011. Something Awesome in InnoDB—The Insert Buffer. Retrieved from https://www.facebook.com/notes/mysql-at-facebook/something-awesome-in-innodb-the-insert-buffer/492969385932/.

[29] Mustafa Canim, Christian A. Lang, George A. Mihaila, and Kenneth A. Ross. 2010. Buffered Bloom filters on solid state storage. In *Proceedings of the International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS'10)*.

[30] Feng Chen, Binbing Hou, and Rubao Lee. 2016. Internal parallelism of flash memory-based solid-state drives. *Trans. Stor.* 12, 3, Article 13 (May 2016), 39 pages. https://doi.org/10.1145/2818376

[31] Shimin Chen, Phillip B. Gibbons, Todd C. Mowry, and Gary Valentin. 2002. Fractal prefetching B$^{*}$-Trees: Optimizing both cache and disk performance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 157–168.

[32] Rezaul Alam Chowdhury, Vijaya Ramachandran, Francesco Silvestri, and Brandon Blakeley. 2013. Oblivious algorithms for multicores and networks of processors. *J. Parallel Distrib. Comput.* 73, 7 (2013), 911–925. https://doi.org/10.1016/j.jpdc.2013.04.008

[33] Douglas Comer. 1979. The Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (June 1979), 121–137.

[34] Alexander Conway, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A. Bender, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, and Martin Farach-Colton. 2017. File systems fated for senescence? Nonsense, says science! In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. 45–58.

[35] Alex Conway, Ainesh Bakshi, Yizheng Jiao, Yang Zhan, Michael A. Bender, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Martin Farach-Colton. 2017. How to fragment your file system. *login:* 42, 2 (2017). Retrieved from https://www.usenix.org/publications/login/summer2017/conway.

[36] Alexander Conway, Martin Farach-Colton, and Philip Shilane. 2018. Optimal hashing in external memory. In *Proceedings of the 45th International Colloquium on Automata, Languages and Programming (ICALP'18).* 39:1–39:14. https://doi.org/10.4230/LIPIcs.ICALP.2018.39

[37] Alex Conway, Eric Knorr, Yizheng Jiao, Michael A. Bender, William Jannen, Rob Johnson, Donald E. Porter, and Martin Farach-Colton. 2019. Filesystem aging: It's more usage than fullness. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'19).*

[38] Peter Desnoyers. 2013. What systems researchers need to know about NAND flash. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'13).*

[39] John Esmet, Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. 2012. The TokuFS streaming file system. In *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'12).* 14.

[40] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 2012. Cache-oblivious algorithms. *ACM Trans. Algor.* 8, 1 (2012), 4.

[41] Pedram Ghodsnia, Ivan T. Bowman, and Anisoara Nica. 2014. Parallel I/O aware query optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 349–360. https://doi.org/10.1145/2588555.2595635

[42] Google, Inc. [n.d.]. LevelDB: A Fast and Lightweight Key/value Database Library By Google. Retrieved from https://github.com/google/leveldb.

[43] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. The unwritten contract of solid state drives. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys'17).* 127–144.

[44] IBM. 2017. Buffered Inserts in Partitioned Database Environments. Retrieved from https://www.ibm.com/support/knowledgecenter/SSEPGG_10.5.0/com.ibm.db2.luw.apdv.embed.doc/doc/c0061906.html.

[45] IBM Informix. [n.d.]. Understanding SQL Insert Cursors. Retrieved from https://www.ibm.com/support/knowledgecenter/en/SSBJG3_2.5.0/com.ibm.gen_busug.doc/c_fgl_InsertCursors_002.htm.

[46] Riko Jacob and Nodari Sitchinava. 2017. Lower bounds in the asymmetric external-memory model. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'17).* 247–254. https://doi.org/10.1145/3087556.3087583

[47] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2015. BetrFS: A right-optimized write-optimized file system. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15).* 301–315.

[48] Chris Jermaine, Anindya Datta, and Edward Omiecinski. 1999. A novel index supporting high volume data warehouse insertion. In *Proceedings of 25th International Conference on Very Large Data Bases (VLDB'99).* 235–246. Retrieved from http://www.vldb.org/conf/1999/P23.pdf.

[49] Bradley C. Kuszmaul. 2009. How fractal trees work. In *Proceedings of the OpenSQL Camp.*

[50] Amanda McPherson. [n.d.]. A Conversation with Chris Mason on Btrfs: The Next Generation File System for Linux. Retrieved from https://www.linuxfoundation.org/blog/2009/06/a-conversation-with-chris-mason-on-btrfs/.

[51] MySQL 5.7 Reference Manual. [n.d.]. Chapter 15 The InnoDB Storage Engine. Retrieved from http://dev.mysql.com/doc/refman/5.7/en/innodb-storage-engine.html.

[52] NuDB. 2016. NuDB: A Fast Key/value Insert-only Database for SSD Drives in C++11. Retrieved from https://github.com/vinniefalco/NuDB.

[53] Patrick O'Neil, Edward Cheng, Dieter Gawlic, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Info.* 33, 4 (1996), 351–385. https://doi.org/10.1007/s002360050048

[54] Oracle. 2017. Tuning the Database Buffer Cache. Retrieved from https://docs.oracle.com/database/121/TGDBA/tune_buffer_cache.htm.

[55] Oracle Corporation. [n.d.]. MySQL 5.5 Reference Manual. Retrieved from https://dev.mysql.com/doc/refman/5.5/en/innodb-file-space.html.

[56] Oracle Corporation. 2015. Oracle BerkeleyDB Reference Guide. Retrieved from http://sepp.oetiker.ch/subversion-1.5.4-rp/ref/am_conf/pagesize.html.

[57] Oracle Corporation. 2016. Setting Up Your Data Warehouse System. Retrieved from https://docs.oracle.com/cd/B28359_01/server.111/b28314/tdpdw_system.htm.

[58] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2016. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'16).* 537–550.

[59] John Paul. [n.d.]. Teradata Thoughts. Retrieved from http://teradata-thoughts.blogspot.com/2013/10/teradata-13-vs-teradata-14_20.html.

[60] Harald Prokop. 1999. *Cache-Oblivious Algorithms*. Master's thesis. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

[61] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. 497–514. https://doi.org/10.1145/3132747.3132765

[62] Mendel Rosenblum and John K. Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.* 10, 1 (Feb. 1992), 26–52. https://doi.org/10.1145/146941.146943

[63] C. Ruemmler and J. Wilkes. 1994. An introduction to disk drive modeling. *IEEE Computer* 27, 3 (1994), 17–29.

[64] SAP. 2017. RLV Data Store for Write-Optimized Storage. Retrieved from http://help-legacy.sap.com/saphelp_iq1611_iqnfs/helpdata/en/a3/13783784f21015bf03c9b06ad16fc0/content.htm.

[65] Keith A. Smith and Margo I. Seltzer. 1997. File system aging—Increasing the relevance of file system benchmarks. In *Measurement and Modeling of Computer Systems*. 203–213.

[66] SNIA. 2018. Solid State Storage (SSS) Performance Test Specification (PTS). Retrieved from https://www.snia.org/sites/default/files/technical_work/PTS/SSS_PTS_2.0.1.pdf.

[67] TokuDB. [n.d.]. https://github.com/percona/PerconaFT, Last Accessed Sep. 24 2018..

[68] Tokutek, Inc. [n.d.]. TokuMX—MongoDB Performance Engine. Retrieved from https://www.percona.com/software/mongo-database/percona-tokumx.

[69] Tokutek, Inc. 2013. TokuDB: MySQL Performance, MariaDB Performance. Retrieved from http://www.tokutek.com/products/tokudb-for-mysql/.

[70] Vertica. 2017. WOS (Write Optimized Store). Retrieved from https://my.vertica.com/docs/7.1.x/HTML/Content/Authoring/Glossary/WOSWriteOptimizedStore.htm.

[71] Jeffrey Scott Vitter. 2001. External-memory algorithms and data structures: Dealing with massive data. *ACM Comput. Surveys* 33, 2 (2001), 209–271.

[72] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. 1994. Algorithms for parallel memory, I: Two-level memories. *Algorithmica* 12, 2-3 (1994), 110–147.

[73] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. 1994. Algorithms for parallel memory, II: Hierarchical multilevel memories. *Algorithmica* 12, 2–3 (1994), 148–169.

[74] James Christopher Wyllie. 1979. *The Complexity of Parallel Computations*. Ph.D. Dissertation. Ithaca, NY. AAI8004008.

[75] Jimmy Xiang. 2012. Apache HBase Write Path. Retrieved from http://blog.cloudera.com/blog/2012/06/hbase-write-path/.

[76] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2016. Optimizing every operation in a write-optimized file system. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. 1–14.

[77] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2017. Writes wrought right, and other adventures in file system optimization. *Trans. Stor.* 13, 1 (2017), 3:1–3:26.

[78] Yang Zhan, Alexander Conway, Yizheng Jiao, Eric Knorr, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. 2018. The full path to full-path indexing. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*. 123–138.

Received January 2020; revised October 2020; accepted February 2021