Small Refinements to the DAM Can Have Big Consequences for Data-Structure Design

Michael A. Bender
Stony Brook University
bender@cs.stonybrook.edu

Alex Conway
Rutgers University and
VMware Research
alexander.conway@rutgers.edu

Martín Farach-Colton
Rutgers University
martin@farach-colton.com

William Jannen
Williams College
jannen@cs.williams.edu

Yizheng Jiao
The University of North Carolina at
Chapel Hill
yizheng@cs.unc.edu

Rob Johnson
VMware Research
robj@vmware.com

Eric Knorr
Rutgers University
eric.r.knorr@gmail.com

Sara McAllister
Harvey Mudd College
smcallister@g.hmc.edu

Nirjhar Mukherjee
The University of North Carolina at
Chapel Hill
nirjhar@unc.edu

Prashant Pandey
Carnegie Mellon University
ppandey2@cs.cmu.edu

Donald E. Porter
The University of North Carolina at
Chapel Hill
porter@cs.unc.edu

Jun Yuan
Pace University
jyuan2@pace.edu

Yang Zhan
The University of North Carolina at
Chapel Hill
yzhan@cs.unc.edu

ABSTRACT

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.

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, the DAM approximates the IO cost on any hardware to within a factor of 2.

Furthermore, the DAM explains the popularity of B-trees in the 70s 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 paper, 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 the B-tree is 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.

SPAA '19, June 22–24, 2019, Phoenix, AZ, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6184-2/19/06…$15.00
https://doi.org/10.1145/3323165.3323210

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. 2019. Small Refinements to the DAM Can Have Big Consequences for Data-Structure

Design. In *31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '19), June 22–24, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3323165.3323210

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 has been a success [4, 64], 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 point***, i.e., the IO size where the latency and bandwidth of the hardware are equal, then the DAM model predicts the IO cost of any algorithm to within a factor of 2 on that hardware. That is, if an algorithm replaces all its IOs with IOs of the half-bandwidth size, then the cost of the IOs increase by a factor of at most 2.

Furthermore, the DAM explains some choices that software architects have made. For example, the DAM model gives an analytical explanation for why B-trees [6, 27] took over in the 70s and why B$^{\varepsilon}$-trees [13, 21], log-structured merge trees [11, 47], and external-memory skip lists [8, 15, 55] are taking over now.

But the DAM model has its limits. For example, the DAM does not explain why B-trees in many databases and file systems use nodes of size 16KiB [1, 44, 45, 49, 50], which is well below the half-bandwidth point on most storage devices, whereas B-trees optimized for range queries use larger node sizes, typically up to around 1MB [51, 53]. Nor does it explain why TokuDB's [62] B$^{\varepsilon}$-tree uses 4MiB nodes and LevelDB's [36] LSM-tree uses 2MiB SSTables for all workloads. In a DAM, all IOs, and hence all nodes, are the same size.

How can a 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 paper, we show that the ***affine*** [3, 57] 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 P IOs of size B in parallel.

Results

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

The affine and PDAM models improve the estimate of the IO cost. In §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 find 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, on the other hand, overestimates the completion time for large numbers of threads by roughly P, the parallelism of the device, which ranges from 2.5 to 12. 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 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.

The affine and PDAM models explain software design choices. In §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 point, whereas B$^{\varepsilon}$-trees have nodes that are larger than the half-bandwidth point. 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 §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 looses a factor of 2, it is blind to such fine-grained optimizations.

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

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

We present the affine model (most predictive of hard disks) and the PDAM (most predictive of SSDs). The DAM simplifies analysis by assuming 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.

We show that these models are accurate across a range of hardware even though they do not explicitly model most hardware effects. Since they are minor refinements of the DAM, when designing data structures, we can reason in the DAM and then optimize in the affine/PDAM models.

[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.

2.1 Disk Access Machine (DAM) Model

The disk-access machine (DAM) model [2] assumes a two-level cache hierarchy with a cache of size M words and a slower storage device that transfers data to and from cache in blocks of size B. 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. It 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. On devices with bounded boundwidth and IO setup time, we can ensure that the number of IOs is within a constant factor of the wall-clock time spent performing IO by setting B to the half-bandwidth point of the device.

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 and NVMe devices, it does not model internal device parallelism nor the incremental cost of larger IOs.

Inaccuracies of DAM. 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 [13, 22, 27]. 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 [13, 16].

2.2 SSDs and the PDAM

The PDAM model improves performance analysis in SSDs and NVMe 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 from RAID arrays. However, P is largely ignored in the literature, and almost all theoretical work has been for $P = 1$.

We argue for reviving P to model the parallelism of SSDs and NVMe devices. Flash storage performs IOs at page granularity, typically 512B–16KB, but has parallelism in terms of multiple channels, packages per channel, and even dies per package [32, 37]. This parallelism is why applications must maintain deep queues in order to get full bandwidth out of an SSD or NVMe device [25, 35].

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 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 paper, IOs are concurrent-read-exclusive-write (CREW) [65] (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 cost*** of s seconds, and then reads data locally off the disk at transfer cost of t seconds/byte, which corresponds to a ***bandwidth cost***.

Parameters s and t are approximate, since the setup cost can vary by an order of magnitude. E.g., a track-to-track may be ∼1ms while a full seek is ∼10ms. Nonetheless, it is remarkably predictive to view these as fixed [57].

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 bandwidth cost.*

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

Lemma 1. *An affine algorithm with cost C can be transformed into a DAM algorithm with cost $2C$, where blocks have size $B = 1/\alpha$. A DAM algorithm with cost C and blocks of sizes $B = 1/\alpha$ can be transformed into an affine algorithm with cost $2C$.*

Thus, if losing a factor of 2 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 factor of 2. 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 paper when we were building and marketing TokuDB [62]. In fact, losing a factor of 2 on large sequential write performance was a serious setback on making BεtrFS a general-purpose file system [28, 41, 67–69].

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 [6, 27]. 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 $\Theta(\log_{B+1} N)$.

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

The systems community often evaluates data structures in terms of their ***write amplification***, which we define below [56].

Definition 3. *The **write amplification** of an update is the amortized amount of data written to disk per operation divided by the amount of data modified per 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) writes are more expensive than reads, and this has algorithmic consequences [7, 18, 19, 40]. 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 give a bad example for write amplification. Consider sufficiently large N where $N = \Omega(BM)$. Assume that nodes are paged to and from RAM atomically. Then a workload comprised of random insertions and deletions achieves this write amplification. On average, a (sized B) node is written back to disk after there have been $O(1)$ (unit-sized) elements written to/deleted from that node.

The upper bound on write amplification follows because the modifications that take place on the tree are dominated by the modifications at the leaves. □

B$^\varepsilon$-trees. The B$^\varepsilon$-tree [21, 22, 42] 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 [28, 33, 41, 41, 43, 52, 60, 61, 67–69]. A more detailed description of the B$^\varepsilon$-tree is available in the prior literature [13].

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 ε ($0 \le \varepsilon \le 1$) that controls the fanout $F = B^\varepsilon + 1$. 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 reduce to a buffered repository tree [22]. Setting ε to a constant in between leads to point-query performance that is within a constant factor 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 ([13, 21]). *In a B$^\varepsilon$-tree with size-B nodes and fanout* $1 + B^\varepsilon$, *for* $\varepsilon \in [0, 1]$,

1. *insertions and deletions take* $O(\frac{1}{B^{1-\varepsilon}} \log_{B^\varepsilon+1}(N/M))$ *IOs,*
2. *point queries take* $O(\log_{B^\varepsilon+1}(N/M))$ *IOs, and*
3. *a range query returning* ℓ *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 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, they have much more predictive power. We can even make predictions and reason about constants. As we will see in the next section, optimizing the constant for various operations will cause some design parameters to change asymptotically.

Unless noted otherwise, all experiments in this paper were collected on a Dell PowerEdge T130 with a 3.0GHz Intel E3-1220 v6 processor, 32GiB of DDR4 RAM, two 500GiB TOSHIBA DT01ACA050 HDDs and one 250GiB Samsung 860 EVO SSD.

4.1 Validating the PDAM Model

The PDAM ignores some issues of real SSDs, such as bank conflicts, which can limit the parallelism available for some sets of IO requests. Despite its simplicity, we verify below that the PDAM accurately reflects real SSD and NVMe performance.

Interestingly, the PDAM predates commercial SSDs [2], but, as we verify, the PDAM fits the performance of a wide range of SSDs. The goodness of fit is particularly striking because SSDs have many complications that are not captured by the PDAM.

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 $p = \{1, 2, 4, 8, \ldots, 64\}$ OS threads that each read 10 GiB of data. We selected 163, 840 random logical block address (LBA) offsets and read 64KiB starting from each. Thus, there were always p outstanding IO requests, and the total data read was $p \times 10$GiB per round.

The PDAM predicts that the time to complete the experiment should be the same for all $p \le P$ and should increase linearly in p for $p > P$. Figure 1 shows the time in seconds taken to perform each round of IO read experiments. As Figure 1 shows, the time is relatively constant until around $p = 2$ or 4, depending on the device, and it increases linearly thereafter. The transition is not perfectly sharp. We suspect this is do to bank conflicts within the device.

We used segmented linear regression to estimate P and B for each device. 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

Device	P	$\propto PB$	R^2
Samsung 860 pro	3.3	530	0.999
Samsung 970 pro	5.5	2500	0.986
Silicon Power S55	2.9	260	0.999
Sandisk Ultra II	4.6	520	0.993

Table 1: Experimentally derived PDAM values for real hardware. We used segmented linear regression to calculate P. After P threads, throughput remains nearly constant at $\propto PB$.

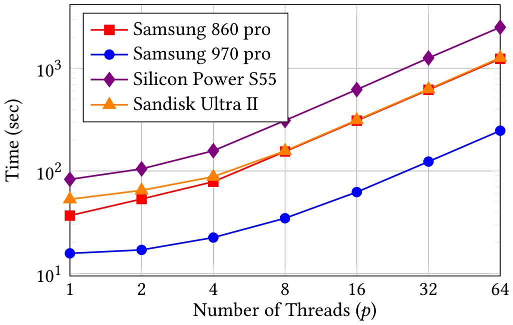

Figure 1: Time to read 10GiB per thread on each SSD tested. The DAM model predicts that the time would increase linearly with the number of LBAs read. However, for a small number of threads the time stays relatively constant.

Disk	Year	s (s)	t (s/4K)	α	R^2
2 TB Seagate	2002	0.018	0.000021	0.0012	0.9994
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
6 TB WD Red	2018	0.016	0.000026	0.0017	0.9972

Table 2: Experimentally derived α values for commodity HDDs. We issued 64 random block-aligned reads with IO sizes ranging from 1 disk block to 16MiB. We conducted linear regression to get the setup cost s and bandwidth cost t. We calculated α by t/s.

	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	$\Theta\left(\frac{F(1+\alpha B)}{B \log F} \log \frac{N}{M}\right)$	$\Theta\left(\frac{F+\alpha F^2+\alpha B}{F \log F} \log \frac{N}{M}\right)$

Table 3: A sensitivity analysis of node sizes for B$^\varepsilon$-trees and B-trees. 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 upserts 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.

each region. Table 1 shows the experimentally derived parallelism, P, and the device saturation, $\propto PB$, 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 0.1% of 1, and we conclude that the PDAM is an excellent fit for SSDs.

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.

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 16MiB. Table 2 shows the experimentally derived values for each HDD. To verify the goodness of fit, we report the R^2 value; a value of 1 indicates that the regression coefficients perfectly predict the observed data. R^2 values are all within 0.1% of 1, and we conclude that the affine model is an excellent fit for hard disks.

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 ℓ 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 $1 + \alpha B$. 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.

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 ℓ 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 $B = 1/\alpha$ achieves the half-bandwidth point. □

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 point 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, 44, 45, 49, 50], which is too small to amortize the setup cost. As a result, range queries run slowly, under-utilizing disk bandwidth [28, 29, 59]. In contrast, B-trees in databases that are more focused on analytical workloads use larger block sizes, typically up to around 1MB [51, 53], 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 their half-bandwidth point.

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 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. □

A straightforward information-theoretic argument shows that Corollary 7 is optimal not just for B-trees, but also for any comparison-based external-memory dictionary.

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 disk, and range-query performance degrades. This is borne out in practice [28, 29, 31, 59].

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 it 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 [30], despite optimizations [5, 9, 12, 14, 17, 23, 24, 38, 39, 46, 48, 58, 63, 66].

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 [13, 21] in the affine model, assuming that IOs only read entire nodes—effectively the natural generalization of the DAM 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 then describe three optimizations to the B$^\varepsilon$-tree that eliminate this trade-off.

This latter result is particularly exciting because, in the DAM model, there is a tight tradeoff between reads and writes [21]. 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.

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*

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

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

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

The affine IO cost of a query is at most

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

The affine IO cost of a range query returning ℓ 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

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

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 v from scratch, reestablishing 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$(v) \leq F^h(1 + 1/\log F)$ and weight$(u) \leq F^h(1 - 1/\log F)$, the maximum number of children that v can have is

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

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

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 $O(\text{weight}(v)) = O(F\,\text{weight}(u))$, 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. For B$^\varepsilon$-trees, insertion are much less sensitive to changes in node size than insertions. 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 α.

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 parameters choices. In contrast, in the DAM, 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 (~4MB), but 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 that you see just from optimizing the constants and how predictive it is of real data-structure designs.

Finally, we show that even in the affine model we can make a B$^\varepsilon$-tree whose search cost is optimal up to low-order terms and whose insert cost is asymptotically faster than a 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.*

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 [50] as a typical B-tree and TokuDB [60] as a typical B$^\varepsilon$-tree. We first inserted 16 GB 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. We limited the memory to 4 GiB to ensure that most of the databases were outside of RAM.

Figure 2 presents the query and insert performance of BerkeleyDB on HDDs. We see that the cost of inserts and queries starts to grow once the nodes are larger than 64 KiB, which is larger than the default node size. After the optimal node size of 64 KiB for inserts, 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 CONCLUSIONS AND PDAM ALGORITHM DESIGN

We conclude with 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 $\Theta(\log_B N)$ in this section in order 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.

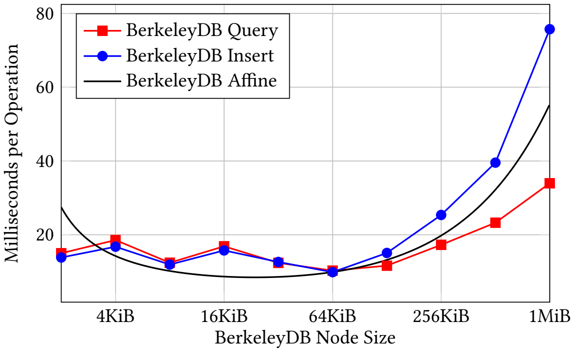

Figure 2: Microseconds per query/insert with increasing node size for BerkeleyDB. The fitted line (black) has an alpha of $1.58357435 \times 10^{-04}$ and a root mean squared (RMS) of 8.4.

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 [16, 26].

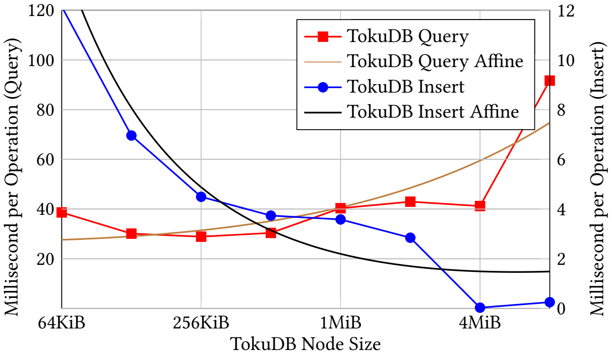

Figure 3: Number of microseconds per query/insert with increasing node size for TokuDB. The fitted lines have an alpha value of $1.58357435 \times 10^{-03}$ and the RMS is 18.7.

One way to resolve the dilemma uses ideas from cache-oblivious data-structures [10, 34]. 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 [10, 54].

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 nodes of size PB has query throughput* $\Omega\left(\frac{k}{\log_{PB/k} N}\right)$ *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, on the other hand, 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., [11, 20] 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.

As this and earlier examples illustrate, seemingly small changes in the DAM model have substantial performance and design implications. The more accurate computational models that we consider are more predictive of 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 has let important design considerations slip through the cracks for decades.

ACKNOWLEDGMENTS

We gratefully acknowledge support from NSF grants CCF 805476, CCF 822388, CNS 1408695, CNS 1755615, CCF 1439084, CCF 1725543, CSR 1763680, CCF 1716252, CCF 1617618, IIS 1247726, and from VMware and a Netapp Faculty Fellowship.

REFERENCES

[1] [n. d.]. mkfs.btrfs Manual Page. https://btrfs.wiki.kernel.org/index.php/Manpage/mkfs.btrfs, Last Accessed Sep. 26, 2018.

[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. *Lecture notes of EEF Summer School on Massive Data Sets, Aarhus* (2002).

[5] Microsoft Azure. 2016. How to use batching to improve SQL Database application performance. https://docs.microsoft.com/en-us/azure/sql-database/sql-database-use-batching-to-improve-performance.

[6] Rudolf Bayer and Edward M. McCreight. 1972. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica* 1, 3 (Feb. 1972), 173–189. https://doi.org/10.1145/1734663.1734671

[7] 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)*. 145–156. https://doi.org/10.1145/2935764.2935767

[8] 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)*. 289–302.

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

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

[11] 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)*. 81–92. https://doi.org/10.1145/1248377.1248393

[12] 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)*. 182–193.

[13] 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; magazine* 40, 5 (October 2015), 22–28.

[14] 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. *In Proceedings of the Very Large Data Bases (VLDB) Endowment* 5, 11 (2012), 1627–1637.

[15] 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)*. 69–78. https://doi.org/10.1145/3034786.3056117

[16] 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)*. 233–242.

[17] 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)*.

[18] 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)*. 14:1–14:18. https://doi.org/10.4230/LIPIcs.ESA.2016.0

[19] 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)*. 247–258. https://doi.org/10.1145/3210377.3210381

[20] 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)*. 1448–1456.

[21] 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)*. 546–554.

[22] Adam L. Buchsbaum, Michael H. Goldwasser, Suresh Venkatasubramanian, and Jeffery Westbrook. 2000. On External Memory Graph Traversal. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 859–860.

[23] Mark Callaghan. 2011. Something awesome in InnoDB – the insert buffer. https://www.facebook.com/notes/mysql-at-facebook/something-awesome-in-innodb-the-insert-buffer/492969385932/.

[24] Mustafa Canim, Christian A. Lang, George A. Mihaila, and Kenneth A. Ross. 2010. Buffered Bloom filters on solid state storage. In *International Workshop*

on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - (ADMS).

[25] Feng Chen, Binbing Hou, and Rubao Lee. 2016. Internal Parallelism of Flash Memory-Based Solid-State Drives. *Transactions on Storage (TOS)* 12, 3, Article 13 (May 2016), 39 pages. https://doi.org/10.1145/2818376

[26] 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 2002 ACM SIGMOD International Conference on Management of Data*. 157–168.

[27] Douglas Comer. 1979. The Ubiquitous B-Tree. 11, 2 (June 1979), 121–137.

[28] 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 *15th USENIX Conference on File and Storage Technologies (FAST)*. 45–58.

[29] 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). https://www.usenix.org/publications/login/summer2017/conway

[30] 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)*. 39:1–39:14. https://doi.org/10.4230/LIPIcs.ICALP.2018.39

[31] Alex Conway, Eric Knorr, Yizheng Jiao, Michael A. Bender, William Jannen, Rob Johnson, Donald E. Porter, and Martin Farach-Colton. 2019. Filesystem Aging: ItâĂŹs more Usage than Fullness. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.

[32] 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)*.

[33] 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)*. 14.

[34] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 2012. Cache-Oblivious Algorithms. *ACM Transactions on Algorithms (TALG)* 8, 1 (2012), 4.

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

[36] Google, Inc. [n. d.]. LevelDB: A fast and lightweight key/value database library by Google. https://github.com/google/leveldb, Last Accessed Sep. 26, 2018.

[37] 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 Twelfth European Conference on Computer Systems (EuroSys)*. 127–144.

[38] IBM. 2017. Buffered inserts in partitioned database environments. https://www.ibm.com/support/knowledgecenter/SSEPGG_10.5.0/com.ibm.db2.luw.apdv.embed.doc/doc/c0061906.html.

[39] IBM Informix. [n. d.]. Understanding SQL insert cursors. https://www.ibm.com/support/knowledgecenter/en/SSBJG3_2.5.0/com.ibm.gen_busug.doc/c_fgl_InsertCursors_002.htm

[40] 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)*. 247–254. https://doi.org/10.1145/3087556.3087583

[41] 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)*. 301–315.

[42] 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)*. 235–246. http://www.vldb.org/conf/1999/P23.pdf

[43] Bradley C. Kuszmaul. 2009. How Fractal Trees Work. In *OpenSQL Camp*. Portland, OR, USA. An expanded version was presented at the MySQL User Conference, Santa Clara, CA, USA April 2010.

[44] Amanda McPherson. [n. d.]. A Conversation with Chris Mason on Btrfs: the next generation file system for Linux. https://www.linuxfoundation.org/blog/2009/06/a-conversation-with-chris-mason-on-btrfs/, Last Accessed Sep. 26, 2018.

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

[46] NuDB. 2016. NuDB: A fast key/value insert-only database for SSD drives in C++11. https://github.com/vinniefalco/NuDB.

[47] Patrick O'Neil, Edward Cheng, Dieter Gawlic, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385. https://doi.org/10.1007/s002360050048

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

[49] Oracle Corporation. [n. d.]. MySQL 5.5 Reference Manual. https://dev.mysql.com/doc/refman/5.5/en/innodb-file-space.html, Last Accessed Sep. 26, 2018.

[50] Oracle Corporation. 2015. Oracle BerkeleyDB Reference Guide. http://sepp.oetiker.ch/subversion-1.5.4-rp/ref/am_conf/pagesize.html, Last Accessed August 12, 2015.

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

[52] 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 2016 Annual Technical Conference (USENIX ATC)*. 537–550.

[53] John Paul. [n. d.]. Teradata Thoughts. http://teradata-thoughts.blogspot.com/2013/10/teradata-13-vs-teradata-14_20.html, Last Accessed Sep. 26, 2018.

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

[55] 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)*. 497–514. https://doi.org/10.1145/3132747.3132765

[56] 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

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

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

[59] 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.

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

[61] Tokutek, Inc. [n. d.]. TokuMX—MongoDB Performance Engine. https://www.percona.com/software/mongo-database/percona-tokumx, Last Accessed Sep. 26, 2018.

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

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

[64] Jeffrey Scott Vitter. 2001. External memory algorithms and data structures: Dealing with massive data. *ACM Computing surveys (CsUR)* 33, 2 (2001), 209–271.

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

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

[67] 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)*. 1–14.

[68] 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. *TOS* 13, 1 (2017), 3:1–3:26.

[69] 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)*. 123–138.