UNDERSTANDING DICTIONARIES AT THE INTERSECTION OF THEORY AND PRACTICE

by

ALEXANDER CONWAY

A dissertation submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Martín Farach-Colton

and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

October, 2020

ABSTRACT OF THE DISSERTATION

Understanding Dictionaries at the Intersection of Theory and Practice

By Alexander Conway

Dissertation Director:

Martín Farach-Colton

Dictionaries are fundamental data structures that associate values to a set of keys. They form the foundation of most storage systems, and are key to the performance of many algorithms.

Dictionaries are well studied from an algorithmic perspective, and many constructions of optimal dictionaries are known. However, these are rarely used in practice, and the ubiquitious implementation, the log-structured merge tree, is theoretically suboptimal.

This work studies a collection of dictionary problems, each of which lies somewhere between theory and practice. These problems take advantage of the flow of ideas back and forth betwen them, yielding interesting and surprising results, both where innovations and ideas in systems have influenced theoretical data structures, and also where those data structures form the foundation for new highly performant systems.

Acknowledgements

I can't say enough to thank my advisor, Martín Farach-Colton. Under his guidance, I have found countless cool things to learn, awesome problems to work on, and great people to work with. More than that, him and his family have been great friends to me, have helped me when I was struggling and helped me grow into a better person. Most of all, these five years have been so much fun.

I have had the pleasure of working with a truly amazing group of collaborators. It has been invaluable to discuss systems with Don Porter and theory with Michael Bender. I have loved working with Ainesh Bakshi, Yizheng Jiao, William Jannen, Nirjhar Mukherjee, Prashant Pandey, Guido Tagliavini Ponce, Meng-Tsung Tsai, Jun Yuan and Yang Zhan. Working together with everyone has been one of my favorite things in graduate school, and is one of the reasons I've been successful and stuck it through.

I would especially like to thank Rob Johnson, with whom I interned at VMware Research Group for 2 years, and with whose help I have secured the next phase of my research career. I am greatly looking forward to working there together in the future.

I would like to thank everyone else whom I've had the pleasure of working with at VMware, specifically Vijay Chidambaram, Kapil Chowksey, Yoni Fogel, Abhishek Gupta, Srinath Premachandran and Amy Tai. Also my time there would not have been the same without my good friends, Reto Achermann, Amogh Akshintala and Soujanya Ponnapalli.

I would also like to thank my committee members for their time and support.

I would like to thank my family for their support. My mother has always pushed me to work hard and live up to my potential, and especially to pursue hard and interesting research. My brother, Oliver, has spent many hours chatting with me about different

problems. And I would also like to thank my mother-in-law, Zsuzsanna, for encouraging and supporting me (not to mention letting me crash with her near Rutgers!).

Finally, I would like to thank my partner, Hanna, who has been there for me for the last 15 years. She has put up with me when I've struggled and has always been there to stand me up on my feet again. I know I could not have done this without her, and I hope that I can support her on her upcoming academic journey as well as she has on mine.

Table of Contents

Chapter 1

Introduction

Dictionaries are fundamental data structures that map a set of keys to values. A dictionary generally must support insertion and lookup, but also optionally delete, update, successor and scan. A systems that implements a dictionary is called a key-value store. Dictionaries lie at the heart of most storage systems and also play an essential role in many algorithms.

The work presented here examines problems related to the theory and implementation of high-performance external memory dictionaries.

Since the invention of the B-tree 50 years ago, the theory of external memory dictionaries and the implementation of key-value stores has progressed to some extent independently and in parallel. Currently, most external memory key-value stores used in practice are based on log-structured merge trees (LSMs), which are suboptimal from a theoretical standpoint.

From a theoretical perspective, there are many dictionary data structures which are optimal under different models. In the comparison DAM model, where the only operations permitted on key are key comparisons, the B^ε-tree was the first known optimal dictionary, and it was followed by an LSM derivative, the COLA, and several other data structures. In the broader DAM model, more general operations—in particular hashing—are permitted, which improves the performance of optimal data structures. In this model, the only previously known optimal dictionary is the external memory hash table of Iacono and Pătraşcu. Of these optimal data structures, only the B^ε-tree has been used successfully in a handful of systems.

1.1 Key Differences between Theory and Practice

Although a proper exposition of this schism between the data structures used in practice and those which are optimal in theory is beyond the scope of this work, it is helpful to examine some of the reasons here in order to motivate and contextualize the results.

Models

One of the difficulties in translating theoretical results into systems is understanding the strengths and limitations of the models used. The disk access model (DAM) has long been the standard model for external memory algorithms, but while it offers a reasonable approximation of the perforamnce of hard drives, it can distort performance by a factor of 2 compared to more realistic models, such as the affine model. These sorts of constant factors are irrelevant in theory, but meaningful in practice.

The assumptions used also matter. For example, it is often the case that a system may have a cache which is 25–50% the size of the dataset, whereas many theoretical results, such as the lower bounds alluded to above, assume that the cache is much smaller, typically $M = o(N/\log N)$. Thus, at first glance, these results may not directly apply.

Hardware

As technology changes and new storage hardware is developed, new models and design principles are required to understand and leverage their performance. As the premium storage technology has evolved from hard drive to solid state device to block-addressable non-volatile memory, much of the external memory literature has remained relatively constant. Interestingly enough, the performance of SSDs and NVMe devices is perhaps best captured by a classical model: the parallel disk access model (PDAM). As a result, in many settings, variations of classical external memory data structures perform well on these devices.

Filters

A filter is a probabilistic set-membership data structure with one-sided error. The first and most well-known example is the Bloom filter, but many other variants have been proposed and implemented, such as the cuckoo filter and the quotient filter. Filters are commonly used to optimize the lookup performance of LSMs and under some conditions can reduce the number of IOs per point lookup to 1.

Filters are commonly only used in memory (although cuckoo and quotient filters have good performance characteristics in external memory as well), and across the dataset consume at least $\Omega N/\log N$ space—implying that $M = \Omega N/\log N$. This part of the parameter space is less theoretically interesting, because the principle lower bounds do not hold. However, as a result, the application of filters to common data structures which asymptotically dominate LSMs, such as COLAs and B^ε-trees, has been overlooked, and these data structures have received much less exposure to the systems community.

Thus filters have steered systems both towards LSMs, by "fixing" their lookups, and towards hardware configurations with a relatively large amount of cache, to enable their use.

1.2 This Work

This work studies a collection of dictionary problems, each of which lies somewhere between theory and practice. The themes above—models and their limitations, changing hardware and the use of filters—frequently recur. However, rather than dividing the theoretic and practical study of dictionaries, these themes reflect the flow of ideas back and forth between them. This yields interesting and surprising results, both where innovations and ideas in systems have influenced theoretical data structures, and also where those data structures form the foundation for new highly performant systems.

Chapter 2: Optimal Hashing in External Memory

This is evident in *Optimal Hashing in External Memory*, in which LSMs—commonly used in practice, but theoretically suboptimal—are modified using a new type of filter, the routing filter, to create a simple optimal external memory dictionary, the BOA. Pushing these ideas further yields the BOT, which is optimal over larger parameter ranges, and the COBOT, which is the first optimal cache-oblivious dictionary.

This content was presented at ICALP 2018 by myself, Martín Farach Colton and Phillip Shillane.

Chapter 3: SplinterDB

This chapter introduces SplinterDB: a highly concurrent key-value store designed to perform on NVMe. SplinterDB takes the ideas underlying BOAs and BOTs, including routing filters, and implements them within a B^ε-tree. SplinterDB outperforms RocksDB, a state-of-the-art key-value store, by 2–8× on insertions and 1.2–2× on point lookups.

This content will appear at ATC 2020 and is by myself, Abhishek Gupta, Vijay Chidambaram, Martín Farach-Colton, Richard Spillane, Amy Tai and Rob Johnson.

Chapter 4: File System Aging

File system aging is thought to be a solved problem: while older file systems are known to age, modern file systems (and devices) are believed to only age under adversarial workloads. However, models for block allocation suggest that the approaches taken by most file systems should lead to aging under a broad range of workloads.

In this work, we present an aging tool based on the version control system, git, which replays development histories spanning years of use. Using this tool, we show that 5 popular file systems suffered catastrophic aging on hard drives and substantially aging on SSDs.

On the other hand, BetrFS, a file system which uses B^ε-trees to manage its on-disk data, should not age because it algorithmically moves stored data to maintain locality.

This is borne out using the git aging tool: BetrFS does not age at all under the same workload.

This content was presented at FAST 2017 by myself, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A Bender, Rob Johnson, Bradley C Kuszmaul, Donald E Porter and Martín Farach-Colton.

Chapter 5: Optimal Ball Recycling

A popular optimization technique for B-tree-based databases is the use of an insertion buffer. This is an in-memory buffer, which stores insertions as they arrive with the hope of batching them together before they are written to the leaves of the B-tree. This chapter models and analyzes insertion buffers (as well as the related update buffers) and provides a tight upper bound of the performance improvement that can be obtained. This result undermines the common assumption that large caches can be used to fix performance problems in storage systems.

This content appeared at SODA 2018 by Michael Bender, Jake Christensen, myself, Martin Farach-Colton, Rob Johnson and Meng-Tsung Tsai.

Chapter 2

Optimal Hashing in External Memory

2.1 Introduction

Dictionaries are among the most heavily used data structures. A dictionary maintains a collection of key-value pairs $\mathcal{S} \subseteq mathcalU \times mathcalV$, under operations[1] INSERT$(x, v, \mathcal{S})$, DELETE$(x, mathcalS)$, and QUERY$(x, \mathcal{S})$, which returns the value corresponding to x when $x \in S$. When data fits in memory, there are many solutions to the dictionary problem.

When data is too large to fit in memory, comparison-based dictionaries can be quite varied. They include the B$^\varepsilon$-tree [29], the write-optimized skip list [20], and the cache-optimized look-ahead array (COLA) [15, 18, 13]. These are optimal in the ***external-memory comparison model*** in that they match the bound established by Brodal and Fagerberg [29] who showed that for any dictionary in this model, if insertions can be performed in $O\left(\frac{\lambda \log_\lambda N}{B}\right)$ amortized IOs, then there exists a query that requires at least $\Omega(\log_\lambda N)$ IOs, where N is the number of items that can be stored in the data structure, B is the size of a memory transfer, and λ is a tuning parameter. In the following M will be the size of memory, and $B = \Omega(\log n)$. This trade off has since been extended in several ways [17, 4].

Iacono and Pătraşcu showed that in the DAM model, in which operations beyond comparisons are allowed on keys, a better tradeoff exists:

Theorem 1 ([57])**.** *If insertions into an external memory dictionary can be performed in $O(\lambda/B)$ amortized IOs, then queries require an expected $\Omega(\log_\lambda N)$ IOs.*

[1] We do not consider dictionaries that also support the SUCC$(x, \mathcal{S})$ and PRED$(x, \mathcal{S})$. SUCC$(x, mathcalS)$ returns $\min\{y | y > x \wedge y \in \mathcal{S}\}$ and PRED$(x, mathcalS)$ is defined symmetrically.

They further describe an external-memory hashing algorithm, which we refer to here as the ***IP hash table***, that performs insertions in $O\left(\frac{1}{B}\left(\lambda + \log_{\frac{M}{B}} N + \log\log N\right)\right)$ IOs and queries in $O(\log_\lambda N)$ IOs w.h.p. Therefore, for $\lambda = \Omega\left(\log_{M/B} N + \log\log N\right)$, the IP hash table meets the tradeoff curve of theorem 1 and is thus optimal.

In dictionaries that do not support successors and predecessors, we can assume that keys are hashed, that is, that they are uniformly distributed and satisfy some independence properties. The IP hash table and the following results hash all keys before insertion and query in the dictionary by a $\Theta(\log N)$-independent hash function.

The base result of this paper is a simple external-memory hashing scheme, the ***Bundle of Arrays Hash Table*** (BOA), that meets the optimal theorem 1 trade off curve for large enough λ. Specifically, we show:

Theorem 2. *A BOA supports N insertions and deletions with amortized per entry cost of $O\left(\left(\lambda + \log_{\frac{M}{B}} N + \log_\lambda N\right)/B\right)$ IOs, for any $\lambda > 1$. A query for a key K costs $O(D_K \log_\lambda N)$ IOs w.h.p., where D_K is the number of times K has been inserted or deleted.*

Thus BOAs are optimal for $\lambda = \Omega(\log_{\frac{M}{B}} N + \log_\lambda N)$. They are readily modified to provide several variations, notably the ***Bundle of Trees Hash Table*** (BOT). BOTs are optimal for the same range of λ as the IP hash table:

Theorem 3. *A BOT supports N insertions and deletions with amortized per entry cost of $O\left(\left(\lambda + \log_{\frac{M}{B}} N + \log\log M\right)/B\right)$ IOs for any $\lambda > 1$. A query for a key K costs $O(D_K \log_\lambda N)$ IOs w.h.p., where D_K is the number of times K has been inserted or deleted.*

We further introduce the first cache-oblivious hash table, the ***Cache-Oblivious Bundle of Trees Hash Table*** (COBOT), which matches the IO performance of BOTs and IP hash tables.

The BOT can also be adapted to models in which disk reads and writes incur different costs. The β-asymmetric BOT adjusts the merging schedule of a regular BOT to trade some writes for more reads.

Theorem 4. *A β-asymmetric BOT supports N insertions and deletions with amortized per entry cost of $O\left(\frac{1}{B}\left(\lambda + \frac{1}{\beta}\log_\lambda N\right)\right)$ writes and $O\left(\frac{1}{B}\left(\lambda+\beta\right)\right)$ reads for any $\lambda > 1$ and $\beta \leq \left\lfloor \log_\lambda \frac{M}{B \log_\lambda N} \right\rfloor$. A query for a key K performs $O(D_K \log_\lambda N)$ reads, where D_K is the number of times K has been inserted or deleted.*

2.2 Preliminaries

Fingerprints and Hashing

In order to achieve our bounds, we need $\Theta(\log N)$-wise independent hash functions, which, once again matches IP hash tables. We note that a k-wise independent hash function is also k-wise independent on individual bits. Furthermore, the following Chernoff-type bound holds:

Lemma 1 ([96])**.** *Let $X_1, X_2, \ldots, X_N$ be $\lceil \mu\delta \rceil$-wise independent binary random variables, $X = \sum_{i=1}^{N} X_i$ and $\mu = \mathrm{E}[X]$. Then $\mathrm{P}\left(X > \mu\delta\right) = O\left(1/\delta^{\mu\delta}\right)$, for sufficiently large δ.*

In the following, we use ***fingerprint*** to refer to any key that has been hashed using a $\Theta(\log N)$-wise independent hash function. Such hash functions have a compact representation and can be specified using $\Theta(\log N)$ words. The universe that is hashed into is assumed to have size $\Theta(N^k)$ for $k \geq 2$. We ignore collisions, but these can be handled as in [57].

For a fingerprint K, it will be convenient to interpret the bits of K as a string of $\log \lambda$ (where lambda is a given parameter) bit characters, $K = K_0 K_1 K_2 \cdots$.

Delta Encoding

We will frequently encounter sorted lists of fingerprint prefixes (possibly with duplicates), together with some data about each. When the size of the list is dense in the space of prefixes, we can compress it using ***delta encoding***, where the difference between prefixes is stored rather than the prefixes themselves.

Lemma 2. *A list of delta-encoded prefixes with density D, that is there are D prefixes in the list for every possible prefix, requires $O(-\log_\lambda D)$ characters per prefix.*

Proof. The average difference between consecutive prefixes is $1/D$. Because logarithms are convex, the average number of characters required to represent this difference is therefore $O(-\log_\lambda D)$. □

Log-structured Merge Trees

Log-structured merge trees (LSMs) are (a family of) external-memory dictionary data structures. They come in two varieties: ***level-tiered LSMs*** (LT-LSMs) and ***size-tiered LSMs*** (ST-LSMs). Both kinds are suboptimal in that they do not meet the optimal insertion-query tradeoff [29], although the COLA [15] is an optimal variant of the LT-LSMs.

An LSM consists of sets of either B-trees or sorted arrays called ***runs***. In this paper, we describe them in terms of runs, since we use runs below.

An LT-LSM consists of a cascade of levels, where each level consists of at most one run. Each level has a ***capacity*** that is λ times greater than the level below it, where λ is called the ***growth factor***.[2] When a level reaches capacity, it is merged into the next level (perhaps causing a merge cascade). The amortized IO cost for insertions is small because sequential merging is fast, although each item will participate in $\lambda/2$ merges on average. The IO cost for a query is high because a query must be performed independently on each of $O(\log_\lambda N)$ levels (although Bloom filters [24, 18] are sometimes used to mitigate this cost).

An ST-LSM further improves insertion IOs at the expense of queries. Each level contains fewer than λ runs. Every run on a given level has the same size, which is λ times larger than the runs on the level beneath it. When λ runs are present at a level, they are merged into one run and placed at the next level. There are therefore $O(\log_\lambda N)$ levels. Insertions are faster than in LT-LSMs because each item is only

[2]Sometimes this and related structures are analyzed with a growth factor of B^ϵ. The two are equivalent. We use λ rather than ϵ as the tuning parameter for consistency with the external-memory hashing literature.

merged once on each level. Queries are slower because each query must be performed $O(\lambda)$ times at each level.

In LSMs, deletions can be implemented by the use of ***upsert messages*** [48, 60], which are a type of insertion with a message that indicates that the key has been deleted. A query for a key K then fetches all the matching key-value pairs and if the last one (temporally) is a deletion upsert, it returns false. To this end, the merges must maintain the temporal order of key-value pairs with the same key. Because a query for a key K must fetch every instance of K, the cost of a query is proportional to the number of times the key has been inserted and deleted, which we refer to as the ***duplication count***, D_K of K. When $N/2$ deletions have been made, the structure is rebuilt to reclaim space. In what follows, deletions will be implemented using the same mechanism.

2.3 Bundle of Arrays Hashing

A ***Bundle of Arrays Hash Table*** (BOA) is an external-memory dictionary based on ST-LSMs. In this section, we describe a simple version which is optimal in the sense of theorem 1, but where the query cost meets the bound only in expectation, not w.h.p. In section 2.4, we give a version that satisfies theorem 2.

As a first step, we show that runs with uniformly distributed, $\Theta(\log N)$-wise independent fingerprints can be searched more quickly than in an ST-LSMs.

Lemma 3. *Let A be a sorted array of N uniformly distributed $\Theta(\log N)$-wise independent keys in the range $[0, K)$, and assume $B = \Omega(\log N)$. Then A can be written to external memory using $O(N)$ space and $O(N/B)$ IOs so that membership in A can be determined in $O(1)$ IOs with high probability.*

Proof. First note that, by lemma 1 and Bonferroni's inequality, if N balls are thrown into $\Theta(N/\log N)$ bins uniformly and $\Theta(\log N)$-wise independently, then every bin has $\Theta(\log N)$ balls with high probability.

Divide the range of keys into N/B uniformly sized buckets; that is, bucket i contains keys in the range $[(i-1)KB/N, iKB/N)$. Because the keys in A are distributed

uniformly, and $B = \Omega(\log N)$, every bucket contains $\Theta(B)$ keys with high probability. Let F be the number of items in the fullest bucket, and write the keys in each bucket to disk in order using F space for each. Because $F = \Theta(B)$, this takes the desired space and IOs.

Now, to find a key, compute which bucket it belongs to. A constant number of IOs will fetch that bucket, whose address is known because all buckets have the same size. □

Corollary 1. *If an ST-LSM contains uniformly distributed and $\Theta(\log N)$-wise independent fingerprints and has growth factor λ, then a query for K can be performed in $O(D_K \lambda \log_\lambda N)$ IOs by writing the levels as in lemma 3. The insertion/deletion cost is unchanged: $O\left(\frac{1}{B}\left(\log_\lambda N + \log_{\frac{M}{B}} N\right)\right)$ amortized IOs.*

While the query performance improves by a factor of $\log N$, the ST-LSM is still off of the optimal tradeoff curve of theorem 1. In particular, queries can be at least exponentially slower than optimal. The BOA uses additional structure in order to reduce this query cost.

2.3.1 Routing Filters

The main result of this section is an auxiliary data structure, the ***routing filter***, that improves the query cost of an ST-LSM by a factor of λ by further exploiting the log-wise independence of fingerprints. Combining these routing filters with fast interpolation search will yield the BOA, a hashing data structure that is optimal for large enough λ.

The purpose of the routing filter is to indicate probabilistically, at each level, which run contains the fingerprint we are looking for. Each level will have its own routing filter, defined as follows. For each level ℓ, let h_ℓ be some number, to be specified below. Let $P_\ell(K)$ be the prefix consisting of the first h_ℓ characters of K. The routing filter F_ℓ for level ℓ is a λ^{h_ℓ}-character array, where $F_\ell[i] = j$ if the jth run $R_{\ell,j}$ contains a fingerprint K such that the $P_\ell(K) = i$, and no later run $R_{\ell,j'}$ (i.e. with $j' > j$) contains such a fingerprint.

We also modify each run $R_{\ell,j}$ during the merge so that each fingerprint-value pair

contains a ***previous field*** of 1 additional character used to specify the previous run containing a fingerprint with the same prefix, or j, to indicate no such run exists. Thus these fingerprint-value pairs now form a singly linked list whose fingerprints share the same prefix, and the routing filter points to the run containing the head.

During a query for a fingerprint K, first $F_\ell[P_\ell(K)]$ is checked to find the latest run containing a fingerprint with a matching prefix. Once that fingerprint-value pair is found, its previous field indicates the next run which needs to be checked and so on until all fingerprints with matching prefix in the level are found. Each fingerprint $K' \neq K$ that matches K's prefix is a false positive.

Such routing filters induce a space/cost tradeoff. The greater h_ℓ is, the more space the table takes but the less likely it is that many runs will have false positives. The rest of this section shows that when $h_\ell = \log_\lambda B + \ell$, in other words, when prefixes grow by a character per level, the BOA lies on the optimal tradeoff curve of theorem 1.

Define β, the ***routing table ratio***, to be the ratio of the number of buckets in the routing filter to the size of a run. The number of entries in a run on level ℓ is $B\lambda^{\ell-1}$, so $\beta = \lambda^{h_\ell}/B\lambda^{\ell-1}$. We first analyze the per-level insertion/deletion cost, and then we compute the expected number of false positives in order to analyze the overall query cost.

Lemma 4. *For a BOA with growth factor λ and routing table ratio β, merging a level incurs $\Theta\left(\frac{1}{B}\left(1 + \log_{\frac{M}{B}} \lambda + \beta \log_N \lambda\right)\right)$ IOs per fingerprint.*

Proof. Merging a level requires merging its runs as well as updating the next level's routing filter. Merging λ sorted arrays takes $\Theta\left(\frac{1}{B}\left(1 + \log_{M/B} \lambda\right)\right)$ IOs per fingerprint.

The routing filter is updated by iterating through it and the new run sequentially. For each fingerprint K appearing in the run, $F_{\ell+1}[P_{h_{\ell+1}}(K)]$ is copied to the previous field in the run, and $F_{\ell+1}[P_{h_{\ell+1}}(K)]$ is set to the number of the current run. Each entry in the routing filter is a character, and the routing filter has β entries for each new fingerprint. Thus, it requires $\Theta\left(\frac{\beta}{B} \log_N \lambda\right)$ IOs per fingerprint to update sequentially.

□

Lemma 5. *For a BOA with growth factor λ and routing table ratio β, querying a fingerprint K on a given level incurs at most $\frac{\lambda}{\beta}$ false positives in expectation.*

Proof. Given some enumeration of the fingerprints in level ℓ, which are not equal to K, denote the ith such fingerprint by K_i. Some of these may be duplicates. Let X_i be the indicator random variable, which is 1 if $P_\ell(K) = P_\ell(K_i)$ and 0 otherwise. K and K_i are uniformly distributed and their bits are pairwise independent. Thus $\mathrm{E}[X_i] \leq \frac{1}{\lambda^{h_\ell}}$. The expected number of fingerprints (excluding K) in the level with prefix $P_\ell(K)$ is thus at most $\sum_{i=1}^{B\lambda^\ell} \mathrm{E}[X_i] \leq \frac{B\lambda^\ell}{\lambda^{h_\ell}} = \frac{\lambda}{\beta}$. □

Lemma 6. *A BOA with growth factor λ and routing table ratio β has insertion/deletion cost $O\left(\frac{1}{B}\left(\beta + \log_{\frac{M}{B}} N + \log_\lambda N\right)\right)$. A query for fingerprint K has expected cost $O\left(\frac{\lambda}{\beta} D_K \log_\lambda N\right)$, where D_K is the duplication count of K.*

Proof. Because a BOA has $\log_\lambda N$ levels, the insertion cost follows from lemma 4.

To query for a fingerprint K, the routing filter on each level is checked, which incurs $O(\log_\lambda N)$ IOs. These routing filters return a collection of runs which contain up to D_K true positives and an expected $O\left(\frac{\lambda}{\beta} \log_\lambda N\right)$ false positives, by lemma 5. By lemma 3, each run can be checked in $O(1)$ IOs. □

So for a fixed λ, there is no advantage to choosing $\beta = \omega(\lambda)$. On the other hand, $\beta = o(\lambda)$ is suboptimal, because then choosing $\beta' = \lambda' = \beta$ changes a linear factor in the query cost to a logarithmic one. Therefore, it is optimal to choose $\beta = \Theta(\lambda)$, and in what follows we will fix $\beta = \lambda$. Thus,

Lemma 7. *A BOA supports N insertions and deletions with amortized per entry cost of $O\left(\left(\lambda + \log_{\frac{M}{B}} N + \log_\lambda N\right)/B\right)$ IOs, for any $\lambda > 1$. A query for a key K costs $O(D_K \log_\lambda N)$ IOs in expectation, where D_K is the duplication count of K.*

2.4 Refined Bundle of Arrays Hashing

In order to obtain high probability bounds for a BOA, we need a stronger guarantee on the number of false positives. This is achieved by including an additional character,

the ***check character*** from each fingerprint in the routing filter, which is also checked during queries and thus eliminates most false positives. To support this, we will need to refine the routing filter so it can maintain check characters even when there are collisions.

The ith check character $C_i(K)$ of a fingerprint K is the ith character from the end of the string representation of K. As described in section 2.2, we assume that the fingerprints are taken from a universe of size at least N^2 so that the check characters do not overlap with the characters used in the prefixes of the routing filters, and by $\Theta(\log N)$-wise independence, the check characters of $O(1)$ fingerprints are independent. Now each fingerprint in the filter has a check character and a array pointer, and we refer to this data as the ***sketch*** of the fingerprint.

When level i of the BOA is queried for a fingerprint K, the refined routing filter (described below) returns a list of sketches, one for each fingerprint in the level with prefix $P_i(K)$. The array indicated in the sketch is only checked if the check character matches $C_i(K)$, which reduces the number of false positives by a factor of λ.

Refined Routing Filter

The routing filter described in section 2.3.1 handles prefix collisions by returning only the last run containing the queried fingerprint and then chaining in the runs. Whereas, to support check characters we need to return a list instead, while having the same performance guarantees.

The idea behind the refined routing filter is to keep the prefix-sketch pairs in a sorted list and use a hash table on prefixes to point queries to the appropriate place. Each pointer may require as many as $\Omega(\log N)$ bits, and we require the routing filter to have $O(1)$ characters per fingerprint. Therefore the hash table must use shorter prefixes so as to reduce the number of buckets and thus reduce its footprint. In particular, it uses prefixes which are $\log_\lambda \log_\lambda N$ characters shorter, which we refer to as ***pivot prefixes***.

The list delta encodes the prefix for each fingerprint K, together with its sketch. In addition, the first entry following each pivot prefix contains the full prefix, rather than just the difference. Otherwise, when the hash table routes a query to that place in the

list, the full prefix wouldn't be immediately computable.

Lemma 8. *A refined routing filter can be updated using $O\left(\frac{\lambda \log \lambda}{B \log N}\right)$ IOs per new entry, and performs lookups in $O(D_K^*)$ IOs w.h.p., where D_K^* is the number of times K appears in the level.*

Proof. We prove first the update bound and then the query bound.

Let C be the capacity of the level. There are at most $\frac{C}{\log_\lambda N}$ pivot prefixes. For each pivot prefix, the hash table stores the bit position in a list with at most C entries, where $C \leq N$. Each entry is at most $\log N$ bits, so this position can be written using $O(\log N)$ bits.

For each fingerprint in the node, the list contains $O(1)$ characters by lemma 2, or $O(\log \lambda)$ bits. Additionally, each pivot prefix has to an initial entry of length $O(\log N)$ bits, so the list all together uses $O(C \log \lambda + \frac{C}{\log_\lambda N} \cdot \log N) = O(C \log \lambda)$ bits.

When the refined routing filter is updated, the old version is read sequentially and the new version is written out sequentially. C/λ fingerprints are added at a time, so this incurs $O\left(\frac{\lambda \log \lambda}{B \log N}\right)$ IOs per entry.

During a query, the pivot bit string of a fingerprint and its successor are accessed from the hash table in $O(1)$ IOs. This returns the beginning and ending bit positions in the list. Because the fingerprints are distributed uniformly and are pairwise independent to K, there are $O(\log_\lambda N + D_K^*)$ fingerprints matching the pivot prefix in expectation. From lemma 1 with $\delta = \log \lambda$, there are $O(\log N + D_K^*)$ fingerprints matching the pivot prefix w.h.p. The encoding of each fingerprint is less than a word, and $B = \Omega(\log N)$ by assumption, so this is $O(D_K^*)$ IOs. □

BOA Performance

We now can show:

Theorem 2. *A BOA supports N insertions and deletions with amortized per entry cost of $O\left(\left(\lambda + \log_{\frac{M}{B}} N + \log_\lambda N\right)/B\right)$ IOs, for any $\lambda > 1$. A query for a key K costs $O(D_K \log_\lambda N)$ IOs w.h.p., where D_K is the number of times K has been inserted or deleted.*

Proof. The insertion/deletion cost is given by lemma 8 and lemma 7.

During a query for a fingerprint K, the expected number of false positives on level i (fingerprints which match the prefix $P_i(K)$ and check character $C_i(K)$ but are not K) is $O\left(\frac{1}{\lambda}\right)$. Thus, the number of false positives across levels is $O\left(\frac{\log_\lambda N}{\lambda}\right)$, so by lemma 1, the number of false positives is $O\left(\log_\lambda N\right)$ w.h.p. □

Thus, a BOA is optimal for large enough λ:

Corollary 2. *Let $\mathcal{B}$ be a BOA with growth factor λ containing N entries. If $\lambda = \Omega\left(\log_{\frac{M}{B}} N + \frac{\log N}{\log\log N}\right)$, then $\mathcal{B}$ is an optimal unsorted dictionary.*

2.5 Bundle of Trees Hashing

In order for a BOA to be an optimal dictionary, its growth factor λ must be $\Omega(\log N/\log\log N)$. Otherwise, the cost of insertion is dominated by the cost of merging, which is slow because it effectively sorts the fingerprints using a λ-ary merge sort. In this section, we present the ***Bundle of Trees Hash Table*** (BOT), which is a BOA-like structure. A BOT stores the fingerprints in a log in the order in which they arrive. Each level of the BOT is like a level of a BOA, where the bundle of arrays on each level is replaced by a search structure on the log (the ***routing tree***) and a data structure needed to merge routing trees (the ***character queue***). The character queue performs a delayed sort on the characters needed at each level, thus increasing the arity of the sort and decreasing the IOs.

A BOT has $s = \lceil \log_\lambda N/B \rceil$ levels, each of which contains a routing tree. The root of the routing tree has degree less than λ and all internal nodes have degree λ. Each node of a routing tree contains a routing filter. As in section 2.4, each routing filter takes as input a fingerprint K and outputs a list of sketches corresponding to fingerprints with the same prefix as K. Each sketch consists of a pointer to a child, a check character and some auxiliary information discussed below.

Each leaf points to a block of B fingerprints in the log. The deepest level s uses a height-s tree to index the beginning of the fingerprint log, the next level then indexes the next section, and so forth, as shown in fig. 2.1. Insertions and deletions (as upsert

messages) are appended to the log until they form a block, at which point they are added to the tree in the 1st level of the BOT.

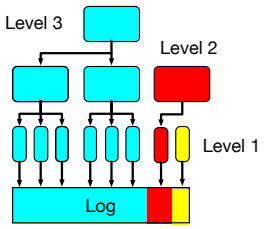


Figure 2.1: The routing trees in a 3 level BOT. The trees cover contiguous portions of the log. The highest level covers the beginning of the log, the next level the beginning of the remainder of the log, and so on.

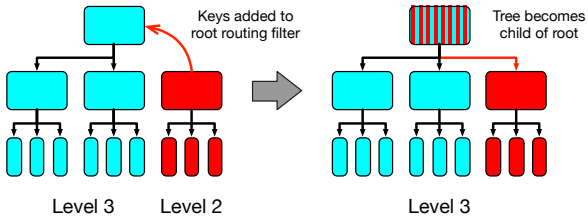


Figure 2.2: When the routing tree on level i fills, it is merged into the routing tree on level $i+1$. The now-full routing tree from level $i+1$ becomes a child of the root on level $i+1$. Its fingerprints are added to the root routing filter. Note that the tree is not moved.

When a level i in the BOT fills, its routing tree is merged into the routing tree of level $i+1$, thus increasing the degree of the target routing tree by 1 (and perhaps filling level $i+1$, which triggers a merge of level $i+1$ into $i+2$, and so on). The merge of level i into level $i+1$ consists of adding the prefix-sketch pairs of the fingerprints from level i to the routing filter of the root on level $i+1$. The child pointers of these pairs will point to the root of the formerly level-i routing tree, so it becomes a child of the root of the level $i+1$ routing tree, although it isn't moved or copied. See fig. 2.2. In this way, a BOT resembles an LT-LSM, as described in section 2.2.

In order to add a fingerprint K from level i to the root routing filter on level $i+1$, the prefix $P_{i+1}(K)$ must be known. However, the root routing filter on level i only stores the prefix $P_i(K)$ for each fingerprint K it contains, so that in particular, the last character of $P_{i+1}(K)$ is missing. As described in section 2.5.2, each level has a character queue, which provides this character, as well as the check characters, in order to merge the routing trees efficiently.

2.5.1 Queries in a BOT

A query to the BOT for a fingerprint K is performed independently at each level, beginning at the root of each routing tree. When a node is queried, its routing filter

returns a list of sketches. The sketches whose check characters match the queried fingerprint indicate to which children the query is passed. This process continues until the query reaches a block of the log, which is then searched in full. In this way queries are "routed" down the tree on each level to the part of log where the fingerprint and its associated value are. Note that as queries descend the routing tree, they may generate false positives which are likewise routed down towards the log.

In this section, we refine routing trees so that they offer two guarantees about false positives. The first is that at each level, the probability that a given false positive is not eliminated is at most $\frac{1}{\lambda}$. The second is that false positives can only be created in the root, so that as the query descends the tree, the number of false positives cannot increase.

During a query to a node of height h for a fingerprint K, the routing filter returns a list of sketches corresponding to fingerprints which match K's prefix. The query only proceeds on those children whose check characters also match the check character $C_h(K)$. Since the characters of the fingerprint are uniformly distributed and $\Theta(\log N)$-wise independent, the check character of each false positive matches with probability $\frac{1}{\lambda}$. Moreover, the characters of each level are non-overlapping, so for fingerprints K, K', the event that $V_h(K) = V_h(K')$ is independent of the event that $V_{h-1}(K) = V_{h-1}(K')$.

To prevent new false positives from being generated when a query passes from a parent to a child, the ***next character*** of each fingerprint is also kept in its sketch in the routing filter. For a fingerprint K in a node of height h, the next character is just the next character that follows the prefix, $P_h(K)$, so that its prefix in the parent, $P_{h+1}(K)$, can be obtained. A false positive in a child which is not in the parent will not match this next character and can be eliminated.

When there are multiple prefix-matching fingerprints in both a parent and its child, we would like to be able to align the lists returned by the routing filters so that known false positives in the parent (either from check or next characters) can be eliminated in the child. Otherwise the check character in the child of a known false positive in the parent may match the queried fingerprint, and therefore more than $\frac{1}{\lambda}$ of the false positives may survive. To this end, we require the routing filter to return the list of

sketches in the order their fingerprint-value pairs appear in the log. Then after the sketches in the child list whose next characters do not match the parent are eliminated, the remaining phrases will be in the same order as in the parent. In this way, known false positives can also be eliminated in the child.

Now we can show:

Lemma 9. *During a query to a routing tree, the following are true:*

1. *A false positive can only be generated in the root.*
2. *At each level, a given false positive survives with probability at most* $\frac{1}{\lambda}$.

Proof. Because of the next characters, false positives may only be created in the root of the routing tree. Each false positive in the root corresponds to a fingerprint K' in the level. At each node on the path to K''s location in the log, we use the ordering to determine which returned sketch corresponds to K', so that the false positive corresponding to K' is eliminated with probability $\frac{1}{\lambda}$. □

2.5.2 Character Queue

The purpose of the character queue is to store all the sketches of fingerprints contained in a level i that will be needed during a merge in the future. When level i is merged into level $i+1$, the character queue outputs a sorted list of the delta-encoded prefix-sketch pairs of all the fingerprints, which is used to update the root routing tree. The character queue is then merged into the character queue on level $i+1$.

The character queue effectively performs a merge sort on the sketches. If it were to merge all the sketches as soon as they are available, this would consist of λ-ary merges. In order to increase the arity of the merges, it defers merging sketches which are not needed immediately. The sketches are stored as a collection of ***series***, by which we mean a collection of sorted runs. Each series stores a continuous range of sketches $S_i(K), S_{i+1}(K), \ldots, S_{i+j}(K)$ for each fingerprint K, together with the prefix up to the first sketch, $P_{i-1}(K)$. These prefixes are delta encoded in their run. Thus the size of an entry is determined by the number of sketches in the range and the length of the prefix relative to the size of the run (by lemma 2).

The character queue tradeoff

We are faced with the following tradeoff. If the character queue merges a series frequently, the delta encoding is more efficient, which decreases the cost of the merging. However the arity is lower, which increases it. The character queue uses a merging schedule which balances this tradeoff and thus achieves optimal insertions.

The character queue merging schedule

The character queue on level i (here we consider blocks of the log to be level 0) contains the sketches $S_{i+1}(K), S_{i+2}(K), \ldots S_s(K)$ of each fingerprint K in the level. These characters are stored in a collection of series $\{\sigma_{j_q}\}$, where j_q is the smallest multiple of 2^q greater than i. Series σ_{j_q} contains the sketches $S_{j_q}(K), \ldots, S_{j_{q+1}-1}(K)$. Each series consists of a collection of sorted runs, each of which stores the delta encoded prefix of each fingerprint together with its sketches.

Initially, when a block of the log is written, all the series σ_{2^q} for $q = 1, 2, 3, \ldots$ are created. When level i fills, the runs in the series σ_{i+1} are merged, and the character queue outputs the delta encoded prefix-sketch pairs, $(P_{i+1}(K), S_{i+1}(K))$ to update the root routing filter on level $i+1$. If $2^{\rho(i+q)}$ is the greatest power of 2 dividing $i+1$ (ρ is sometimes referred to as the ***ruler function*** [107]), then σ_{i+1} also contains the next $2^{\rho(i+1)} - 1$ sketches of each fingerprint. These are batched and delta encoded to become runs in the series σ_{j_q} for $q = [0, \rho(i+1)]$. The runs in the remaining series of level i become runs of their respective series on level $i+1$.

Note that for the lower levels, some runs may be shorter than B due to the delta encoding. For a run in a series σ_q, this is handled by buffering them with the runs σ_q of higher levels and writing them out once they are of size B. Note that this requires $O(B \log\log N)$ memory.

This leads to the following merging pattern: σ_j batches $2^{\rho(j)}$ sketches, and has delta encoded prefixes of $2^{\rho(j)}$ characters on average, by lemma 2. Therefore,

Lemma 10. *A series σ_j in a character queue contains $O(2^{\rho(j)})$ characters per fingerprint.*

This leads to a merging schedule where the characters per item merged on the jth level is $O(2^{\rho(j)})$. Starting from 1 this is $1, 2, 1, 4, 1, 2, 1, 8, 1, 2, 1, 4, 1, 2, 1, 16, \ldots$, which resemble the tick marks of a ruler, hence the name ruler function.

We now analyze the cost of maintaining the character queues.

Lemma 11. *The total per-insertion/deletion cost to update the character queues in a BOT is* $\Theta\left(\frac{1}{B}\left(\log_{\frac{M}{B}} N + \log\log M\right)\right)$.

Proof. When σ_j is merged, $\lambda^{2\rho(j)}$ runs are merged, which has a cost of $O\left(\frac{2^{\rho(j)}}{B}\left\lceil \log_{M/B}\left(\lambda^{2^{\rho(j)}}\right)\right\rceil\right)$ characters per fingerprint.

There are $\log_\lambda \frac{N}{B} = O(\log_\lambda N)$ levels, so this leads to the following total cost in terms of characters:

$$\begin{aligned} O\left(\sum_{i=1}^{\log_\lambda N} 2^{\rho(j)}\left\lceil \log_{\frac{M}{B}}\left(\lambda^{2^{\rho(j)}}\right)\right\rceil\right) &= O\left(\sum_{k=0}^{\log\log_\lambda N} \frac{\log_\lambda N}{2^k}\cdot 2^k\left\lceil \log_{\frac{M}{B}}\left(\lambda^{2^k}\right)\right\rceil\right) \\ &= O\left(\log_\lambda N\left(\log\log M + \sum_{k=\log\log M}^{\log\log_\lambda N} 2^k \log_{\frac{M}{B}} \lambda\right)\right) \\ &= O\left(\log_\lambda N\left(\log\log M + \log_{\frac{M}{B}} N\right)\right), \end{aligned}$$

where the last equality is because the RHS sum is dominated by its last term. Because there are $\log_\lambda N$ characters in a word, and all reads and writes are performed sequentially in runs of size at lease B, the result follows. □

2.5.3 Performance of the BOT

We can now prove Theorem 3:

Theorem 3. *A BOT supports N insertions and deletions with amortized per entry cost of $O\left(\left(\lambda + \log_{\frac{M}{B}} N + \log\log M\right)/B\right)$ IOs for any $\lambda > 1$. A query for a key K costs $O(D_K \log_\lambda N)$ IOs w.h.p., where D_K is the number of times K has been inserted or deleted.*

Proof. By lemma 8, the cost of updating the routing filters is $O\left(\frac{\lambda}{B}\right)$, since there are $O(\log_\lambda N)$ levels. This, together with the cost of updating the character queues given

by lemma 11, is the insertion cost.

By lemma 9, a query for fingerprint K on level i incurs $O\left(\frac{1}{\lambda}\right)$ false positives in the root, and $O(1)$ nodes are accessed along each of their root-to-leaf paths. By lemma 8, each false positive thus incurs $O(D_K)$ IOs.

There are an expected $O\left(\frac{\log_\lambda N}{\lambda}\right)$ false positives across all levels, so, using lemma 1 with $\delta = \lambda$, $O(D_K \log_\lambda N)$ nodes are accessed due to false positives w.h.p. For each time K appears in the BOT, $O(\log_\lambda N)$ nodes are accessed on its root-to-leaf path. By lemma 8 the node accesses along each path incur $O(D_K \log_\lambda)$ IOs w.h.p., so accessing the nodes incurs $O(\log_\lambda N)$ IOs w.h.p.

A block of the log is scanned at most D_K times for true positives and also whenever a false positive from the level-i root survives i times. The expected number of such false positives for level i is $1/\lambda^i$, so the expected number across levels is $O\left(\frac{1}{\lambda}\right)$. Therefore by lemma 1, the number of blocks scanned is $O(D_K \log_\lambda N)$ w.h.p. □

Corollary 3. *Let $\mathcal{B}$ be a BOT with growth factor λ containing N entries. If $\lambda = \Omega\left(\log_{\frac{M}{B}} N + \log\log M\right)$, then $\mathcal{B}$ is an optimal dictionary.*

2.6 Cache-Oblivious BOTs

In this section, we show how to modify a BOT to be cache oblivious. We call the resulting structure a cache-oblivious hash tree (COBOT).

Much of the structure of the BOT translates directly into the cache-oblivious model. However, some changes are necessary. In particular, when the series of character queues are merged, this merge must be performed cache-obliviously using funnels [52], rather than with an (up to) M/B-way merge. Also, the log cannot be buffered into sections of size $O(B)$, and so instead they are buffered into sections of constant size, items are immediately added to the routing filter, and the extra IOs are eliminated by optimal caching.

When an insertion is made into a COBOT, its fingerprint-value pair is appended to the log, and it is immediately inserted into level 1. Thus, the leaves of the routing trees point to single entries in the log.

The series of the character queues must be placed more carefully as well. In particular the runs of series σ_j must be laid out back-to-back for all j (rather than just small j as in section 2.5.2), so that the caching algorithm can buffer them appropriately.

The series are merged using a ***partial funnelsort***. Funnelsort is a cache-oblivious sorting algorithm that makes use of K-funnels [52]. A K-funnel is a CO data structure that merges K sorted lists of total length N. We make use of the the following lemma.

Lemma 12 ([52])**.** *A K-funnel merges K sorted lists of total length $N \geq K^3$ in $O\left(\frac{N}{B}\log_{M/B}\frac{N}{B} + K + \frac{N}{B}\log_K\frac{N}{B}\right)$ IOs, provided the tall cache assumption that $M = \Omega(B^2)$ holds.*

The partial funnelsort used to merge K runs of a series with total length L (in words) performs a single merge with a K-funnel if $L \geq K^3$, and recursively merges the run in groups of $K^{1/3}$ runs otherwise.

Corollary 4. *A partial funnelsort merges K runs of total word length L in $O\left(\frac{L}{B}\log_{M/B}\frac{L}{B} + \frac{L}{B}\log_K\frac{L}{B}\right)$ IOs, provided the tall cache assumption that $M = \Omega(B^2)$ holds.*

Proof. The base case of the recursion occurs either when there is only 1 list remaining or the remaining lists fit in memory. In any other case of the recursion, since $L = \Omega(B^2)$ by the tall cache assumption, the K term in lemma 12 is dominated.

The recurrence is dominated by the cost of the funnel merges, which yields the result. □

Theorem 5. *If $M = \Omega(B^2)$, then a COBOT with N entries and growth factor λ has amortized insertion/deletion cost $\Theta\left(\frac{1}{B}\left(\lambda + \log\log M + \log_{M/B} N/B\right)\right)$. A query for key K has cost $\Theta\left(D_K \log_\lambda N\right)$, w.h.p., where D_K is the duplication count of K.*

Proof. We may assume that the caching algorithm sets aside enough memory that the last B items in the log, together with the subtree rooted at their least common ancestor, are cached. Thus the log is updated at a per-item cost of $O(1/B)$.

The proof of theorem 3 now carries over to the COBOT. The routing filters are updated the same way, and the cost of updating the character queues is unchanged, by corollary 4.

Queries are performed as in section 2.5.1, except that now the level 1 nodes cover $O(1)$ fingerprints, but the depth of the tree is unchanged, so the cost is the same. □

2.7 Asymmetric BOTs

In this section we describe the ***β-asymmetric Bundle of Arrays Hash table*** (Asymmetric BOT), which adapts BOTs to the asymmetric external memory model [23]. This model is similar to the regular external memory model of Aggarwal and Vitter, except that the cost of reading a block is 1, but where the cost of writing a block is $\omega > 1$.

The underlying idea is to not write character on most levels and instead read the necessary information from the queues of descendant nodes. This only requires reads, provided no more than M/B such queues are read at a time. Thus, every $\beta \leq \left\lfloor \log_\lambda \frac{M}{B \log_\lambda N} \right\rfloor$ levels, all the character queues must be merged and stored. We say level i is a ***queue level*** if i is divisible by β.

When the ith level root fills, where level i is not a queue level, it is added to level $i+1$. For each key K covered by the now-full orphan root, the high-order bits $\mathcal{H}_{i+1}(K)$, check characters $C_{i+1}(K)$ and next characters $D_{i+2}(K)$ are obtained by merging the character queues on the highest queue level below i, which is $\ell = i - (i \bmod \beta)$. Thus $L_\ell^{\ell+1}, \ldots, L_\ell^{i+1}$ are read; for each key K in sorted order this yields: $\mathcal{H}_\ell(K)$ together with the characters $D_{\ell+1}(K), \ldots, D_{i+2}(K)$, from which $\mathcal{H}_{i+1}(K)$ and $D_{i+2}(K)$ can be computed, as well as the check character $C_{i+1}(K)$. This computation occurs sequentially, so that the intermediate results need not be written out.

When the ith level root fills on a queue level, its character queues are created before it is added to level $i+1$. This involves computing $\mathcal{H}_i(K)$ in sorted order as above and then merging all the character queues $L_{i-\beta}^{i+1}, L_{i-\beta}^{i+2}, \ldots$ into new character queues $L_i^{i+1}, L_i^{i+2}, \ldots$. These merges are performed with a single λ^β-way merge. Then the

now-full level root can be added to the routing filter of the $(i+1)$st level root using the character queue L_i^{i+1}.

We refer to this modified BOT as an ***β-asymmetric BOT***.

Theorem 4. *A β-asymmetric BOT supports N insertions and deletions with amortized per entry cost of $O\left(\frac{1}{B}\left(\lambda + \frac{1}{\beta}\log_\lambda N\right)\right)$ writes and $O\left(\frac{1}{B}\left(\lambda+\beta\right)\right)$ reads for any $\lambda > 1$ and $\beta \leq \left\lfloor \log_\lambda \frac{M}{B\log_\lambda N}\right\rfloor$. A query for a key K performs $O(D_K \log_\lambda N)$ reads, where D_K is the number of times K has been inserted or deleted.*

Proof. Each insertion will eventually be added to each level i. When the level i is not a queue level, $i \bmod \beta$ characters per item will be read from the character queues on the queue level below i. Summed over all levels, this yields

$$\sum_{i=0}^{\log_\lambda N} \frac{i \bmod \beta}{B\log_\lambda N} = O\left(\frac{\beta}{B}\right)$$

reads per insertion.

A λ^β-way merge is performed every β levels. Since $\beta \leq \left\lfloor \log_\lambda \frac{M}{B\log_\lambda N}\right\rfloor$, this requires $O(1/B)$ reads and writes per element. This contributes $O\left(\frac{\log_\lambda N}{\beta B}\right)$ reads and writes per insertion in total.

The per-insertion read and write cost of building the routing filters is $\Theta(\lambda/B)$ as in the proof of theorem 3.

The cost per query is the same as in theorem 3. □

Theorem 4 can improve the write cost when $\lambda = o(\log\log M)$ and $\log\log M = \Omega\left(\log_{\frac{M}{B}} N\right)$. In that case, β can be tuned to optimize insertion performance relative to the ω of the AEMM by solving the quadratic equation $\beta = \omega \cdot \left(\lambda + \frac{1}{\beta}\log_\lambda N\right)$.

In particular, an interesting consequence of theorem 4 is in the case where N is polynomial in M. Then insertions can be performed into a $(\log_\lambda M/B)$-asymmetric BOT with growth factor $\lambda = O(1)$ with constant write amplification. This does come at the cost of more reads than when using a regular BOT:

Corollary 5. *If $N = O(M^c)$ for some constant c, then a $(\log_\lambda M/B)$-asymmetric BOT with growth factor $\lambda = O(1)$ containing N key-value values performs $\Theta(1/B)$ amortized*

writes per insertion, $\Theta\left(\frac{1}{B}\left(\lambda + \log\frac{M}{B}\right)\right)$ *amortized reads per insertion and* $\Theta(\log N)$ *reads per query.*

Chapter 3

SplinterDB

3.1 Introduction

Key-value stores form an integral part of system infrastructure. Google's LevelDB and Facebook's RocksDB are widely used, both within and outside of their companies. Their importance has spurred research into several aspects of key-value store design, such as increasing write throughput, reducing write amplification, and increasing concurrency [87, 50, 97, 91, 26, 43, 44, 42, 15, 29, 12, 20, 72, 34, 39, 57, 109, 74, 103, 98, 53, 114, 111, 67, 65, 54, 68, 11].

Existing key-value stores face new challenges with the increasing use of high-performance NVMe solid state drives (SSDs) in industry. NVMe SSDs offer substantially higher bandwidth (500K-600K IOPS) and lower latency (10-20 micro-seconds) than other SSDs.

These key-value stores struggle to utilize all the available bandwidth in modern SSDs. For example, we find that for the common case of small key-value pairs, RocksDB is able to use only 30% of the bandwidth supplied by an Optane-based Intel 905p NVMe SSD (even when using 20 or more cores).

We find that the bottleneck has shifted from the storage device to the CPU: reading data multiple times during compaction, cache misses, and thread contention cause RocksDB to be CPU-bound when running atop NVMe SSDs. Thus, there is a need to redesign key-value stores to avoid these CPU inefficiencies. While KVell [70], a new research key-value store, also tries to reduce CPU overhead, it presents a design optimized for large key-value pairs. In particular, we show that KVell experiences an extreme performance cliff when it does not have enough memory to hold its in-memory

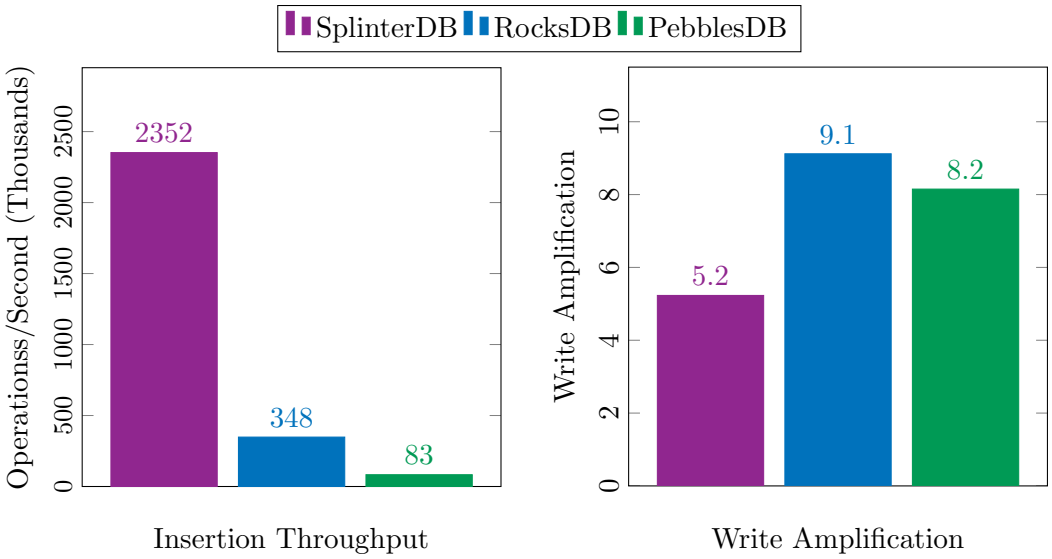


Figure 3.1: YCSB load throughput and write amplification benchmark results with 24-byte keys and 100-byte values.

index, a limitation also acknowledged by its authors.

We present SplinterDB, a key-value store designed for high performance on NVMe SSDs. For example, on small key-value pairs, SplinterDB is able to fully utilize the device bandwidth and achieves almost 2× lower write amplification than RocksDB (see fig. 3.1). We show that compared to state-of-the-art key-value stores such as RocksDB and PebblesDB, SplinterDB is able to ingest new data 6–28× faster (see fig. 3.1) while using the same or less memory. For queries, SplinterDB is 1.5-3× faster than RocksDB and PebblesDB.

Three novel ideas contribute to the high performance of SplinterDB: the *STB$^\varepsilon$-tree*, a concurrent memtable that removes the insertion scalability bottleneck, and a concurrent user-level cache that reduces cache-interference in highly-concurrent settings. All three components are designed to enable the CPU to drive high IOPS without wasting cycles.

At the heart of SplinterDB is the STB$^\varepsilon$-tree, a novel data structure that combines ideas from log-structured merge trees and B$^\varepsilon$-trees. The STB$^\varepsilon$-tree adapts the idea of size-tiering (also known as fragmentation) from key-value stores such as Cassandra and PebblesDB and applies them to B$^\varepsilon$-trees to reduce write amplification by reducing the number of times a data item is re-written during compaction. The STB$^\varepsilon$-tree also

enables localized, fine-grained compactions that increase compaction concurrency across the entire tree. By enabling fine-grained, localized compactions, STB$^\varepsilon$-trees push ideas from PebblesDB to their logical conclusion.

In key-value stores such as RocksDB, all inserted data is first stored in an in-memory component called the memtable. We find that the memtable in RocksDB provides low concurrency and becomes the bottleneck when used on top of the highly-concurrent STB$^\varepsilon$-tree on NVMe devices. We redesigned the memtable for high concurrency: similar to the STB$^\varepsilon$-tree, the SplinterDB memtable is based on a B-tree designed using a 4KB disk block size and the L3 cache-line size; as a result, when data needs to be moved between the memtable and STB$^\varepsilon$-tree, it can be done using simple pointer manipulations, reducing the CPU cost.

Finally, key-value stores such as RocksDB and PebblesDB use the Linux page cache, but we found the page cache ill-suited for high concurrency. We designed a new user-level concurrent cache for SplinterDB that uses fine-grained, distributed reader-writer locks to avoid contention and ping-ponging of cache lines, as well as a direct map to enable lock-free cache operations. All the data read and written by SplinterDB flows through this concurrent cache.

SplinterDB is not without limitations. Like all key-value stores based on size-tiering, SplinterDB sacrifices the performance of small range queries, although less than one might expect. For large range queries, SplinterDB can use the full device bandwidth. Similarly, size-tiering is known to temporarily increase space usage until multiple versions of a single data item are compacted together. Finally, SplinterDB is meant for scenarios where good performance is required when memory is low; if memory is plentiful and query performance is not as important, then key-value stores such as KVell might be a better fit. Despite these limitations, SplinterDB represents an interesting new point in the design spectrum for key-value stores.

In summary, the contributions of SplinterDB are as follows:

- We introduce the STB$^\varepsilon$-tree, which reduces write amplification and enables fine-grained concurrency in compaction operations (sections 3.3 to 3.5).

- We design and build a highly-concurrent memtable that is able to drive enough operations to the underlying STB$^\varepsilon$-tree (section 3.6).
- We combine the STB$^\varepsilon$-tree, memtable, and user-level cache in SplinterDB, a key-value store that can fully utilize NVMe SSD bandwidth. (section 3.7).

3.2 Background

This section describes the building blocks we use to construct the STB$^\varepsilon$-tree. We also describe related data structures, such as the log-structured merge tree (LSM), in order to put our data-structural innovations in context. Finally, we summarize theoretical performance analyses of the data structures.

The DAM model.

We use the Disk Access Machine (DAM) model [5] in our performance analyses. In the DAM model, data is transferred between disk and RAM of size M in blocks of size B words. The I/O cost of a data structure is the number of block transfers. In DAM model analyses, key-value pairs have size $O(1)$ words, so the write-amp of a key-value workload is $\Theta(B \times W)$, where W is the amortized number of writes per insertion.

B-trees.

We analyze the write-amplification of B-trees to serve as a baseline. Each insertion into a B-tree requires $\approx \log_B N$[1] writes in the worst case, but almost all insertions modify only a leaf of the B-tree and hence require only 1 write. Although caching can help when the data set is small or the insertion workload has locality, in the worst case of random inserts into a large B-tree, each insertion will have to bring in a new leaf, causing an old, dirty leaf to be written back to disk. Thus the worst-case write amplification of a B-tree is $\approx B$, much greater than that of LSMs and B$^\varepsilon$-trees.

[1]Throughout the paper, we use $\approx$ to indicate an analytical result accurate up to lower-order terms. In other words, when we write $\approx X$, we mean $X + o(X)$.

It is common to size the hardware so that all the indexing nodes of the B-tree fit in RAM, i.e. RAM has size $M = N/B$. In this case each query and insert costs $O(1)$ I/O once the cache is warm.

Log-structured Merge Trees.

A generic LSM consists of $k \approx \log_F \frac{N}{M}$ levels, $L_0, \ldots, L_{k-1}$. Each level contains a sorted ***run*** of key-value pairs. The first run, L_0, has capacity $\Theta(M)$, and each subsequent run has capacity F times bigger than the previous. The ***fanout*** F is a parameter that trades off between insertion throughput and query latency. Each sorted run is called an *SSTable*.

New items are inserted into L_0. When an SSTable fills, it is merged into the next table, which is called compaction. An SSTable can receive F such merges before it is full, so each element participates in an average of $F/2$ compactions on a level before being compacted into the next level. Compaction is just a sequential scan since both tables are sorted. A compaction of total size K takes $\approx K/B$ writes, or $\approx 1/B$ writes per key-value pair, so the average write amplification is approximately $\frac{F}{2} \log_F \frac{N}{M}$. Assuming the cache has size $M = N/B$, this simplifies to $\frac{F}{2} \log_F B$.

Most implementations also store indexing information about each SSTable, so that queries in a table have similar performance to queries in a B-tree. Naively, a query in an LSM requires searching in each level. for a total cold-cache query cost of $O(\log_F \frac{N}{M} \log_B N)$ I/Os. Given a warm cache of size $M = N/B$, we can cache the indexing information of all the SSTables, so that each SSTable query requires $O(1)$ I/Os. Then, on a warm cache, the query cost becomes $O(\log_F N/M)$, which is significantly higher than the $O(1)$ warm-cache query in a B-tree given the same cache size. This is for a generic LSM; most LSM implementations use filters to reduce queries to $O(1)$ I/Os, as described below.

An off-the-shelf LSM is typically asymptotically much faster than a B-tree for insertions but, for some insertion workloads that have high locality, can be asymptotically slower. Queries can also be asymptotically slower than in a B-tree, but this can largely be mitigated with filters.

B$^\varepsilon$-Trees.

The generic B$^\varepsilon$-tree [29] is a B-tree that uses part of the space in each node to buffer items recently inserted into the subtree rooted at that node. Queries must check for relevant mutations in each buffer along their search path.

Insertions place an item in the root node's buffer. When the buffer in a node becomes full, the B$^\varepsilon$-tree moves some of the elements in the buffer to the buffer of one of its children. The B$^\varepsilon$-tree always moves items to the child that would be examined during a search for those items, ensuring that a future query for one of those items will find it. This process is called a *flush* and is analogous to an LSM compaction. There are several options for how to select the items to be flushed. The most common policy is to select the child for which the most items are buffered in the parent and then flush all the items buffered for that child.

To analyze the B$^\varepsilon$-tree in the DAM model, let $F \ll B$ be the fanout of the B$^\varepsilon$-tree.[2] Thus pivots and child pointers consume $O(F)$ space in each node. The remaining $B - O(F) \approx B$ space in each node is used for buffering. Thus the height of the tree is $\approx \log_F N$ and the cold-cache query cost is $\approx \log_F N$ I/Os.

Each flush costs $O(1)$ I/Os and moves at least B/F elements one level down the tree. Since each element moves at most $O(\log_F N)$ levels down the tree, insertions cost $O((F \log_F N)/B)$ amortized IOs, which is the same as that of an LSM. Likewise, the write amplification of a B$^\varepsilon$-tree is $O(F \log_F N)$ without caching.

With a cache of size $M = N/B$, we can cache the top of the tree, reducing query costs to $O(\log_F B)$ I/Os. Insertions become $O((F \log_F B)/B)$ I/Os, and write amplification reduces to $O(F \log_F B)$, which are both the same as an LSM with the same size cache.

One advantage of B$^\varepsilon$-trees is that they can naturally exploit locality in the insertion workload to improve insertion performance, much as a B-tree can. This is because flushes are not done on a level-wide basis, but node by node. Thus, for example, if a workload consists of insertions all destined for a small sub-tree, a B$^\varepsilon$-tree can cache

[2]Historically $F = B^\epsilon$, where $0 \leq \epsilon \leq 1$, which is where the name comes from. We use F to ease the comparison with LSMs.

that sub-tree to reduce write amplification and improve insertion throughput.

Filters.

Many LSM implementations mitigate the high query cost of a generic LSM by using ***filters***; bloom filters [24] are the most well known filters. The space requirement of a filter is $O(n \log 1/\epsilon)$ for a set of a size n. For typical values of $\epsilon \approx 1\%$, this is about 1 or 2 bytes per element. When filters are small enough to fit in RAM, they can reduce the I/O costs of cold-cache point queries in LSMs to $O(\log_B N)$, which is the same as in B-trees. With a cache of size $M = N/B$, queries cost $O(1)$ I/Os. The same calculation holds for B^ε-trees. Note, however, that filters cannot be used to speed up range queries, since filters do not support range emptiness queries.

In our work we use a variant of quotient filters [18], because inserts and lookups access only a constant number of distinct locations [18], making them generally faster than Bloom filters. Furthermore, quotient filters are, for the false-positive rate used in SplinterDB, roughly the same size as Bloom filters.

Size Tiering.

Cassandra introduced the notion of *size tiering* in an LSM. In a size-tiered LSM (STLSM), each level has up to F SSTables of approximately the same size. When a level reaches its maximum number of SSTables, its SSTables are merged into one SSTables, which is moved to the next level. The advantage of size-tiering is that each item is involved in only one compaction per level. The downside is that queries now have more places to search.

Kuszmaul [68] showed that with size tiering, write amplification decreases by a factor of F from that of a generic LSM to $O(\log_F \frac{N}{M})$, while queries increase by a factor of F to $O(F \log_F \frac{N}{M} \log_B N)$ IOs. Even if we assume that the cache has size $M = N/B$, size tiering still trades off a factor of F reduction in write amplification for an F-fold increase in query costs. However, by maintaining a filter for each SSTable on every level, the point query costs can be kept at $O(1)$ IO per positive query and no IOs per negative query. Size tiering also increases the costs of small range queries, and filters

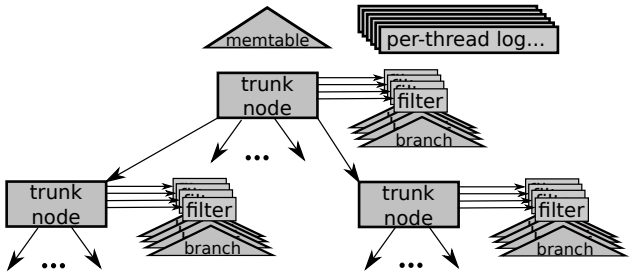


Figure 3.2: Overall design of STB$^\varepsilon$-trees and SplinterDB. Trunk nodes contain pivots and child pointers and pointers to a collection of branches and their associated filters. Each branch is a B-tree. SplinterDB also keeps a queue of memtables to enable pipelining of memtable compactions.

don't mitigate that cost.

3.3 High-Level Design of STB$^\varepsilon$-trees

We now describe the design features of STB$^\varepsilon$-trees that give them low write amplification, low pass complexity, and high concurrency, without sacrificing lookup performance.

At a high level, a STB$^\varepsilon$-tree is a B$^\varepsilon$-tree, as shown in fig. 3.2, albeit with several modifications to reduce I/O amplification and exploit the I/O parallelism of NVMe devices.

On a spinning disk, data locality is paramount. Thus, B$^\varepsilon$-trees designed for spinning disks typically store each node, including pivots, child pointers, and buffer contents, contiguously on disk. Furthermore, nodes are large—typically over a megabyte in size—in order to amortize the cost of the seek required to access the node. The downside of large nodes is that they make point queries expensive. Even if a point query has to load only a single leaf into cache, it still has to transfer a megabyte or more of data. Thus some B$^\varepsilon$-trees divide their nodes into a header and physically contiguous partitions. The header contains pivots, child pointers, and an index on the partitions, so that queries need only load the relevant partition in a node. Headers and partitions are typically 32-64KBs, which is small enough to ensure that, on a hard drive, point query performance is seek bound rather than bandwidth bound.

On NVMe, however, even transferring 32KB per query is too much. For example, on an Optane NVMe device, locality offers essentially no gain in throughput, i.e. the device can deliver its full bandwidth via a random I/O workload as long as the device queues are kept sufficiently full. Thus transferring 32KB per query would directly reduce maximum query throughput to 1/8th of the device's random I/O throughput.

Consequently, our STB$^\varepsilon$-tree strives not for locality, but rather for low I/O amplification and high I/O parallelism. Our STB$^\varepsilon$-tree is a tree of trees. The ***trunk tree*** (or simply trunk) is analogous to the headers of a traditional B$^\varepsilon$-tree, i.e. trunk nodes contain pivots, child pointers, and pointers and metadata for the node's branches. Trunk nodes are kept small—4KB in our implementation—so that they do not waste cache space. In all practical use cases, trunk nodes comprise less than 0.1% of the total data and so are essentially always cached.

Each ***branch*** is a static B-tree, also with 4KB nodes in our current implementation. Since each branch is constructed once and never modified, the B-tree nodes are always fully packed, which improves cache efficiency for point queries and reduces I/O amplification during compactions and range queries. Note that the B-tree nodes of a branch are not necessarily stored contiguously on disk, since locality is less important for utilizing the bandwidth of NVMe devices. Range queries and compaction use the pointers in branch nodes to prefetch leaves in advance, enabling our STB$^\varepsilon$-tree to take advantage of the I/O parallelism of NVMe devices.

Each branch also has an associated ***quotient filter***. Quotient filters serve the same role as Bloom filters in many LSM implementations. However, we choose quotient filters because they have substantially greater insert and query performance than Bloom filters [18], reducing the CPU costs of both queries and compactions, while using roughly the same or slightly less space than Bloom filters for the false-positive rate used in SplinterDB (1/256). Quotient filters are also efficient if they get paged out to disk, since each lookup accesses only one page.

SplinterDB uses ***memtables*** to collect new insertions. Our memtable is a dynamic B-tree. In fact, it is the same structure as the B-trees used to store branches except that, since it is dynamic, the nodes are not always fully packed. When a memtable

fills, it is locked for new insertions, then the quotient filter is constructed and then the memtable is inserted as a branch into the trunk root. No serialization or other work needs to happen. When former memtables participate in compactions, the resulting B-tree is packed.

SplinterDB uses write-ahead logical logging for crash recovery. SplinterDB uses per-thread logs to support highly concurrent updates. Inter-log ordering is maintained by cross-referencing log entries with timestamps on the leaves of the memtable (see section 3.6.4 for details).

Note that all of the above data structures—memtables, trunk nodes, branch nodes, filters and logs—are pageable. SplinterDB has a unified CLOCK cache for all these structures.

3.4 Size-Tiering with Workload-Driven Compaction

One of the goals of SplinterDB is to get both the benefits of size- tiering and the benefits of B$^\varepsilon$-tree's workload-driven compaction and flushing. Size-tiering reduces write amplification of all workloads. Workload-driven compaction and flushing further reduces write amplification when the workload is not uniformly random. Our flushing and compaction algorithm is designed to preserve worst-case performance guarantees of size-tiered LSM trees while exceeding their performance on non-random workloads.

The challenge is that a flushing and compaction algorithm must balance two competing objectives. First, we want to move data from one level of the tree to the next in large batches. This is the key reason that LSMs and B$^\varepsilon$-trees are so much faster than B-trees for insertions. On the other hand, we want to limit the number of locations that must be searched during a query. This means we cannot accumulate data indefinitely before merging it into lower levels of the tree.

We begin by explaining the structure of STB$^\varepsilon$-tree nodes. This structure will enable us to cleanly accomplish the above goals while also enabling us to skip some compactions when the workload allows it.

Node structure. Each trunk node has a list of branches, sorted from oldest to

newest. The trunk node also stores, for each child, the index of the next branch to be flushed to that child. Branches are flushed to a child in chronological order, so all older branches have already been flushed to that child, and all newer branches are yet to be flushed to that child. We say that a branch is ***live*** for a child if it hasn't been flushed to that child. We say that a message in a branch is ***live*** if the message's branch is live for the message's target child. Finally, each trunk node stores, for each child, a rough estimate of the number of live messages for that child across all the parent's branches. This estimate is made by scanning the top-level nodes of the branches and estimating the amount of data in each subtree that falls entirely within the pivots for a child. Since branches are packed, these estimates are quite accurate (typically to within less than 1%).

Trunk nodes have a fixed number of branches that they can hold. In the current SplinterDB implementation, each trunk node can hold up to 84 branches. However, trunk nodes begin flushing when they have $3F$ live branches, which is typically far less than 84 (e.g. F is in the range of 8 to 16). This extra capacity is used to enable nodes to absorb new incoming branches while compacting old branches.

Branches may be referenced by more than one trunk node. Each trunk node knows the range of keys that it covers so, for example, when a parent and child both refer to the same branch, the parent might refer to all the messages in the branch, while the child refers to only the subset of messages in the branch for keys covered by the child. Branches are immutable and refcounted, so sharing branches is safe.

This node structure and branch sharing means that we can "flush" a live branch from a node to one of its children by simply adding a pointer to the branch to the child. We can then mark the branch as dead for that child in the parent. If the branch becomes completely dead in the parent (i.e. dead for all of its children), then the parent can delete its references to the branch, decrementing the branch's refcount. Thus flushes are extremely cheap—just a few pointer and refcount updates.

Flush then Compact. SplinterDB avoids some intermediate compactions by using a "flush-then-compact" approach. Each flush may trigger some compactions and some further, recursive flushes (see below for a description of SplinterDB's flushing policy).

The idea of flush-then-compact is to perform all the recursive flushes first. Only once all the branches have been moved as far down the trunk as possible do we begin performing compactions. This will enable some branches to skip intermediate compactions within the tree.

Once all the flushes from a node have completed, SplinterDB initiates background compactions on all the nodes that received new branches. Each background compaction compacts only the new branches in that node. Thus no message gets compacted twice without being flushed from one trunk node to another.

Compaction does not interfere with other concurrent tasks. The trunk node is not locked during the compaction—other threads may perform queries or flush more branches to or from the trunk node. During this time, the compaction thread constructs a new branch that is the compaction of all its input branches. When the compaction thread finishes, it briefly locks the trunk node to replace the old branch pointers with a pointer to the new, compacted branch.

This mechanism also reduces contention at the root, since no compactions are ever performed in the root. Rather, whenever the root fills, branches are moved to some of its children and compactions are performed there, immediately making room for new items in the root.

Note that compactions skip over portions of the branches that are not relevant to the compaction. For example, when we flush branches from the root to one of its children, the branches may contain keys outside the range covered by that child. When these branches get compacted in the child, the compaction process won't even look at those keys. We can do this efficiently because branches are B-trees that support iterators starting anywhere in the branch.

Flushing policy. The memtable has a maximum size of m messages. Once it reaches size m, it is added as a new branch to the root trunk node.

Flushes from a trunk node to one of its children are triggered by two conditions: either the trunk node has more than Fm live data, or one of the trunk node's children has more than $3F$ live branches, where F is the fanout of the trunk. These two conditions serve distinct performance objectives.

The too-much-live-data trigger works with the compact-then-flush algorithm to detect localized insertion workloads and move them quickly down the tree without performing unnecessary intermediate compactions.

For example, imagine a sequential insertion workload. These inserts first go to the memtable. Once the memtable fills, it is added as a branch to the root. Once the root accumuates F such branches, it will go over the max-live-data threshold, causing a flush. This flush will move all the branches down the tree towards their target leaf. This will immediately cause the child to exceed the max-live-data threshold, so the branches will get flushed towards their target leaf again. This will repeat until the branches reach the leaf, at which time SplinterDB will perform a compaction (and will probably split the leaf). The next batch of inserts will go to the new leaf created by the split. Thus each message will be involved in only a constant number of compactions, giving $O(1)$ a write- and pass-complexity.

This method automatically adapts to varying degrees of locality. For example, if the workload is half inserts to a single leaf, then every other time the root fills, we will perform a flush from the root to the target leaf. Or, if the workload consists of random inserts of keys that all fall within a subtree T of height, say, $h/2$, then every time the root fills, SplinterDB will flush all the branches to the root of the subtree without performing intermediate compactions, skipping half the compactions that would occur in a size-tiered LSM tree.

This policy does not weaken the worst-case insertion performance guarantees of a size-tiered LSM: each message undergoes at most one compaction per level of the tree, and the height of the tree is still $\log_F N/m$.

The second flushing trigger is designed to bound the number of filters and branches that must be examined during a query. Whenever there are more than $3hF$ branches on a search path (where h is the height of the trunk), at least one of the trunk nodes will violate the max-live-branches condition. This will trigger a flush and compaction, which will reduce the number of branches checked by queries along that path.

3.5 Preemptive Splitting for STB$^\varepsilon$-trees

Splits and merges pose problems for hand-over-hand locking in B-trees (and B$^\varepsilon$-trees). Hand-over-hand locking proceeds from root to leaf, but splits and merges proceed from the leaves up.

An approach to solving this issue in B-trees is to use preemptive splitting and merging [92]. During a B-tree insert, if a child already has the maximum number of children, then it is split while the insertion thread still holds a lock on its parent. Then the insertion can release the parent's lock and proceed down the tree, assured that the child will not need to split again as part of this insertion. Analogously, deletions merge a child with one of its neighbors if the child has the minimum number of children. This works because insertion and deletions can increase or decrease the number of children of a node by at most 1.

This approach does not work in B$^\varepsilon$-trees, because a flush to a leaf could cause that leaf to split multiple times. In STB$^\varepsilon$-tree with flush-then-compact, we can move all pending messages along a root-to-leaf path to the leaf before performing any compaction, splits, or merges. The total number of messages moved to the leaf is bounded by $O(B \log_F N)$, i.e. the height of the tree times the maximum amount of data that can be stored in branches at each trunk node. The leaf can therefore split into as many as $O(\log_F N)$ new leaves of size B. Similarly, a collection of flushes full of delete messages to several leaves of a single parent can reduce the parent's number of children by $O(\log_F N)$.

In practice, the height of the tree is less than 10 for typical fanouts $F \approx 8$ and dataset size $N \leq 2^{80}$ key-value pairs.

We extend preemptive splitting and merging to STB$^\varepsilon$-trees as follows. We reserve space in each node to accommodate up to $F + H$ children, where H is an upper bound on the tree height, e.g. $H = 10$. We then apply preemptive splitting, except we preemptively split a node during a flush if its fanout is above F. For merges, we take a similar approach. If, during a flush, we encounter a node with less than $F/2$ children, then we merge or rebalance it with one of its siblings.

Thus all operations on the STB$^\varepsilon$-tree—flushes, splits, and merges—proceed from root to leaf and can therefore use hand-over-hand locking.

The mechanisms for flush-then-compact make it easy to handle branches during splits. Recall that each branch can be marked dead or alive for each child, and branches are refcounted and hence can be shared by multiple trunk nodes. Thus we can split a trunk node by simply giving its new sibling references to all the same branches as the node had before the split. In the new node, we copy the liveness information for each branch along with the children that are moved to the new sibling.

3.6 From STB$^\varepsilon$-trees to SplinterDB

In this section, we discuss the details of SplinterDB's implementation. SplinterDB targets NVMe SSDs, and on NVMe SSDs, CPU is the primary bottleneck to write performance, and concurrency is the primary bottleneck to read performance. As a result, what would be minor design decisions for a key-value store which targets a slower storage medium become performance critical when targeting NVMe storage.

3.6.1 User-level Cache and Distributed Locks

SplinterDB has a single user-level cache which keeps recently accessed pages in memory. Almost all the memory that SplinterDB uses comes from this cache, so pages from all parts of the data structure—trunk node pages, branch pages, filter pages and memtable pages—are all stored there. Only cache and file-system metadata, as well as small allocations used to enqueue compaction tasks are allocated from system memory.

This design allows nearly all the free memory to be used for whichever operations are being performed, so that parts of the data structure which are not in use can be paged out.

The cache at a high level is a clock cache, but with several features designed to improve concurrency.

Each thread has a thread-local hand of the clock, which covers 64 pages. The thread draws free pages from the hand, and if it has exhausted them, it acquires a new hand

from a global variable using a compare-and-swap. It then writes out dirty pages from the hand which is a quarter turn ahead, and evicts any evictable pages in its new hand. Thus threads clean and evict pages from distinct cache lines within the cache metadata, avoiding contention and cache-line ping-ponging.

SplinterDB uses distributed reader-writer locks [55] to avoid cache-line thrashing between readers. Briefly, a distributed reader-writer lock consists of a per-thread reader counter and a shared write bit. Each reader counter is on a separate cache line to avoid cache-line ping-ponging when readers acquire the lock. Writers set the write bit (using compare and swap) and then wait for all the read counters to become zero. Readers acquire the lock by incrementing their read counter and then checking that the writer bit is 0. If it is not, they decrement their reader counter and restart.

Distributed reader-writer locks allow readers to scale essentially perfectly linearly, at the cost that acquiring a write lock is expensive. However, the design of SplinterDB makes writing rare enough that this is a good trade-off.

We make distributed reader-writer locks space efficent by storing each thread's reader counters in an array indexed by cache-entry index. Each reader counter is one byte, so the total space used by locks is $t \times c$ bytes, where t is the number of threads and c is the number of cache entries.

SplinterDB supports three levels of lock: read locks, "claims", and write locks. A claim is a read lock that can be upgraded to a write lock. Only one thread can hold a claim at a time. After obtaining a read lock, a thread may try to obtain a claim by trying to set a shared claim bit with a test-and-set. If this fails, they must drop the read lock and start over. Otherwise, they can upgrade their claim to a write lock by setting a shared write bit and waiting for all the read counters to go to zero.

3.6.2 Branch Trees and Memtables

SplinterDB uses the same B-tree implementation for both its branches and its memtables, although there are some differences to optimize for their use cases. By using the same data structure, memtables can be incorporated into the STB$^\varepsilon$-tree directly as branch trees with no serialization. The only processing needed is the construction of a

quotient filter.

Branch trees

When a branch is created from a compaction, its key-value pairs are packed into the leaves of the B-tree, and the leading edge of internal nodes are created to index them. The nodes in each level are allocated in extents of 32 pages, and the header of each node stores the address of the following node, but also of the next extent. In this way, the nodes of each level form a singly linked list.

Iteration through a branch is performed by walking the linked list formed by its leaves. Whenever the iterator reaches the beginning of a new extent, it issues an asynchronus prefetch request for the next extent.

Memtables

The basic design of the memtables mirrors that of the branch B-trees, but includes some optimizations designed to increase their insertion performance and concurrency.

As in the case of the static branch trees, the nodes on each level of the memtable form a singly-linked list, and nodes are allocated in extents. However, because nodes are created on demand as nodes split, we do not try to guarantee that successive nodes reside in the same extent. Furthermore, since memtables are almost always in RAM, we do not perform prefetching during memtable traversals.

The memtable uses hand-over-hand locking, together with preemptive splitting. At each index node, first a read lock is obtained, which is upgraded only if a split is required. If an index node is full, the inserter tries to upgrade it to a claim; if this fails, that means another thread is already splitting the node, and the inserter attempts to continue down the tree. If it cannot continue because of held locks, or if it finds that the leaf is full, then it aborts and tries again from the root.

To ensure locks are held briefly, especially on nodes near the top of the tree, the tree uses a new technique called ***shadow splitting***. To split a node c, a claim is obtained on c and the parent p. We allocate a physical block number (PBN) n for the new sibling, c'. However, in the cache, we initially point n to c. We also add a new pivot to the

parent p, pointing to the new PBN n. At this point, we can release all locks on p. We then allocate space for c' and fill in its contents. We then update the PBN n to point to c' in the cache, and then release all locks on c'. Finally we upgrade to a write lock on c, truncate its child list (via a metadata operation) and then release all locks on c.

3.6.3 Quotient filters

Bloom filters [24] are the standard filter for most LSMs [54, 26, 91]. However, the cost of Bloom filter insertions can dominate the cost of sorting the data in a compaction. Therefore modern key-value stores often use more efficient filters; for example, RocksDB uses blocked Bloom filters [90];

Similarly, SplinterDB uses quotient filters [19, 18, 86] instead of Bloom filters. A full presentation of quotient filters is out of scope for this paper, but we review their salient features for SplinterDB. See Pandey, et al. for a full presentation on quotient filters [86]. The key feature of quotient filters is that, like blocked Bloom filters, each insert or query accesses $O(1)$ cache lines (and hence $O(1)$ page accesses). Quotient filters are roughly as space efficient as Bloom filters—for the range of parameters used in SplinterDB, quotient filters use between 0.8× and 1.2× the space of a blocked Bloom filter. We view the space as essentially a wash. Quotient filter inserts and lookups also require only one hash function computation. In past work, quotient filter insertions and queries were shown to be 2-4× faster than in a Bloom filter.

SplinterDB further reduces the CPU costs of filter building during compaction by using a bulk build algorithm. During the merging phase of compaction or when inserting into a memtable, SplinterDB builds an unsorted array of all the hashes of all the tuples compacted or inserted. The array is then sorted (by hash value) and the quotient filter is built. Since the quotient filter also stores the hashes in sorted order, this means that the process of inserting all the hashes is a linear scan of the sorted array and of the quotient filter. Hence it has good locality and can benefit from cache prefetching.

3.6.4 Logging and Recovery

SplinterDB uses per-thread write-ahead logical logging for recovery. By using per-thread logs, we avoid contention on the head of a single, shared log.

The challenge is to resolve the order of operations across logs after a crash. For this, we use a technique similar to "cross-referenced logs" [56]. Our scheme works as follows. Each leaf of the memtable has a generation number. Whenever a thread inserts a new message into the memtable, it records and increments the generation number of the memtable leaf for the inserted key. It then appends the inserted message to its per-thread log, tagged with the leaf's generation number. During recovery, the generation numbers in the logs give a total order on the operations performed on each leaf (and hence on all the keys for that leaf), so that the recovery procedure can replay the operations on each key in the proper order. When a leaf of the memtable splits, the new leaf gets the same generation number as the old leaf.

3.7 Evaluation

We evaluate the performance of SplinterDB on several microbenchmarks and on the standard YCSB application benchmark[40]. We compare this performance against that of two state-of-the-art key-value stores, RocksDB and PebblesDB. The following questions drive our evaluation:

- Does SplinterDB achieve its primary goal of improved insertion performance?
- To what extent is this performance achieved through reduced write amplification as opposed to other factors?
- Does increasing insert performance come at a cost to [range] query performance? In particular, do queries in SplinterDB require more I/O, due to size tiering, than in non-size-tiered systems?
- Are sequential (or otherwise local) insertions faster, as predicted on SplinterDB? Do they have lower write amplification?

- Can SplinterDB utilize device bandwidth for large range queries?
- SplinterDB is designed to be highly concurrent; do point lookups scale with the number of threads?

3.7.1 Setup and Workloads

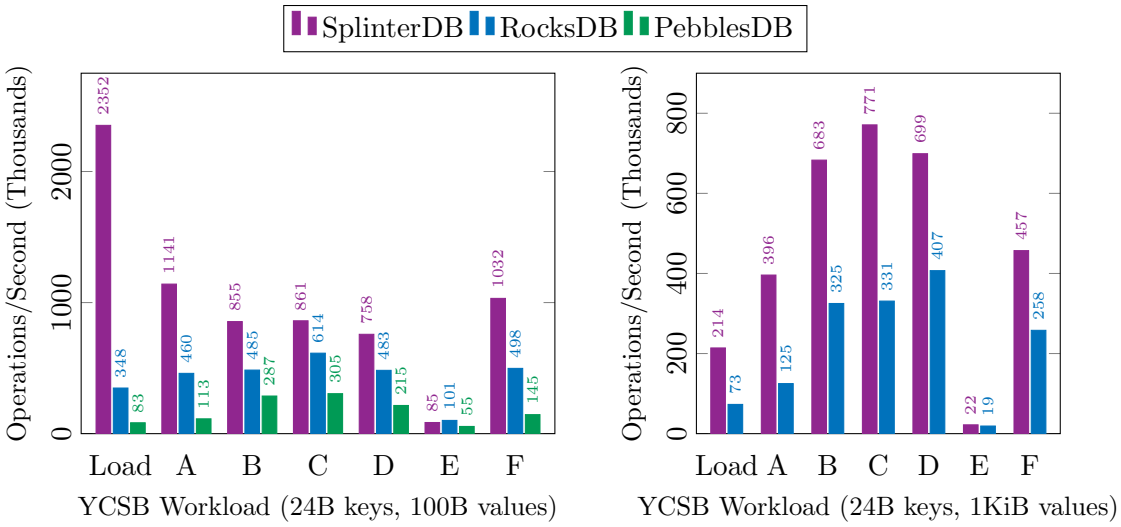
All results are collected on a Dell PowerEdge R720 with a 32-core 2.00 GHz Intel Xeon CPU, 256 GiB RAM and a 960GiB Intel Optane 905p PCI Express 3.0 NVMe device. The block size used was 4096 bytes.

In general, we use workloads derived from YCSB traces with 24B keys. We generally use 100B values, but also include a set of YCSB benchmarks for 1KiB values. We instrumented dry runs of YCSB in order to collect workload traces for the load and $A - F$ YCSB workloads and replay them on each of the databases evaluated. In order to eliminate the overhead of reading from a trace file during the experiment, the trace replayer `mmap`s the trace file before starting the experiment. We use the same traces for each system.

In general, we limit the available memory to 10% of the dataset size or less. In order to perform the benchmarks on reasonably sized datasets, we restrict the available system memory with a type 1 Linux `cgroup`, sized to the target memory size plus the size of the trace, which we pin so that it cannot be swapped out. Unless otherwise noted, the target memory size is 4GiB. PebblesDB has an apparent memory leak, which causes it to consume the available memory, so we allow it to use the full system memory. On the YCSB load benchmarks, this causes it to swap for a small portion at the end, but this was less than 10% of the run time.

Unless otherwise noted, SplinterDB uses a max fanout of 8, a memtable size of 24MiB and a total cache size of 3.25GiB. The difference between this cache size and the target memory size of 4GiB is to accommodate other in-memory data structures maintained by SplinterDB.

Each system is run with the thread count which yields the highest throughput. RocksDB is configured to use background threads equal to the number of cores minus



(a) Throughput on YCSB workloads with 24B keys and 100B values. Load is 673M operations, E is 20M operations and others are 160M operations. Higher is better.

(b) Throughput on YCSB workload with 24B keys and 1KiB values. Load is 84M operations, E is 1.3M operations and others are 10M operations. Higher is better.

Figure 3.3: YCSB throughput and I/O benchmark results.

the number of foreground threads, with a minimum of 4. PebblesDB uses its default number of background compaction threads. SplinterDB is configured without background compaction threads.

3.7.2 YCSB

We measure application performance using the Yahoo Cloud Services Benchmark (YCSB). The core YCSB workloads consist of load phases and run phases. The load phases create a dataset by inserting uniformly random key-value pairs. The run phases emulate various workload mixes. Workload A is 50% updates, 50% reads, workload B is 95% reads, 5% updates), workload C is 100% reads, workload D is read latest (95% reads, 5% insertions), workload E is short range scans (95% scans, 5% insertions) and workload F is read-modify-writes (50% reads, 50% RMWs).

We perform the benchmark with 24B keys and two different value size configurations: one with small 100B values, shown in figs. 3.3a and 3.4 and one with large 1KiB values, shown in fig. 3.3b.

On the load phase, SplinterDB is faster by almost an order of magnitude. Because

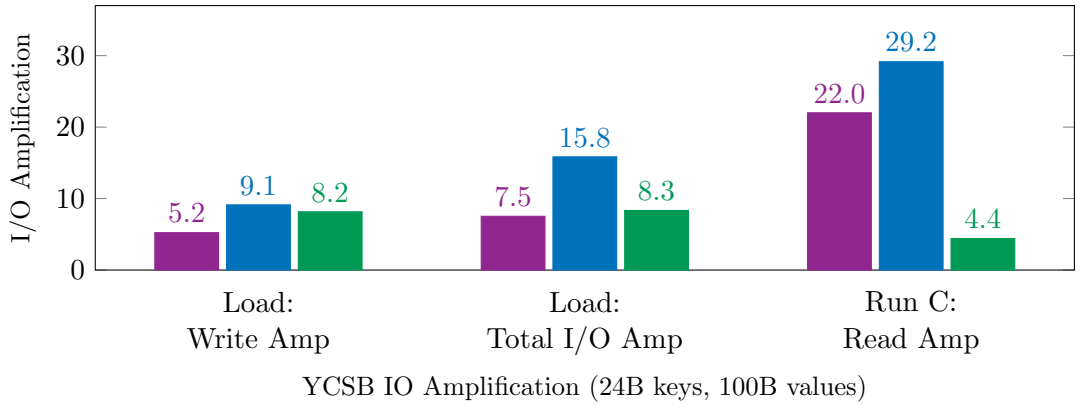


Figure 3.4: IO amplification on YCSB load and Run C workloads, as measured with iostat. Lower is better.

of size-tiering and its compaction/flushing policy SplinterDB has about 1/2 the write amplification of the other systems. Note PebblesDB performs almost no reads because it was given unlimited memory. Surprisingly PebblesDB does not show substantially lower write amplification than RocksDB.

On the run phases, which the exception of E, SplinterDB is 40–150% faster than RocksDB, the next fastest system. On E, SplinterDB is roughly half as fast as RocksDB.

3.7.3 KVell

KVell [70] is a key-value store also designed to utilize full NVMe bandwidth. It has an in-memory B-tree index that maps all keys to disk page offsets. It does well on large (4KiB) key-value pairs, but on small key-value pairs, the overhead of the in-memory index becomes a significant fraction of the dataset size. In particular, it was impossible to run KVell in a memory `cgroup` of 4GiB. Figure 3.5 shows KVell's performance on the YCSB workload with 100B values, for different memory sizes. At 22GiB, which is around the size of the in-memory index, KVell's performance starts to drop. At 20GiB, KVell becomes unusable. Therefore in realistic memory settings, KVell is not a viable option for the small key-value sizes that SplinterDB targets.

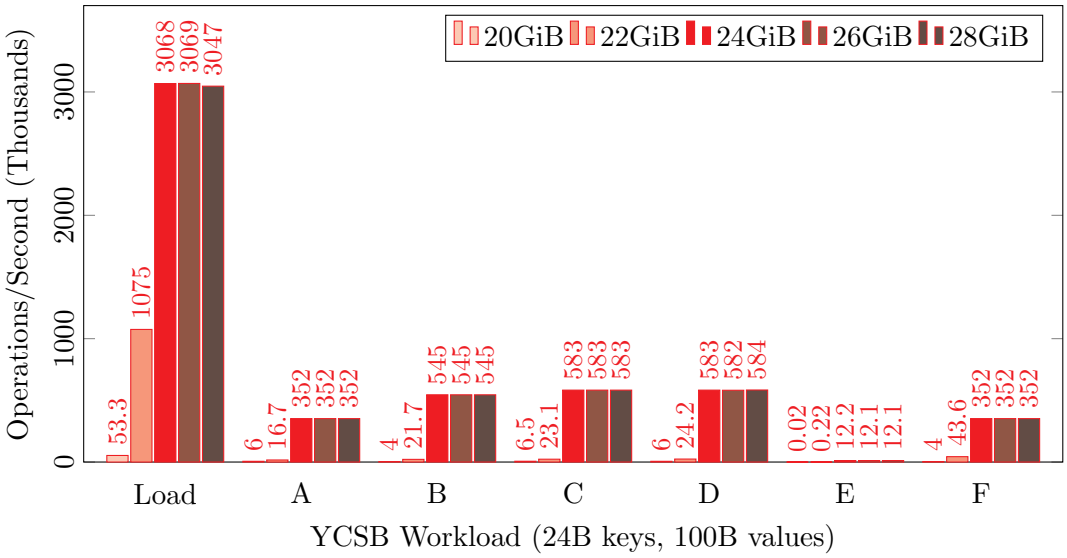


Figure 3.5: Throughput of Kvell on YCSB workloads with varying amounts of available RAM. Load consists of 673M operations, E consists of 20M operations and all other workloads consist of 160M operations. Higher is better.

3.7.4 Sequential Insertion Performance

Because SplinterDB is based on an STB$^\varepsilon$-tree and makes use of a flush-then-compact policy, we predict that its performance will improve substantially on insertion workloads with a high degree of locality (see section 3.4). We test this hypothesis by performing 20GiB of single-threaded insertions from a trace composed of interleaved sequential and random keys in different proportions. For comparison, we perform the same workload on RocksDB.

As shown in fig. 3.6, SplinterDB's performance improves smoothly from 349K insertions per second for a purely random workload to 614K insertions per second for a purely sequential workload, which is 76% faster. This improvement is partially obscured by the log, which adds a constant additive IO overhead. If we disable the log, SplinterDB improves from 430K insertions per second on a purely random workload to 866K operations per second on a purely sequential workload, 100% faster. Note that we would expect the intermediate throughputs in the best case to be the [weighted] harmonic mean of the pure cases, because they are rates. At 50% random, 50% sequential for SplinterDB with no log this is 575K insertions/second, so its actual performance of

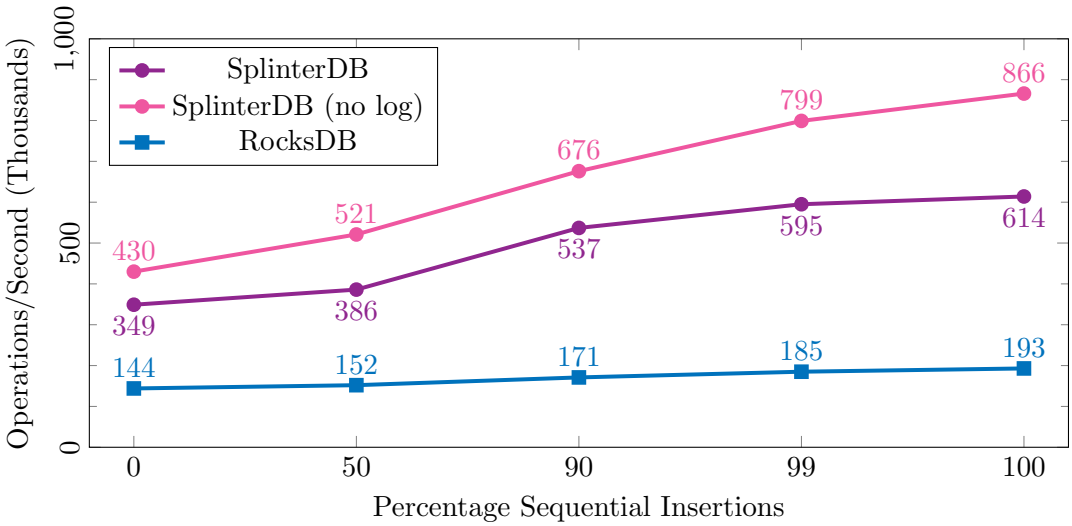


Figure 3.6: Single-threaded insertion throughput by locality. X-axis indicates the percentage of sequential keys. X-axis not to scale. Higher is better.

521K insertions/second captures a substantial amount of the potential improvement.

RocksDB also improves as the workload becomes more sequential, but this effect is much smaller, a 35% speedup. Furthermore, RocksDB shows less than 20% speedup until the workloads becomes 99% sequential.

Figure 3.7 shows that as predicted, SplinterDB incurs less IO amplification on more sequential workloads. With the log disabled, its write amp approaches 1 as the workload approaches purely sequential. In contrast, while RocksDB also has less IO amplification on more sequential workloads, it still incurs write amplification of 4.1 even when 99% of the keys are sequential. It is only when the workload becomes 100% sequential that the write amplification becomes close to 1 (because of caching it even falls below 1).

3.7.5 Concurrency Scaling

SplinterDB is designed to scale with the number of available cores up to the performance limits of the storage device. This is especially true for reads, where the use of distributed reader-writer locks and a highly concurrent cache design, together with a careful avoidance of dirtying cache lines, can avoid almost all contention between threads.

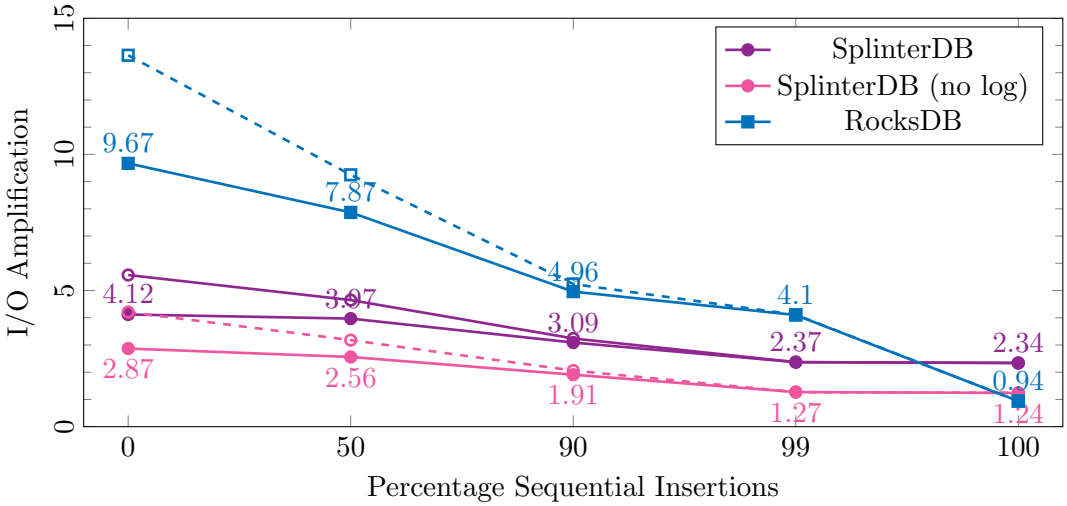


Figure 3.7: I/O amplification of mixed sequential/random insertion workloads. Shown are write amplification (solid) and total IO amplification (dashed) as measured with `iostat`. X-axis not to scale. Lower is better.

Read Concurrency

We test the read concurrency scaling of SplinterDB by running YCSB workload C with 160M key-value pairs, where (as in fig. 3.3a) each instance of the test divides the keys into N evenly divided batches, which are then performed in parallel by N threads. The results are in fig. 3.8.

The results show nearly linear scaling—throughput with 24 threads is 18.5× the single-threaded throughput. Between roughly 24 and 32 threads, the scaling flattens out, but at that point the measured throughput is 2.07–2.24 GiB/sec, which is 88–95% of the device's advertised random read capability.

While RocksDB also scales well, its throughput with 24 threads is 17.4× its single-threaded throughput, and with 32 threads it uses 91% of the device's advertised random read capability. Therefore, even though SplinterDB can perform more operations per second, RocksDB is still making nearly full use of the device for reads. We conclude here that SplinterDB is making better use of the available memory for caching, since it has noticeably lower read amplification.

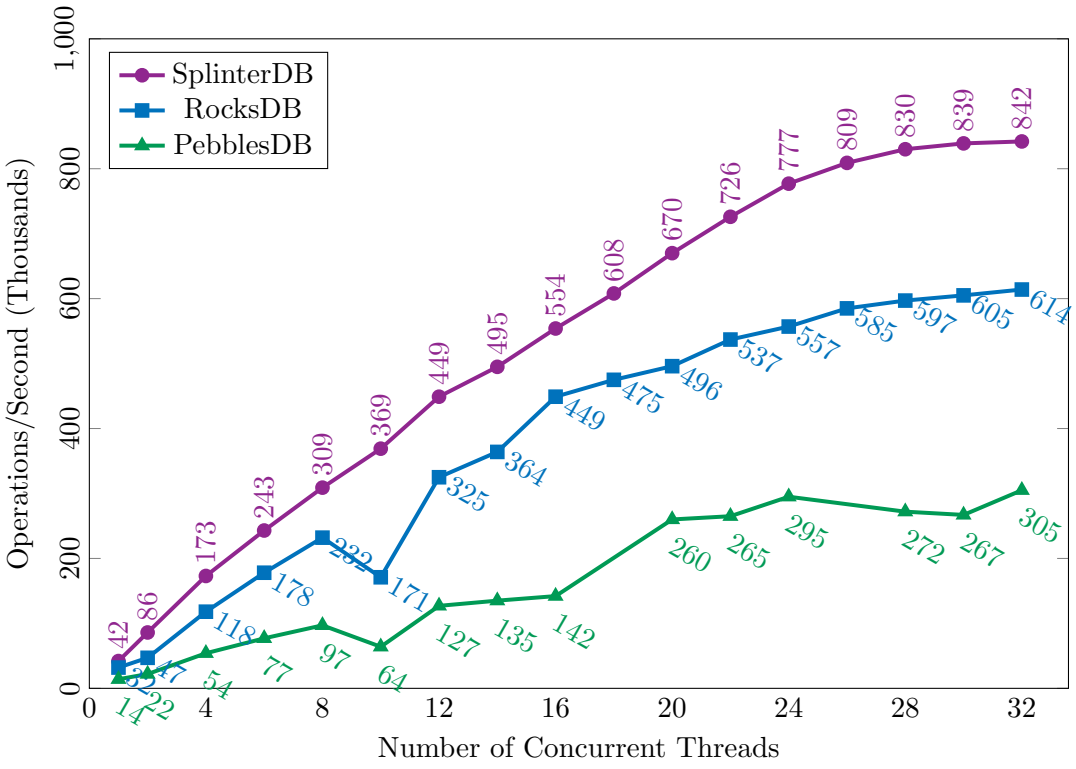


Figure 3.8: Read throughput performance (YCSB workload C) by number of threads. Each instance performs 160M reads divided evenly between threads. Higher is better.

Insertion Concurrency

We test the insertion concurrency scaling of SplinterDB by running the YCSB load workload with 673M key-value pairs divided into N batches, each of which is inserted in parallel by a different thread. Figure 3.9 shows selected N for each system, including its peak throughput.

The results show that SplinterDB scales almost linearly up to 10 threads. With 10+ threads, it performs 2.0-2.4M insertions per second with IO amplification around 7.5, which implies that it uses 1.9-2.2GiB/sec of bandwidth, which is at or near the device's sequential bandwidth of 2.2GiB/sec.

RocksDB's insertion performance also scales as the number of threads increases up to 14 threads, by a factor of 2.7. At its peak, it uses 754GiB/sec of bandwidth. PebblesDB scales slightly as well. For both RocksDB and PebblesDB, as many background threads as available are used for flushing and compaction during this benchmark.

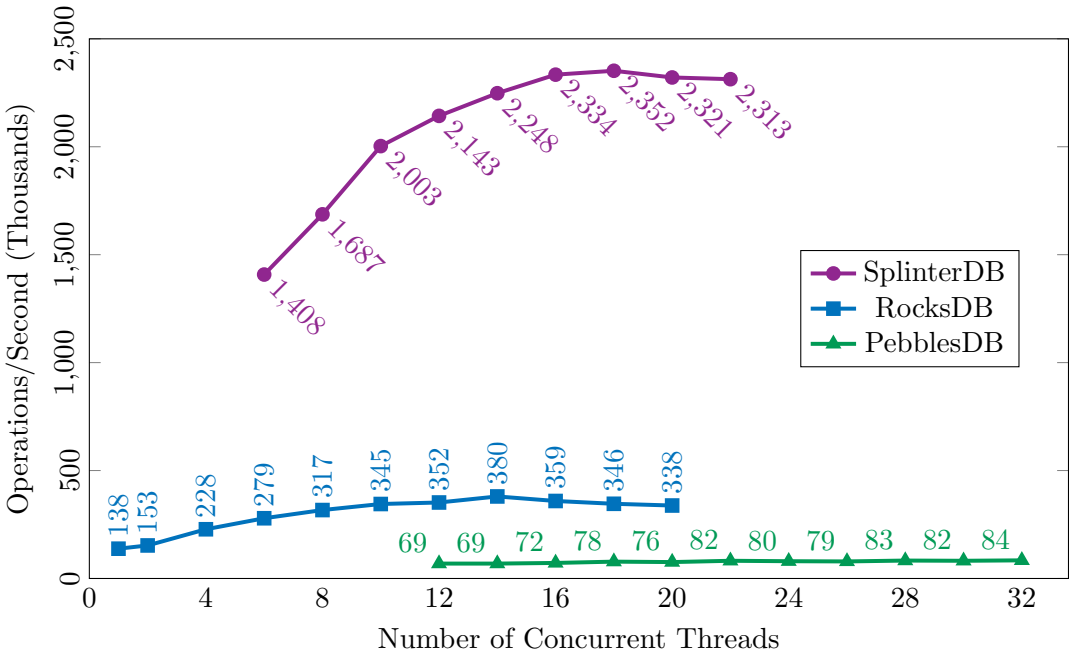


Figure 3.9: Write throughput performance (YCSB Load) by number of threads. Each instance performs 673M writes divided evenly between threads. Higher is better.

3.7.6 Scan Performance

An inherent disadvantage of size-tiering is that short scans must search every branch along the root-to-leaf path to the starting key. Each of these searches is likely to incur an IO to the device. As a result, as seen in fig. 3.3a, SplinterDB with 124B key-value pairs has scan throughput on small ranges that is about 85% that of RocksDB. During that workload, SplinterDB performed 2.26 GiB/sec of IO, which is within 96% of the device's advertised random read capability (short scans of small key-value pairs are essentially random reads).

However, once the initial search for the successor to the starting key has completed, the root-to-leaf path within each relevant branch will be in memory. Together with prefetching, this allows subsequent keys to be fetched at near disk bandwidth. Therefore, we expect that scans have a relatively high startup cost for the search to the starting key, followed by a very low iteration cost of obtaining subsequent keys.

Thus, when the amount of data requested grows to multiple pages, the disadvantage begins to dissipate. One way this happens is with larger key-value pairs: with 1kib

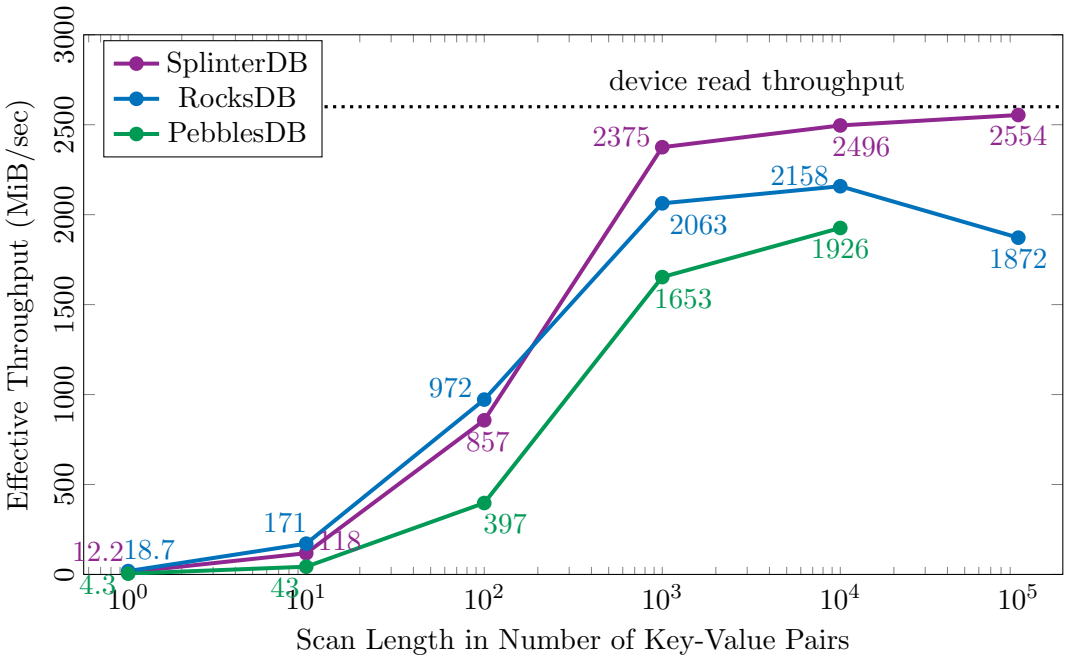


Figure 3.10: Scan throughput in MiB/sec as a function of scan length. For small scans, the start up cost dominates, but as the scans get longer, the throughput approaches the device's advertised bandwidth (2.6GiB/sec). The x-axis is on a log scale. Higher is better.

values, SplinterDB is about 16% faster than RocksDB.

Another way this can happen is with scans of more key-value pairs. We modify YCSB workload E to have only fixed-length scans of N key-value pairs, where N is 1, 10, 100, 1K, 10K or 100K. We perform runs of 10M scans of length 1, 10 and 100, 1M scans of length 1000, 100K scans of length 10000 and 10K scans of length 100000. Each run is performed on a dataset of 80GiB (with 24B keys and 100B values) and 4GiB memory.

The result is shown in fig. 3.10. Short scans on SplinterDB have low effective bandwidth, and in fact the bandwidth scales close to linearly with the scan length for scans of up to 100 key-value pairs. This suggests that for scans of this length, the startup cost dominates the iteration cost, which is as expected. As the scan length increases, the effective bandwidth of the scans approaches the device's advertised sequential read bandwidth, delivering 91% at scans of 1,000 key-value pairs. At scans as small as 100 key-value pairs, SplinterDB returns data at nearly half the bandwidth of the device.

3.8 Related Work

The closest work to ours is Tucana [87], a B^{ε}-tree optimized for SSDs. They also focus on CPU cost, concurrency, and write amplification. Our work pushes this to the even more demanding case of NVMe devices.

Size-Tiering. Cassandra [50], Scylla [97] PebblesDB [91], and RocksDB [26] (in "universal compaction" mode) use size-tiering to reduce write amplification. Size-tiering delays compaction of sorted runs in order to reduce write amplification. This can harm query performance because queries must search in more runs to find the queried item. Fluid LSMs [43], Dostoevsky [43], LSM bushes [44], and Wacky [44] use hybrids between size-tiering and level-tiering to tune the trade-off between write amplification and query performance. See [91] for a survey of LSM-compaction schemes.

Bloom filters in LSM trees. Almost all LSM trees use Bloom filters [24] to improve point-query performance, and specifically to mitigate the impact of size-tiering. Monkey [42] and ElasticBF [71] investigated how to allocate RAM to Bloom filters to improve query performance. Bloom filters do not affect range query performance, however, since they support only point queries.

Additional indexing. Numerous key-value stores use additional indexing to improve query performance. For example, COLAs [15] use fractional cascading, B^{ε}-trees [29] follow a B-tree-like structure, and PebblesDB uses randomized skip-list-based fractional cascading (referred to as guards in [91]). External-memory skip lists were analyzed in [12] and write optimized in [20]. The main technical challenge in such skip lists is the high variance of the size of runs, which in PebblesDB was addressed by turning long runs into mini-B-trees.

Write amplification vs. range queries. Several systems sacrifice range-query performance in order to reduce write amplification in other ways. Wisckey [72] reduces write amplification by declustering their key-value store: they log values and only store keys in the LSM-Tree. Since values are stored on disk in arrival order, a range query must gather values from the log. On NVMe, this is not a problem once the values are 4KB or larger. However, for smaller values, this can induce huge read amplification,

limiting range query performance to a tiny fraction of device bandwidth. HashKV [34] builds on Wisckey by introducing hash-based data-grouping to further reduce write amplification, but inherits Wisckey's range query performance limitations.

Other systems improve write amplification by sacrificing range queries altogether. Conway et al. [39] describe a write-optimized hash table, called the BOA, that also uses size-tiering with an LSM. In a BOA, SSTables of sorted runs are replaced with hash tables. They also introduce the concept of a routing filter, which extends the functionality of Bloom filters, in order to speed up queries. The principle advantage of routing filters is that performance does not degrade as much when they don't fit in RAM. The BOA meets a provable lower bound on the I/O costs of insertions and queries [57]. Thus the BOA is essentially the best possible on-disk data structure for random insertions and point queries. The downside is that the BOA does not support range queries, which are crucial to many key-value-store applications. LSM-tries [109] organize the LSM tree using tries, resulting in reduced write amplification. However, LSM-tries do not support range queries.

Other approaches. Researchers have also attempted to reduce write amplification by exploiting special hardware features such as flash translation layers [74] and vector interfaces [103]. VT-Tree [98] uses indirection to avoid copy data that is already sorted. "Trivial moves" are a similar idea is in RocksDB and PebblesDB. TRIAD [11] reduces write amplification by holding hot keys in memory, delaying compaction until different runs have significant key overlap, and by reducing redundancy between log and LSM tree writes. All these techniques are orthogonal to our work and can be used in conjunction with our techniques.

Concurrency is also an important aspect of key-value store performance. One of the first works in increasing concurrency in LSM-based stores was cLSM [53] which introduces a new compaction algorithm. Zuo et al. [114] show how to tune a cuckoo hash for NVM. Such a scheme suffers from high write amplification, since each insertion must re-write all keys in a data block. Zuo et al. do not report write amplification numbers but instead focus on concurrency.

Recent work on fast key-value stores includes GearDB [111], a key-value store that

avoids garbage collection on HM-SMR. Eisenman et al. [47] address the issue of large DRAM requirements of key-value stores for NVM via several techniques, including in-memory compression and NVM-specific caching schemes. Kourtis, et al. describe several systems-level optimizations for improving key-value-store throughput on NVMe, such as efficient use of user-level asynchronous I/O and low-latency scheduling [67]. Their techniques are largely orthogonal to the work in this paper. Kaiyrakhmet, et al. use persistent memory to simplify and improve performance relative to LevelDB [65].

Chapter 4

File System Aging

4.1 Introduction

File systems tend to become fragmented, or ***age***, as files are created, deleted, moved, appended to, and truncated [99, 76].

Fragmentation occurs when logically contiguous file blocks—either blocks from a large file or small files from the same directory—become scattered on disk. Reading these files requires additional seeks, and on hard drives, a few seeks can have an outsized effect on performance. For example, if a file system places a 100 MiB file in 200 disjoint pieces (i.e., 200 seeks) on a disk with $100\,\mathrm{MiB\,s^{-1}}$ bandwidth and 5 ms seek time, reading the data will take twice as long as reading it in an ideal layout. Even on SSDs, which do not perform mechanical seeks, a decline in logical block locality can harm performance [77].

The state of the art in mitigating aging applies best-effort heuristics at allocation time to avoid fragmentation. For example, file systems attempt to place related files close together on disk, while also leaving empty space for future files [76, 33, 102, 75]. Some file systems (including ext4, XFS, Btrfs, and F2FS among those tested in this paper) also include defragmentation tools that attempt to reorganize files and file blocks into contiguous regions to counteract aging.

Over the past two decades, there have been differing opinions about the significance of aging. The seminal work of Smith and Seltzer [99] showed that file systems age under realistic workloads, and this aging affects performance. On the other hand, there is a widely held view in the developer community that aging is a solved problem in production file systems. For example, the Linux System Administrator's Guide [108]

says:

> Modern Linux file systems keep fragmentation at a minimum by keeping all blocks in a file close together, even if they can't be stored in consecutive sectors. Some file systems, like ext3, effectively allocate the free block that is nearest to other blocks in a file. Therefore it is not necessary to worry about fragmentation in a Linux system.

There have also been changes in storage technology and file system design that could substantially affect aging. For example, a back-of-the-envelope analysis suggests that aging should get worse as rotating disks get bigger, as seek times have been relatively stable, but bandwidth grows (approximately) as the square root of the capacity. Consider the same level of fragmentation as the above example, but on a new, faster disk with 600MiB/s bandwidth but still a 5ms seek time. Then the 200 seeks would introduce four-fold slowdown rather than a two-fold slowdown. Thus, we expect fragmentation to become an increasingly significant problem as the gap between random I/O and sequential I/O grows.

As for SSDs, there is a widespread belief that fragmentation is not an issue. For example, PCWorld measured the performance gains from defragmenting an NTFS file system on SSDs[1], and concluded that, "From my limited tests, I'm firmly convinced that the tiny difference that even the best SSD defragger makes is not worth reducing the life span of your SSD."

In this paper, we revisit the issue of file system aging in light of changes in storage hardware, file system design, and data-structure theory. We make several contributions: (1) We give a simple, fast, and portable method for aging file systems. (2) We show that fragmentation over time (i.e., aging) is a first-order performance concern, and that this is true even on modern hardware, such as SSDs, and on modern file systems. (3) Furthermore, we show that aging is not inevitable. We present several techniques for avoiding aging. We show that BetrFS [62, 112, 61, 48], a research prototype that includes several of these design techniques, is much more resistant to aging than the other file systems we tested. In fact, BetrFS essentially did not age in our experiments,

establishing that aging is a solvable problem.

Results.

We use realistic application workloads to age five widely-used file systems—Btrfs [93], ext4 [33, 102, 75], F2FS [69], XFS [100] and ZFS [25]—as well as the BetrFS research file system. One workload ages the file system by performing successive git checkouts of the Linux kernel source, emulating the aging that a developer might experience on her workstation. A second workload ages the file system by running a mail-server benchmark, emulating aging over continued use of the server.

We evaluate the impact of aging as follows. We periodically stop the aging workload and measure the overall read throughput of the file system—greater fragmentation will result in slower read throughput. To isolate the impact of aging, as opposed to performance degradation due to changes in, say, the distribution of file sizes, we then copy the file system onto a fresh partition, essentially producing a defragmented or "unaged" version of the file system, and perform the same measurement. We treat the differences in read throughput between the aged and unaged copies as the result of aging.

We find that:

- All the production file systems age on both rotating disks and SSDs. For example, under our git workload, we observe over 50× slowdowns on hard disks and 2–5× slowdowns on SSDs. Similarly, our mail-server slows down 4–30× on HDDs due to aging.

- Aging can happen quickly. For example, ext4 shows over a 2× slowdown after 100 git pulls; Btrfs and ZFS slow down similarly after 300 pulls.

- BetrFS exhibits essentially no aging. Other than Btrfs, BetrFS's aged performance is better than the other file systems' unaged performance on almost all benchmarks. For instance, on our mail-server workload, unaged ext4 is 6× slower than aged BetrFS.

- The costs of aging can be staggering in concrete terms. For example, at the end of our git workload on an HDD, all four production file systems took over 8 minutes to grep through 1GiB of data. Two of the four took over 25 minutes. BetrFS took 10 seconds.

We performed several microbenchmarks to dive into the causes of aging and found that performance in the production file systems was sensitive to numerous factors:

- If only 10% of files are created out of order relative to the directory structure (and therefore relative to a depth-first search of the directory tree), Btrfs, ext4, F2FS, XFS and ZFS cannot achieve a throughput of $5\,\mathrm{MiB\,s^{-1}}$. If the files are copied completely out of order, then of these only XFS significantly exceeds $1\,\mathrm{MiB\,s^{-1}}$. This need not be the case; BetrFS maintains a throughput of roughly $50\,\mathrm{MiB\,s^{-1}}$.

- If an application writes to a file in small chunks, then the file's blocks can end up scattered on disk, harming performance when reading the file back. For example, in a benchmark that appends 4 KiB chunks to 10 files in a round-robin fashion on a hard drive, Btrfs and F2FS realize 10 times lower read throughput than if each file is written completely, one at a time. ext4 and XFS are more stable but eventually age by a factor of 2. ZFS has relatively low throughput but did not age. BetrFS throughput is stable, at two thirds of full disk bandwidth throughout the test.

4.2 Related Work

Prior work on file system aging falls into three categories: techniques for artificially inducing aging, for measuring aging, and for mitigating aging.

4.2.1 Creating Aged File Systems

The seminal work of Smith and Seltzer [99] created a methodology for simulating and measuring aging on a file system—leading to more representative benchmark results than running on a new, empty file system. The study is based on data collected from

daily snapshots of over fifty real file systems from five servers over durations ranging from one to three years. An overarching goal of Smith and Seltzer's work was to evaluate file systems with representative levels of aging.

Other tools have been subsequently developed for synthetically aging a file system. In order to measure NFS performance, TBBT [113] was designed to synthetically age a disk to create an initial state for NFS trace replay.

The Impressions framework [6] was designed so that users can synthetically age a file system by setting a small number of parameters, such as the organization of the directory hierarchy. Impressions also lets users specify a target layout score for the resulting image.

Both TBBT and Impressions create file systems with a specific level of fragmentation, whereas our study identifies realistic workloads that induce fragmentation.

4.2.2 Measuring Aged File Systems

Smith and Seltzer also introduced a ***layout score*** for studying aging, which was used by subsequent studies [8, 6]. Their layout score is the fraction of file blocks that are placed in consecutive physical locations on the disk. We introduce a variation of this measure, the ***dynamic layout score***, in section 4.4.

The ***degree of fragmentation*** (***DoF***) is used in the study of fragmentation in mobile devices [63]. DoF is the ratio of the actual number of extents, or ranges of contiguous physical blocks, to the ideal number of extents. Both the layout score and DoF measure how one file is fragmented.

Several studies have reported file system statistics such as number of files, distributions of file sizes and types, and organization of file system namespaces [7, 46, 94]. These statistics can inform parameter choices in aging frameworks like TBBT and Impressions [113, 6].

Feature	Btrfs	ext4	F2FS	XFS	ZFS	BetrFS
Grouped allocation within directories	✓	✓		✓	✓	✓
Extents	✓	✓		✓	✓	
Delayed allocation	✓	✓	✓	✓	✓	✓
Packing small files and metadata	✓ (by OID)					✓
Default Node Size	16 K	4 K	4 K	4 K	8 K	2–4 M
Maximum Node Size	64 K	64 K	4 K	64 K	128 K	2–4 M
Rewriting for locality						✓
Batching writes to reduce amplification			✓			✓

Table 4.1: Principal anti-aging features of the file systems measured in this paper. The top portion of the table are commonly-deployed features, and the bottom portion indicates features our model (section 3.7.4) indicates are essential; an ideal node size should match the natural transfer size, which is roughly 4 MiB for modern HDDs and SSDs. OID in Btrfs is an object identifier, roughly corresponding to an inode number, which is assigned at creation time.

4.2.3 Existing Strategies to Mitigate Aging

When files are created or extended, blocks must be allocated to store the new data. Especially when data is rarely or never relocated, as in an update-in-place file system like ext4, initial block allocation decisions determine performance over the life of the file system. Here we outline a few of the strategies use in modern file systems to address aging, primarily at allocation-time (also in the top of table 4.1).

Cylinder or Block Groups.

FFS [76] introduced the idea of ***cylinder groups***, which later evolved into block groups or allocation groups (XFS). Each group maintains information about its inodes and a bitmap of blocks. A new directory is placed in the cylinder group that contains more than the average number of free inodes, while inodes and data blocks of files in one directory are placed in the same cylinder group when possible.

ZFS [25] is designed to pool storage across multiple devies [25]. ZFS selects from one of a few hundred ***metaslabs*** on a device, based on a weighted calculation of several factors including minimizing seek distances. The metaslab with the highest weight is chosen.

In the case of F2FS [69], a log-structured file system, the disk is divided into

segments—the granularity at which the log is garbage collected, or cleaned. The primary locality-related optimization in F2FS is that writes are grouped to improve locality, and dirty segments are filled before finding another segment to write to. In other words, writes with temporal locality are more likely to be placed with physical locality.

Groups are a best-effort approach to directory locality: space is reserved for co-locating files in the same directory, but when space is exhausted, files in the same directory can be scattered across the disk. Similarly, if a file is renamed, it is not physically moved to a new group.

Extents.

All of the file systems we measure, except F2FS and BetrFS, allocate space using ***extents***, or runs of physically contiguous blocks. In ext4 [33, 102, 75], for example, an extent can be up to 128 MiB. Extents reduce bookkeeping overheads (storing a range versus an exhaustive list of blocks). Heuristics to select larger extents can improve locality of large files. For instance, ZFS selects from available extents in a metaslab using a first-fit policy.

Delayed Allocation.

Most modern file systems, including ext4, XFS, Btrfs, and ZFS, implement delayed allocation, where logical blocks are not allocated until buffers are written to disk. By delaying allocation when a file is growing, the file system can allocate a larger extent for data appended to the same file. However, allocations can only be delayed so long without violating durability and/or consistency requirements; a typical file system ensures data is dirty no longer than a few seconds. Thus, delaying an allocation only improves locality inasmuch as adjacent data is also written on the same timescale; delayed allocation alone cannot prevent fragmentation when data is added or removed over larger timescales.

Application developers may also request a persistent preallocation of contiguous blocks using fallocate. To take full advantage of this interface, developers must know each file's size in advance. Furthermore, fallocate can only help intrafile fragmentation;

there is currently not an analogous interface to ensure directory locality.

Packing small files and metadata.

For directories with many small files, an important optimization can be to pack the file contents, and potentially metadata, into a small number of blocks or extents. Btrfs [93] stores metadata of files and directories in copy-on-write B-trees. Small files are broken into one or more fragments, which are packed inside the B-trees. For small files, the fragments are indexed by object identifier (comparable to inode number); the locality of a directory with multiple small files depends upon the proximity of the object identifiers.

BetrFS stores metadata and data as key-value pairs in two B^ε-trees. Nodes in a B^ε-tree are large (2–4 MiB), amortizing seek costs. Key/value pairs are packed within a node by sort-order, and nodes are periodically rewritten, copy-on-write, as changes are applied in batches.

BetrFS also divides the namespace of the file system into ***zones*** of a desired size (512 KiB by default), in order to maintain locality within a directory as well as implement efficient renames. Each zone root is either a single, large file, or a subdirectory of small files. The key for a file or directory is its relative path to its zone root. The key/value pairs in a zone are contiguous, thereby maintaining locality.

4.3 A Framework for Aging

4.3.1 Natural Transfer Size

Our model of aging is based on the observation that the bandwidth of many types of hardware is maximized when I/Os are large; that is, sequential I/Os are faster than random I/Os. We abstract away from the particulars of the storage hardware by defining the ***natural transfer size*** (NTS) to be the amount of sequential data that must be transferred per I/O in order to obtain some fixed fraction of maximum throughput, say 50% or 90%. Reads that involve more than the NTS of a device will run near bandwidth.

From fig. 4.1, which plots SSD and HDD bandwidth as a function of read size, we

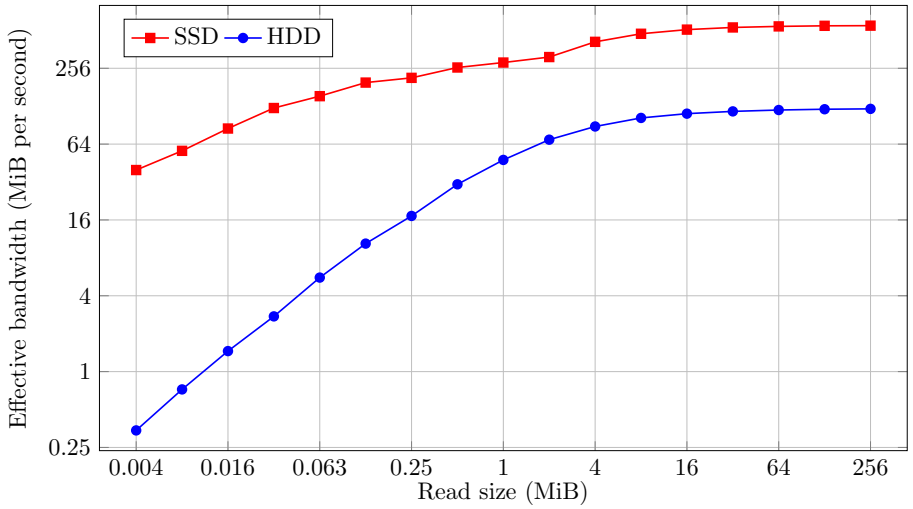


Figure 4.1: Effective bandwidth vs. read size (log-log scale, higher is better). Even on SSDs, large I/Os can yield an order of magnitude more bandwidth than small I/Os.

conclude that a reasonable NTS for both the SSDs and HDDs we measured is 4MiB.

The cause of the gap between sequential- and random-I/O speeds differs for different hardware. For HDDs, seek times offer a simple explanation. For SSDs, this gap is hard to explain conclusively without vendor support, but common theories include: sequential accesses are easier to stripe across internal banks, better leveraging parallelism [64]; some FTL translation data structures have nonuniform search times [73]; and fragmented SSDs are not able to prefetch data [35] or metadata [63]. Whatever the reason, SSDs show a gap between sequential and random reads, though not as great as on disks.

In order to avoid aging, file systems should avoid breaking large files into pieces significantly smaller than the NTS of the hardware. They should also group small files that are logically related (close in recursive traversal order) into clusters of size at least the NTS and store the clusters near each other on disk. We consider the major classes of file systems and explore the challenges each file system type encounters in achieving these two goals.

4.3.2 Allocation Strategies and Aging

The major file systems currently in use can be roughly categorized as B-tree-based, such as XFS, ZFS, and Btrfs, update-in-place, such as ext4, and log-structured, such as F2FS [69]. The research file system that we consider, BetrFS, is based on B^{ε}-trees. Each of these fundamental designs creates different aging considerations, discussed in turn below. In later sections, we present experimental validation for the design principles presented below.

B-trees.

The aging profile of a B-tree depends on the leaf size. If the leaves are much smaller than the NTS, then the B-tree will age as the leaves are split and merged, and thus moved around on the storage device.

Making leaves as large as the NTS increases *write amplification*, or the ratio between the amount of data changed and the amount of data written to storage. In the extreme case, a single-bit change to a B-tree leaf can cause the entire leaf to be rewritten. Thus, B-trees are usually implemented with small leaves. Consequently, we expect them to age under a wide variety of workloads.

In section 4.7, we show that the aging of Btrfs is inversely related to the size of the leaves, as predicted. There are, in theory, ways to mitigate the aging due to B-tree leaf movements. For example, the leaves could be stored in a packed memory array [14]. However, such an arrangement might well incur an unacceptable performance overhead to keep the leaves arranged in logical order, and we know of no examples of B-trees implemented with such leaf-arrangement algorithms.

Write-Once or Update-in-Place Filesystems.

When data is written once and never moved, such as in update-in-place file systems like ext4, sequential order is very difficult to maintain: imagine a workload that writes two files to disk, and then creates files that should logically occur between them. Without

moving one of the original files, data cannot be maintained sequentially. Such pathological cases abound, and the process is quite brittle. As noted above, delayed allocation is an attempt to mitigate the effects of such cases by batching writes and updates before committing them to the overall structure.

B$^\varepsilon$-trees.

B$^\varepsilon$-trees batch changes to the file system in a sequence of cascading logs, one per node of the tree. Each time a node overflows, it is flushed to the next node. The seeming disadvantage is that data is written many times, thus increasing the write amplification. However, each time a node is modified, it receives many changes, as opposed to B-tree, which might receive only one change. Thus, a B$^\varepsilon$-tree has asymptotically lower write amplification than a B-tree. Consequently, it can have much larger nodes, and typically does in implementation. BetrFS uses a B$^\varepsilon$-tree with 4MiB nodes.

Since 4MiB is around the NTS for our storage devices, we expect BetrFS not to age—which we verify below.

Log-structured merge trees (LSMs) [82] and other write-optimized dictionaries can resist aging, depending on the implementation. As with B$^\varepsilon$-trees, it is essential that node sizes match the NTS, the schema reflect logical access order, and enough writes are batched to avoid heavy write amplification.

4.4 Measuring File System Fragmentation

This section explains the two measures for file system fragmentation used in our evaluation: recursive scan latency and dynamic layout score, a modified form of Smith and Seltzer's layout score [99]. These measures are designed to capture both intra-file fragmentation and inter-file fragmentation.

Recursive grep test.

One measure we present in the following sections is the wall-clock time required to perform a recursive grep in the root directory of the file system. This captures the effects

of both inter- and intra-file locality, as it searches both large files and large directories containing many small files. We report search time per unit of data, normalizing by using ext4's du output. We will refer to this as the grep test.

Dynamic layout score.

Smith and Seltzer's layout score [99] measures the fraction of blocks in a file or (in aggregate) a file system that are allocated in a contiguous sequence in the logical block space. We extend this score to the dynamic I/O patterns of a file system. During a given workload, we capture the logical block requests made by the file system, using blktrace, and measure the fraction that are contiguous. This approach captures the impact of placement decisions on a file system's access patterns, including the impact of metadata accesses or accesses that span files. A high dynamic layout score indicates good data and metadata locality, and an efficient on-disk organization for a given workload.

One potential shortcoming of this measure is that it does not distinguish between small and large discontiguities. Small discontiguities on a hard drive should induce fewer expensive mechanical seeks than large discontiguities in general, however factors such as track length, difference in angular placement and other geometric considerations can complicate this relationship. A more sophisticated measure of layout might be more predictive. We leave this for further research. On SSD, we have found that the length of discontiguities has a smaller effect. Thus we will show that dynamic layout score strongly correlates with grep test performance on SSD and moderately correlates on hard drive.

4.5 Experimental Setup

Each experiment compares several file systems: BetrFS, Btrfs, ext4, F2FS, XFS, and ZFS. We use the versions of XFS, Btrfs, ext4 and F2FS that are part of the 3.11.10 kernel, and ZFS 0.6.5-234_ge0ab3ab, downloaded from the zfsonlinux repository on `www.github.com`. We used BetrFS 0.3 in the experiments[1]. We use default recommended

[1]Available at `github.com/oscarlab/betrfs`

file system settings unless otherwise noted. Lazy inode table and journal initialization are turned off on ext4, pushing more work onto file system creation time and reducing experimental noise.

All experimental results are collected on a Dell Optiplex 790 with a 4-core 3.40 GHz Intel Core i7 CPU, 4 GB RAM, a 500 GB, 7200 RPM ATA Seagate Barracuda ST500DM002 disk with a 4096 B block size, and a 240 GB Sandisk Extreme Pro—both disks used SATA 3.0. Each file system's block size is set to 4096 B. Unless otherwise noted, all experiments are cold-cache.

The system runs 64-bit Ubuntu 13.10 server with Linux kernel version 3.11.10 on a bootable USB stick. All HDD tests are performed on two 20GiB partitions located at the outermost region of the drive. For the SSD tests, we additionally partition the remainder of the drive and fill it with random data, although we have preliminary data that indicates this does not affect performance.

4.6 Fragmentation Microbenchmarks

We present several simple microbechmarks, each designed around a write/update pattern for which it is difficult to ensure both fast writes in the moment and future locality. These microbenchmarks isolate and highlight the effects of both intra-file fragmentation and inter-file fragmentation and show the performance impact aging can have on read performance in the worst cases.

Intrafile Fragmentation.

When a file grows, there may not be room to store the new blocks with the old blocks on disk, and a single file's data may become scattered.

Our benchmark creates 10 files by first creating each file of an initial size, and then appending between 0 and 100 4KiB chunks of random data in a round-robin fashion until each file is 400KiB. In the first round the initial size is 400KiB, so each entire file is written sequentially, one at a time. In subsequent rounds, the initial size becomes smaller, so that the number of round-robin chunks increases until in the last round

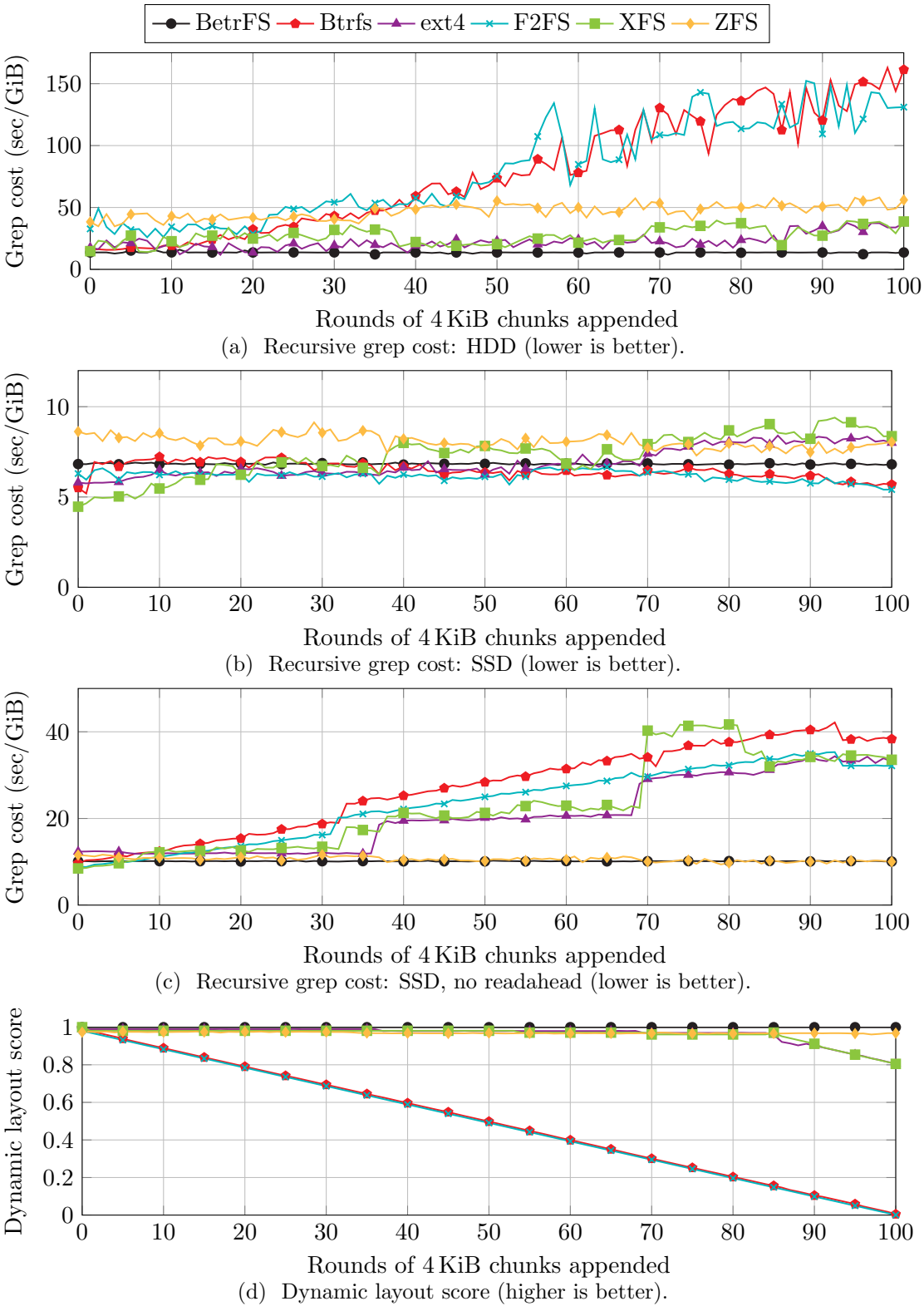
the data is written entirely with a round-robin of 4KiB chunks. After all the files are written, the disk cache is flushed by remounting, and we wait for 90 seconds before measuring read performance. Some file systems appear to perform background work immediately after mounting that introduced experimental noise; 90 seconds ensures the file system has quiesced.

The aging process this microbenchmark emulates is multiple files growing in length. The file system must allocate space for these files somewhere, but eventually the file must either be moved or will fragment.

Given that the data set size is small and the test is designed to run in a short time, an fsync is performed after each file is written in order to defeat deferred allocation. Similar results are obtained if the test waits for 5 seconds between each append operation. If fewer fsyncs are performed or less waiting time is used, then the performance differences are smaller, as the file systems are able to delay allocation, rendering a more contiguous layout.

The performance of these file systems on an HDD and SSD are summarized in fig. 4.2. On HDD, the layout scores generally correlate (-0.93) with the performance of the file systems. On SSD, the file systems all perform similarly (note the scale of the y-axis). In some cases, such as XFS, ext4, and ZFS, there is a correlation, albeit at a small scale. For Btrfs, ext4, XFS, and F2FS, the performance is hidden by read-ahead in the OS, or in the case of Btrfs, also in the file system itself. If we disable read-ahead, shown in fig. 4.2c, the performance is more clearly correlated ($-.67$) with layout score. We do note that this relationship on an SSD is still not precise; SSDs are sufficiently fast that factors such as CPU time can also have a significant effect on performance.

Because of the small amount of data and number of files involved in this microbenchmark, we can visualize the layout of the various file systems, shown in fig. 4.3. Each block of a file is represented by a small vertical bar, and each bar is colored uniquely to one of the ten files. Contiguous regions form a colored rectangle. The visualization suggests, for example, that ext4 both tries to keep files and eventually larger file fragments sequential, whereas Btrfs and F2FS interleave the round robin chunks on the end of the sequential data. This interleaving can help explain why Btrfs and F2FS perform



(a) Recursive grep cost: HDD (lower is better).

(b) Recursive grep cost: SSD (lower is better).

(c) Recursive grep cost: SSD, no readahead (lower is better).

(d) Dynamic layout score (higher is better).

Figure 4.2: Intrafile benchmark: 4KiB chunks are appended round-robin to sequential data to create 10 400KiB files. Dynamic layout scores generally correlate with read performance as measured by the recursive grep test; on an SSD, this effect is hidden by the readahead buffer.

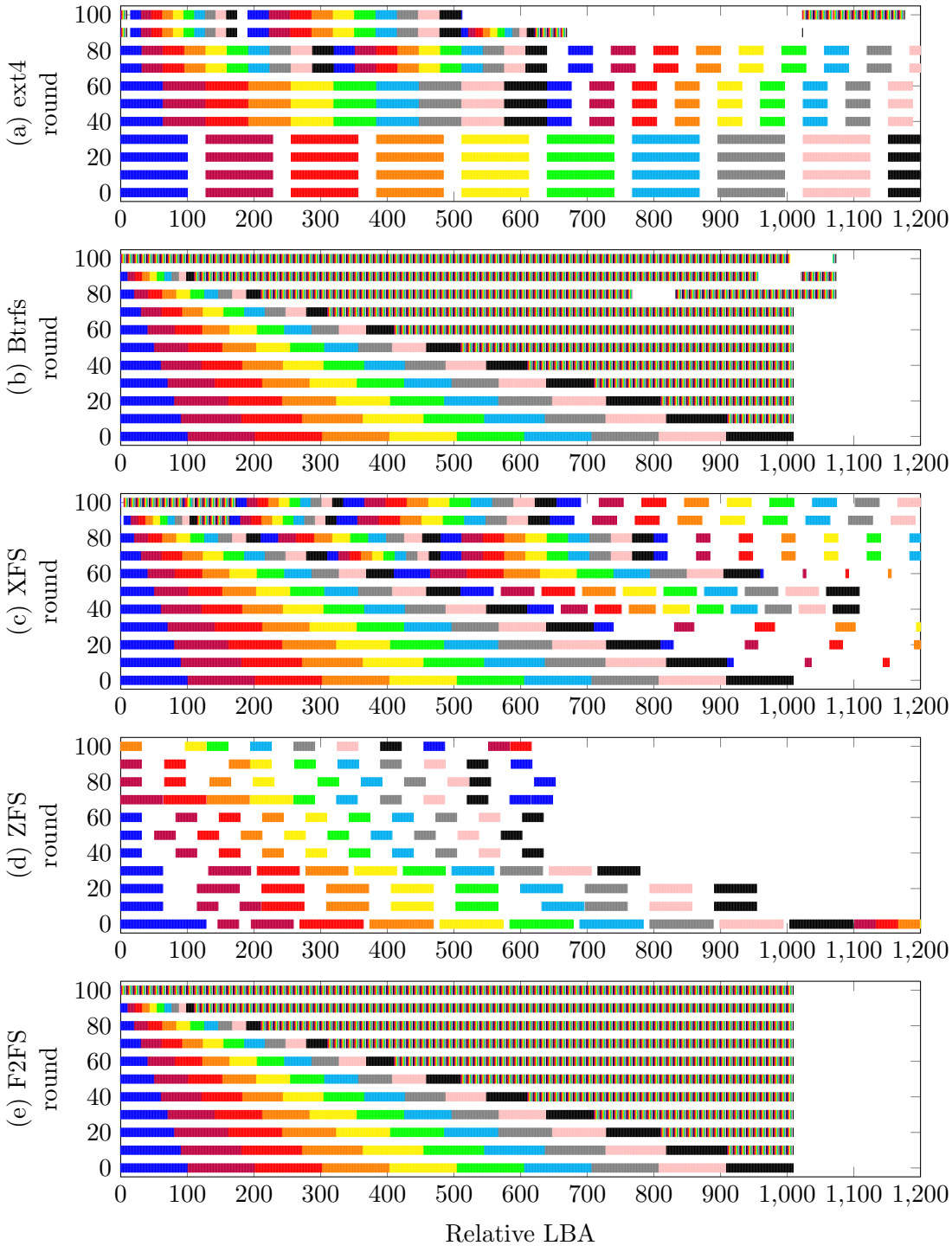


Figure 4.3: Intrafile benchmark layout visualization. Each color represents blocks of a file. The x-axis is the logical block address (LBA) of the file block relative to the first LBA of any file block, and y-axis is the round of the experiment. Rectangle sizes indicate contiguous placement, where larger is better. The brown regions with vertical lines indicate interleaved blocks of all 10 files. Some blocks are not shown for ext4, XFS and ZFS.

the way they do: the interleaved sections must be read through in full each time a file is requested, which by the end of the test takes roughly 10 times as long. ext4 and XFS manage to keep the files in larger extents, although the extents get smaller as the test progresses, and, by the end of the benchmark, these file systems also have chunks of interleaved data; this is why ext4 and XFS's dynamic layout scores decline. ZFS keeps the files in multiple chunks through the test; in doing so it sacrifices some performance in all states, but does not degrade.

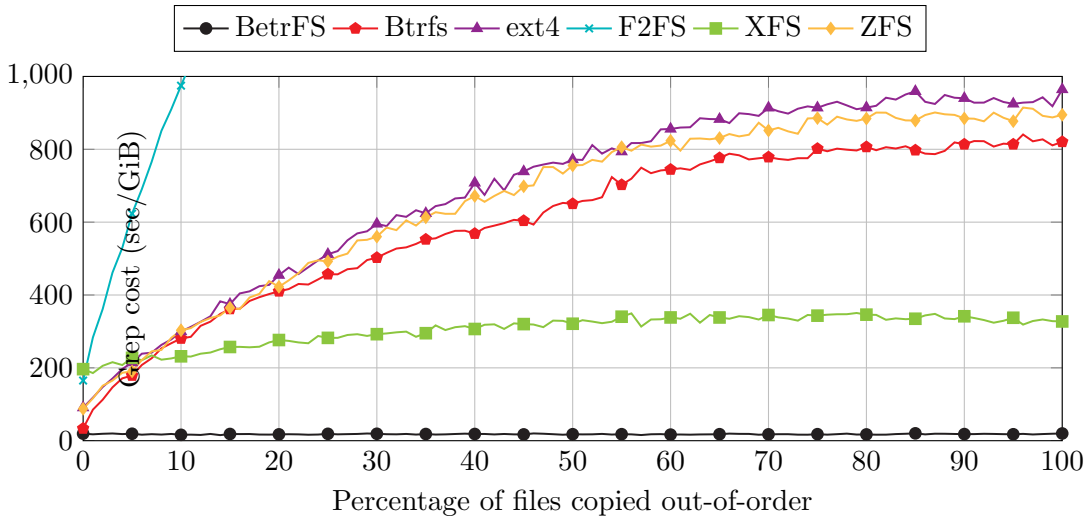
Unfortunately, this sort of visualization doesn't work for BetrFS, because this small amount of data fits entirely in a leaf. Thus, BetrFS will read all this data into memory in one sequential read. This results is some read amplification, but, on an HDD, only one seek.

Interfile Fragmentation.

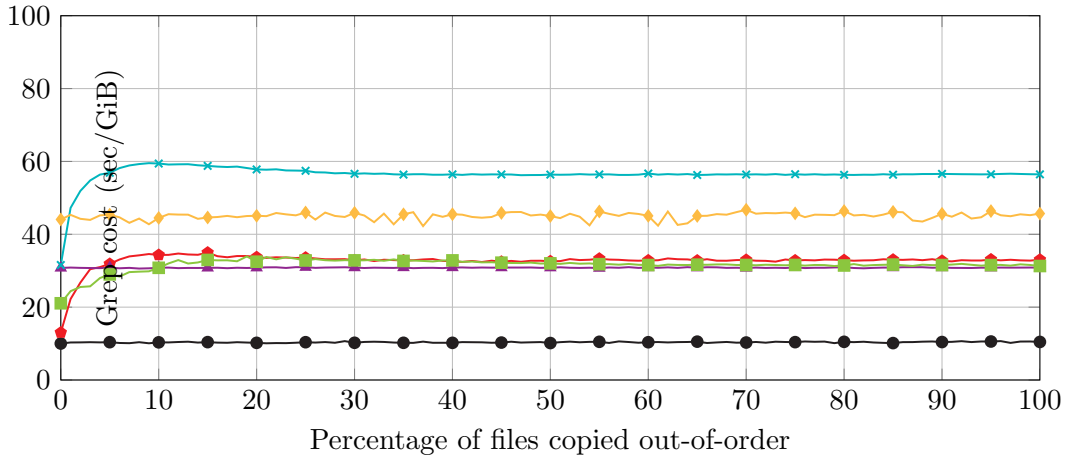
Many workloads read multiple files with some logical relationship, and frequently those files are placed in the same directory. Interfile fragmentation occurs when files which are related—in this case by being close together in the directory tree—are not collocated in the LBA space.

We present a microbenchmark to measure the impact of namespace creation order on interfile locality. It takes a given "real-life" file structure, in this case the Tensorflow repository obtained from `github.com`, and replaces each of the files by 4KiB of random data. This gives us a "natural" directory structure, but isolates the effect of file ordering without the influence of intrafile layout. The benchmark creates a sorted list of the files as well as two random permutations of that list. On each round of the test, the benchmark copies all of the files, creating directories as needed with `cp --parents`. However, on the nth round, it swaps the order in which the first $n\%$ of files appearing in the random permutations are copied. Thus, the first round will be an in-order copy, and subsequent rounds will be copied in a progressively more random order until the last round is a fully random-order copy.

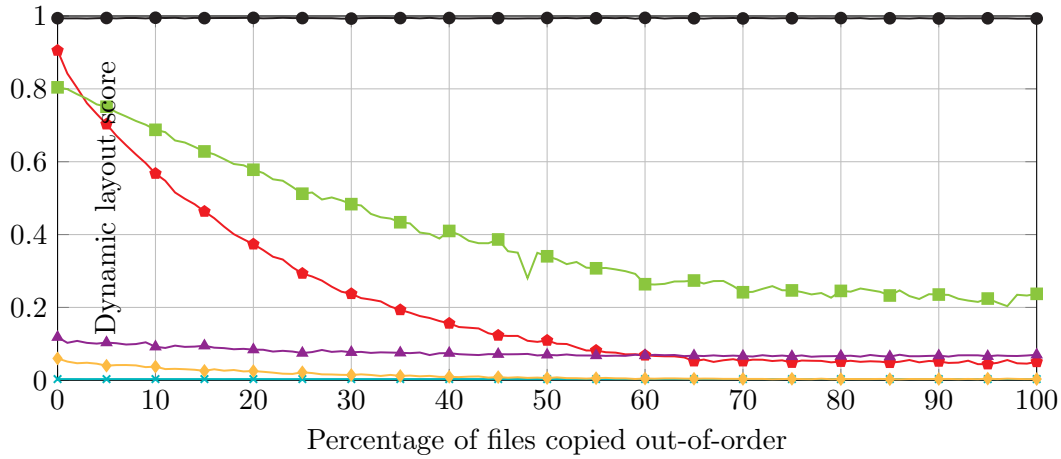
The results of this test are shown fig. 4.4. On hard drive, all the file systems except BetrFS and XFS show a precipitous performance decline even if only a small percentage



(a) Recursive grep cost: HDD (Lower is better).



(b) Recursive grep cost: SSD (Lower is better).



(c) Dynamic layout score (higher is better).

Figure 4.4: Interfile benchmark: The TensorFlow github repository with all files replaced by 4KiB random data and copied in varying degrees of order. Dynamic layout scores again are predictive of recursive grep test performance.

of the files are copied out of order. F2FS's performance is poor enough to be out of scale for this figure, but it ends up taking over 4000 seconds per GiB at round 100; this is not entirely unexpected as it is not designed to be used on hard drive. XFS is somewhat more stable, although it is 13-35 times slower than drive bandwidth throughout the test, even on an in-order copy. BetrFS consistently performs around 1/3 of bandwidth, which by the end of the test is 10 times faster than XFS, and 25 times faster than the other file systems. The dynamic layout scores are moderately correlated with this performance (-0.57).

On SSD, half the file systems perform stably throughout the test with varying degrees of performance. The other half have a very sharp slowdown between the in-order state and the 10% out-of-order state. These two modes are reflected in their dynamic layout scores as well. While ext4 and ZFS are stable, their performance is worse than the best cases of several other file systems. BetrFS is the only file system with stable fast performance; it is faster in every round than any other file system even in their best case: the in-order copy. In this cases the performance strongly correlates with the dynamic layout score (-0.83).

4.7 Application Level Read-Aging: Git

To measure aging in the "real-world," we create a workload designed to simulate a developer using git to work on a collaborative project.

Git is a distributed version control system that enables collaborating developers to synchronize their source code changes. Git users ***pull*** changes from other developers, which then get merged with their own changes. In a typical workload, a Git user may perform pulls multiple times per day over several years in a long-running project. Git can synchronize all types of file system changes, so performing a Git pull may result in the creation of new source files, deletion of old files, file renames, and file modifications. Git also maintains its own internal data structures, which it updates during pulls. Thus, Git performs many operations which are similar to those shown in section 4.6 that cause file system aging.

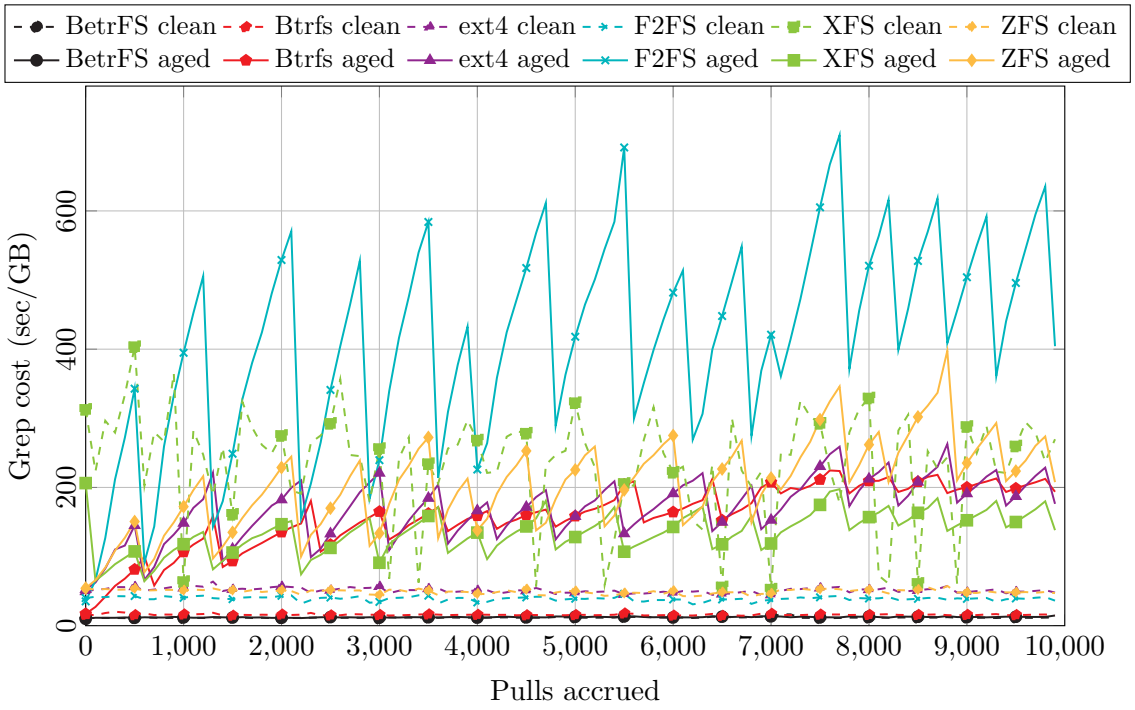
We present a git benchmark that performs 10,000 pulls from the Linux git repository, starting from the initial commit. After every 100 pulls, the benchmark performs a recursive grep test and computes the file system's dynamic layout score. This score is compared to the same contents copied to a freshly formatted partition.

On a hard disk (fig. 4.5), there is a clear aging trend in all file systems except BetrFS. By the end of the experiment, all the file systems except BetrFS show performance drops under aging on the order of at least 3x and as much as 15x relative to their unaged versions. All are at least 15x worse than BetrFS. The dynamic layout scores througout the benchmark are shown in fig. 4.7. In all of the experiments in this section, F2FS ages considerably more than all other file systems, commensurate with significantly lower layout scores than the other file systems—indicating less effective locality in data placement. The overall correlation between grep performance and dynamic layout score is moderate, at -0.41.

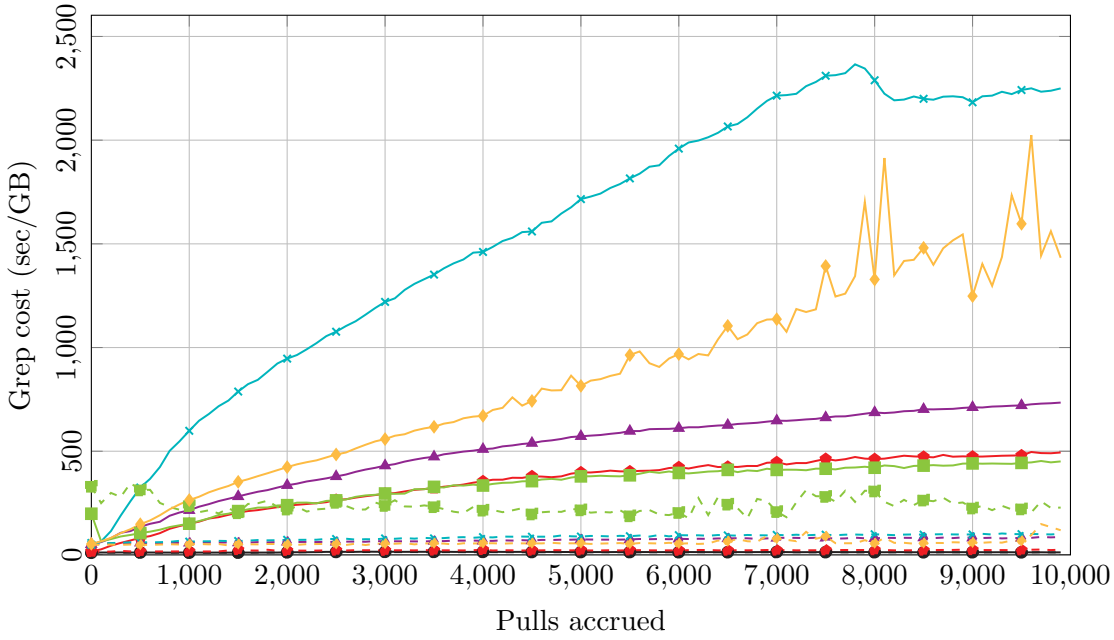
On an SSD (fig. 4.6), Btrfs and XFS show clear signs of aging, although they converge to a fully aged configuration after only about 1,000 pulls. While the effect is not as drastic as on HDD, in all the traditional file systems we see slowdowns of 2x-4x over BetrFS, which does not slow down. In fact, aged BetrFS on the HDD outperforms all the other aged file systems on an SSD, and is close even when they are unaged. Again, this performance decline is strongly correlated (-0.79) with the dynamic layout scores.

The aged and unaged performance of ext4 and ZFS are comparable, and slower than several other file systems. We believe this is because the average file size decreases over the course of the test, and these file systems are not as well-tuned for small files. To test this hypothesis, we constructed synthetic workloads similar to the interfile fragmentation microbenchmark (section 4.6), but varied the file size (in the microbenchmark it was uniformly 4KB).fig. 4.8 shows both the measured, average file size of the git workload (one point is one pull), and the microbenchmark. Overall, there is a clear relationship between the average file size and grep cost.

The zig-zag pattern in the graphs is created by an automatic garbage collection process in Git. Once a certain number of "loose objects" are created (in git terminology),

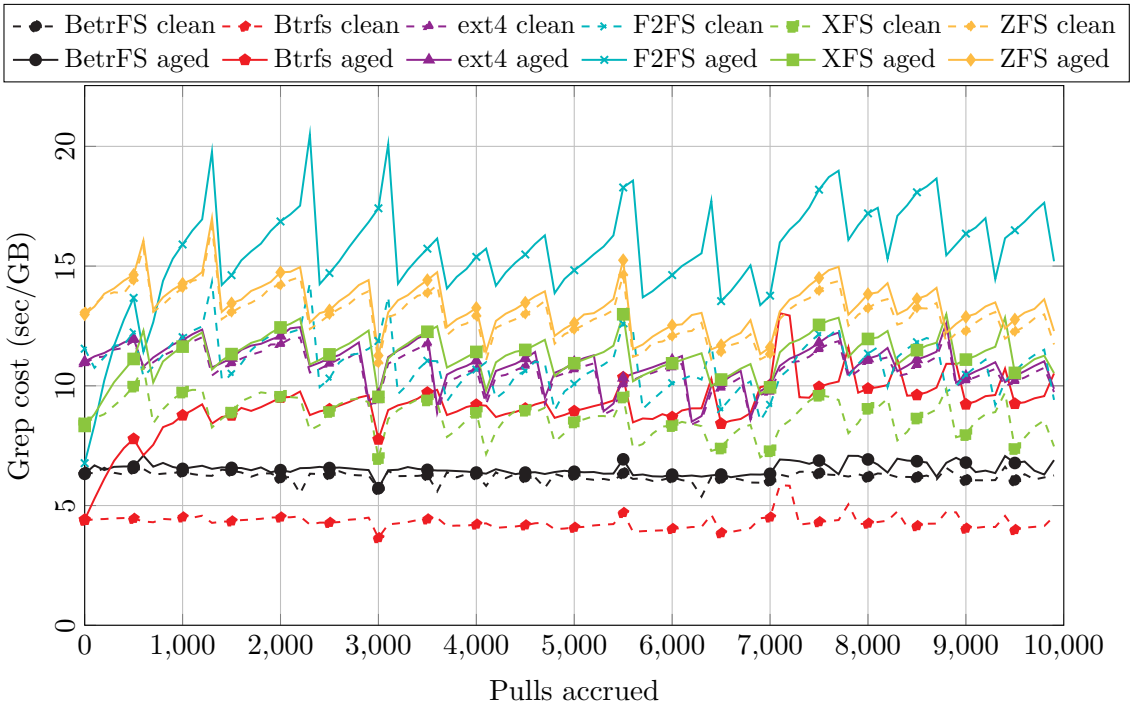


(a) HDD, git garbage collection on (Lower is better).

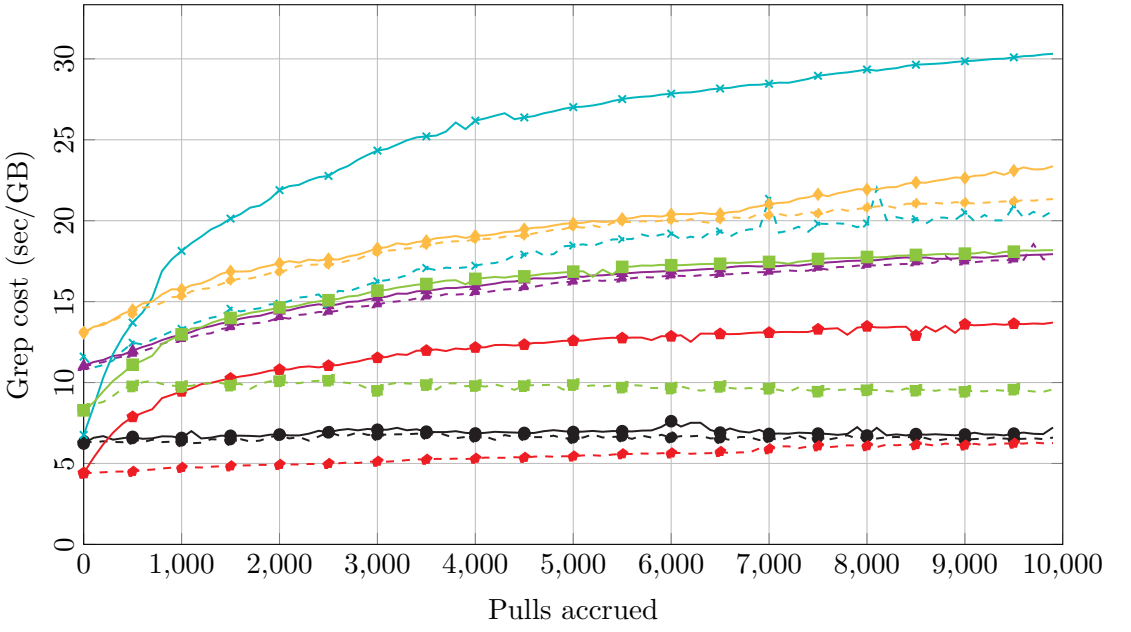


(b) HDD, git garbage collection off (Lower is better).

Figure 4.5: Git read-aging experimental results on HDD: On-disk layout as measured by dynamic layout score generally is predictive of read performance.

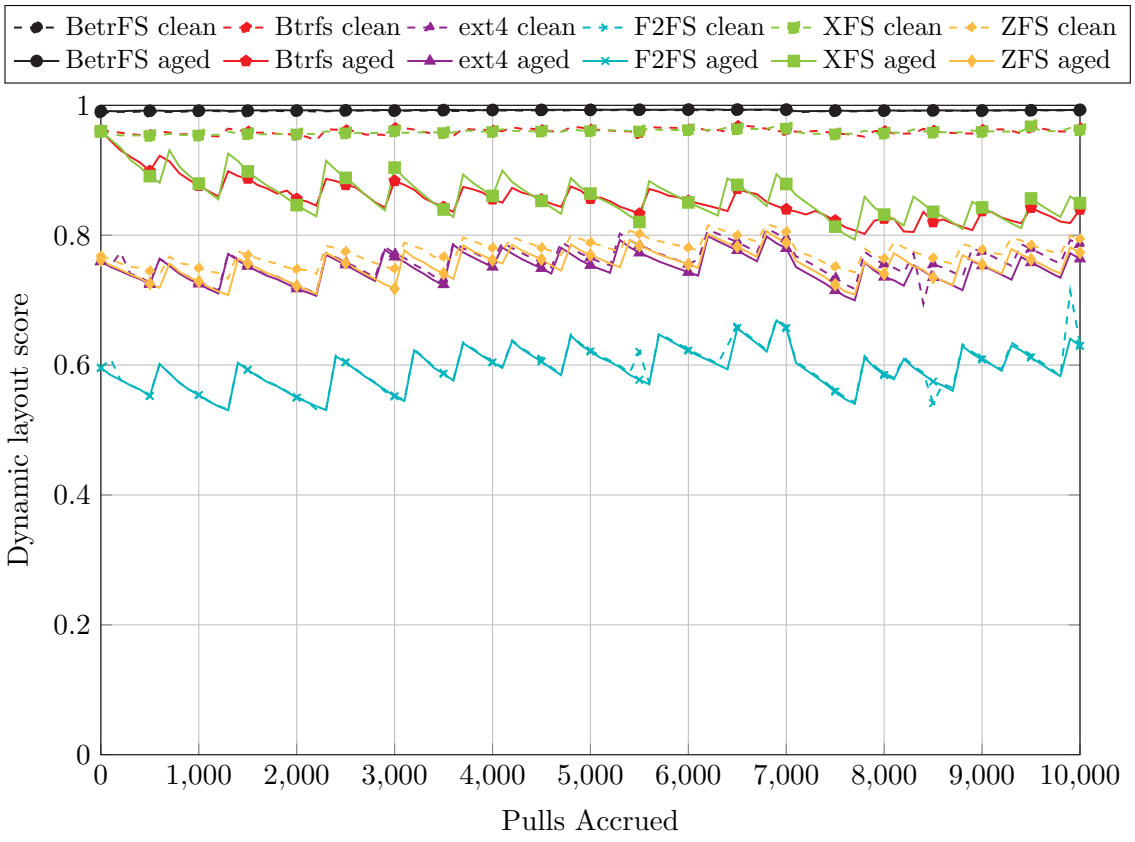


(a) SSD, git garbage collection on (Lower is better).

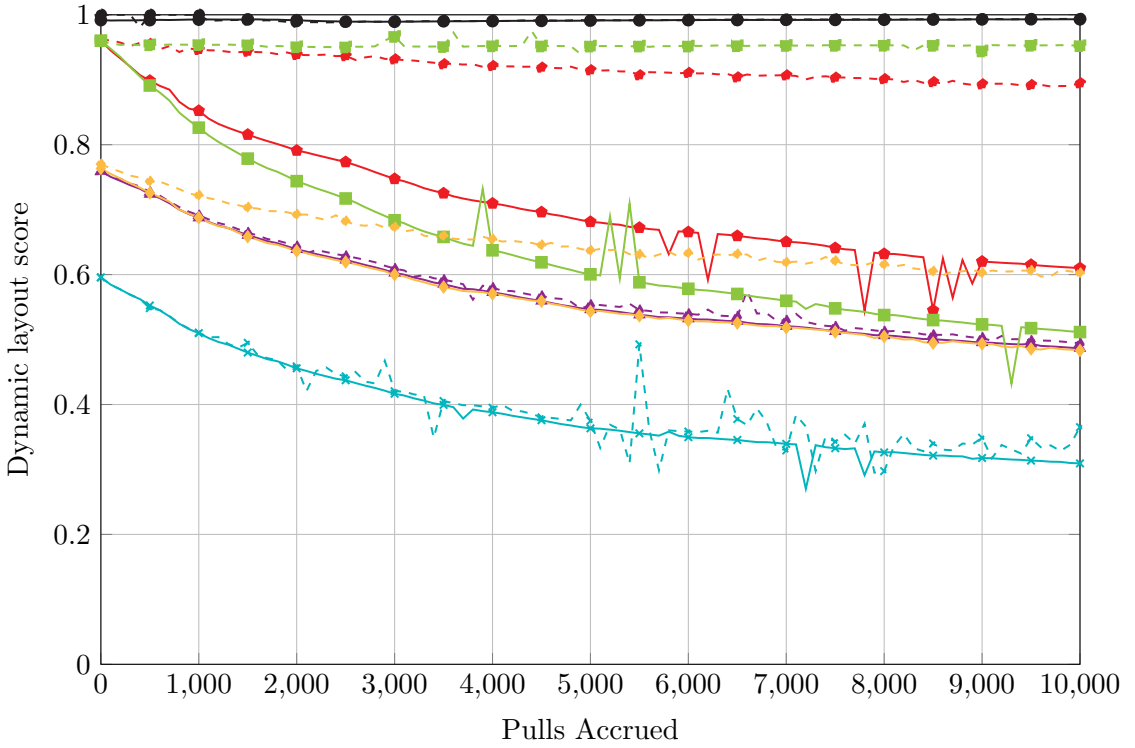


(b) SSD, git garbage collection off (Lower is better).

Figure 4.6: Git read-aging experimental results on SDD: On-disk layout as measured by dynamic layout score generally is predictive of read performance.



(a) Dynamic layout score: git garbage collection on (Higher is better).



(b) Dynamic layout score: git garbage collection off (Higher is better).

Figure 4.7: Git read-aging layout scores.

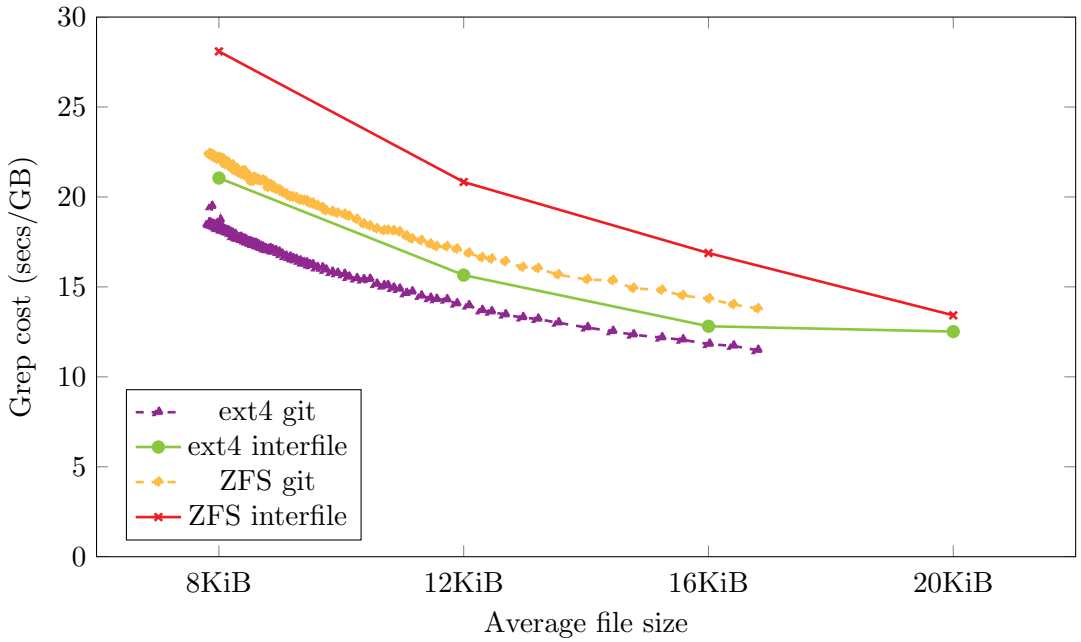


Figure 4.8: Average file size versus unaged grep costs (lower is better) on SSD. Each point in the git line is the average file size for the git experiment, compared to a microbenchmark with all files set to a given size.

many of them are collected and compressed into a "pack." At the file system level, this corresponds to merging numerous small files into a single large file. According to the Git manual, this process is designed to "reduce disk space and increase performance," so this is an example of an application-level attempt to mitigate file system aging. If we turn off the git garbage collection, as show in figs. 4.5b, 4.6b and 4.7b, the effect of aging is even more pronounced, and the zig-zags essentially disappear.

On both the HDD and SSD, the same patterns emerge as with garbage collection on, but exacerbated: F2FS aging is by far the most extreme. ZFS ages considerably on the HDD, but not on the SSD. ZFS on SSD and ext4 perform worse than the other file systems (except F2FS aged), but do not age particularly. XFS and Btrfs both aged significantly, around 2x each, and BetrFS has strong, level performance in both states. This performance correlates with dynamic layout score both on SSD (-0.78) and moderately so on HDD (-0.54).

We note that this analysis, both of the microbenchmarks and of the git workload, runs counter to the commonly held belief that locality is solely a hard drive issue. While

the random read performance of solid state drives does somewhat mitigate the aging effects, aging clearly has a major performance impact.

Git Workload with Warm Cache.

The tests we have presented so far have all been performed with a cold cache, so that they more or less directly test the performance of the file systems' on-disk layout under various aging conditions. In practice, however, some data will be in cache, and so it is natural to ask how much the layout choices that the file system makes will affect the overall performance with a warm cache.

We evaluate the sensitivity of the git workloads to varying amounts of system RAM. We use the same procedure as above, except that we do not flush any caches or remount the hard drive between iterations. This test is performed on a hard drive with git garbage collection off. The size of the data on disk is initially about 280MiB and grows throughout the test to approximately 1GiB.

The results are summarized in fig. 4.9. We present data for ext4 and F2FS; the results for Btrfs, XFS and ZFS are similar. BetrFS is a research prototype and unstable under memory pressure; although we plan to fix these issues in the future, we omit this comparison.

In general, when the caches are warm and there is sufficient memory to keep all the data in cache, then the read is very fast. However, as soon as there is no longer sufficient memory, the performance of the aged file system with a warm cache is generally worse than unaged with a cold cache. In general, unless all data fits into DRAM, a good layout matters more than a having a warm cache.

Git Workload on BTRFS with Different Node Sizes

We present the git test with a 4KiB node size, the default setting, as well as 8KiB, 16KiB, 32KiB, and 64KiB (the maximum). fig. 4.10a shows similar performance graphs to fig. 4.5, one line for each node size. The 4KiB node size has the worst read performance by the end of the test, and the performance consistently improves as we increase the node size all the way to 64KiB. fig. 4.10b plots the number of 4KiB blocks written

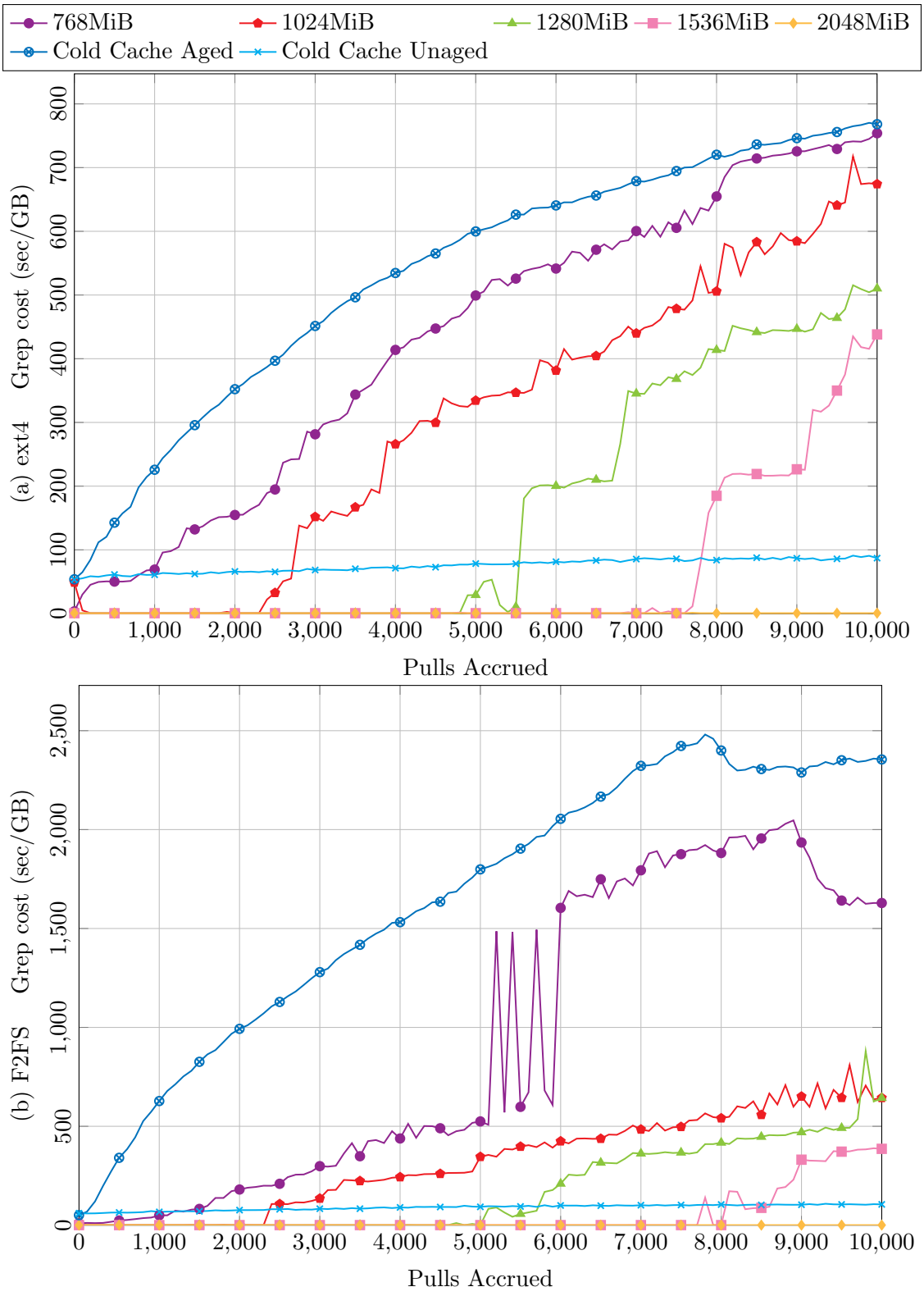


Figure 4.9: Grep costs as a function of git pulls with warm cache and varying system RAM on ext4 (top) and F2FS (bottom). Lower is better.

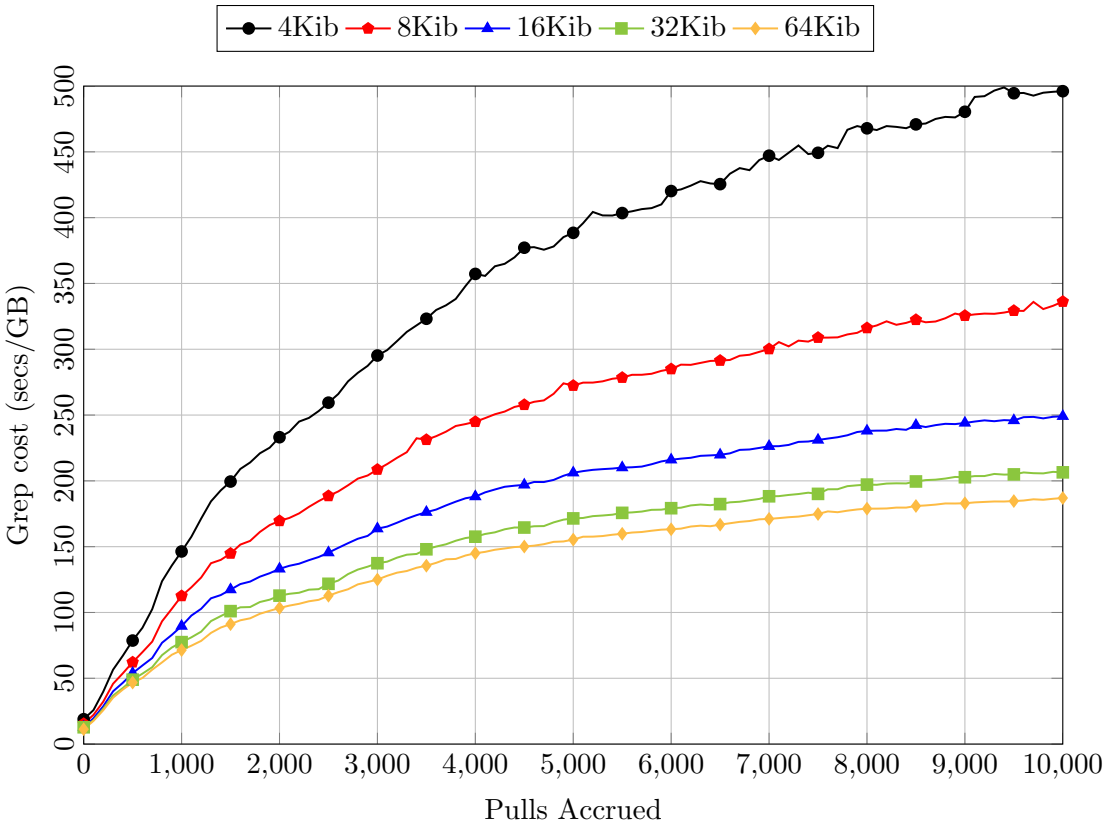
to disk between each test (within the 100 pulls). As expected, the 64KiB node size writes the maximum number of blocks and the 4KiB node writes the least. We thus demonstrate—as predicted by our model—that aging is reduced by a larger block size, but at the cost of write-amplification.

4.8 Application Level Aging: Mail Server

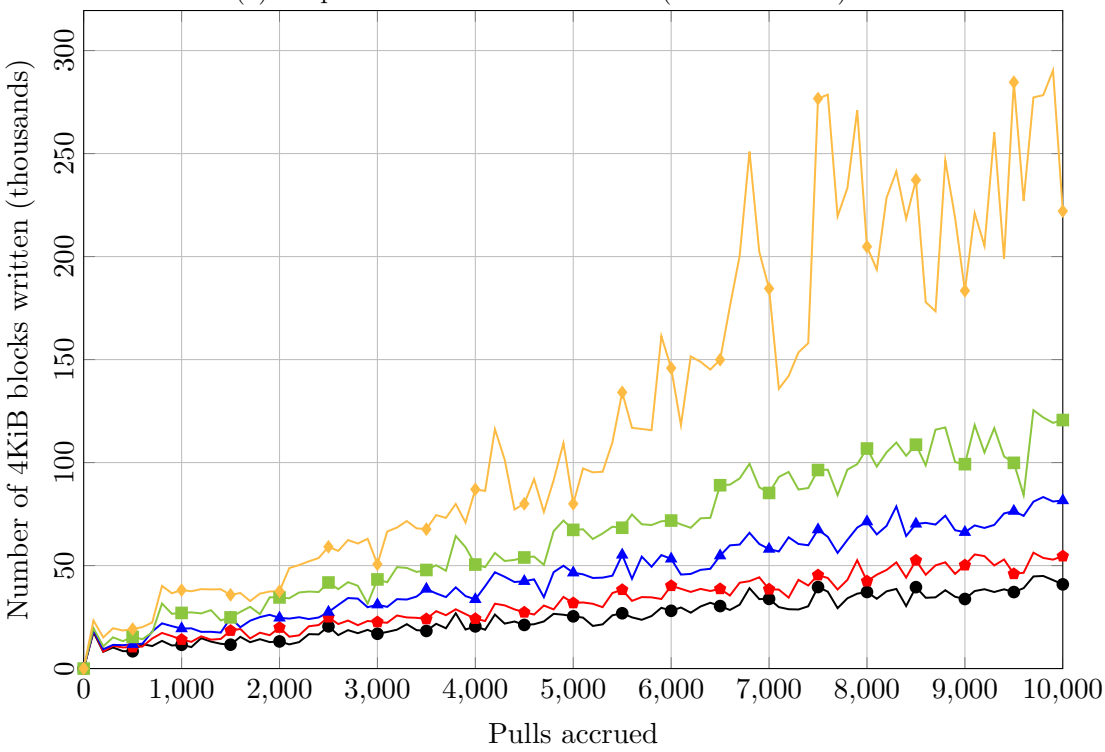
In addition to the git workload, we evaluate aging with the Dovecot email server. Dovecot is configured with the Maildir backend, which stores each message in a file, and each inbox in a directory. We simulate 2 users, each having 80 mailboxes, receiving new email, deleting old emails, and searching through their mailboxes.

A cycle or "day" for the mailserver comprises 8,000 operations, where each operation is equally likely to be an insert or a delete, corresponding to receiving a new email or deleting an old one. Each email is a string of random characters, the length of which is uniformly distributed over the range [1, 32K]. Each mailbox is initialized with 1,000 messages, and, because inserts and deletes are balanced, mailbox size tends to stay around 1,000. We simulate the mailserver for 100 cycles and after each cycle we perform a recursive grep for a random string. As in our git benchmarks, we then copy the partition to a freshly formatted file system, and run a recursive grep.

fig. 4.11 shows the read costs in seconds per GiB of the grep test on hard disk. Although the unaged versions of all file systems show consistent performance over the life of the benchmark, the aged versions of ext4, Btrfs, XFS and ZFS all show significant degradation over time. In particular, aged ext4 performance degrades by 4.4×, and is 28× slower than aged BetrFS. XFS slows down by a factor of 7 and Btrfs, by a factor of 30. ZFS slows down drastically, taking about 20 minutes per GiB by cycle 20. However, the aged version of BetrFS does not slow down. As with the other HDD experiments, dynamic layout score is moderately correlated (-0.63) with grep cost.

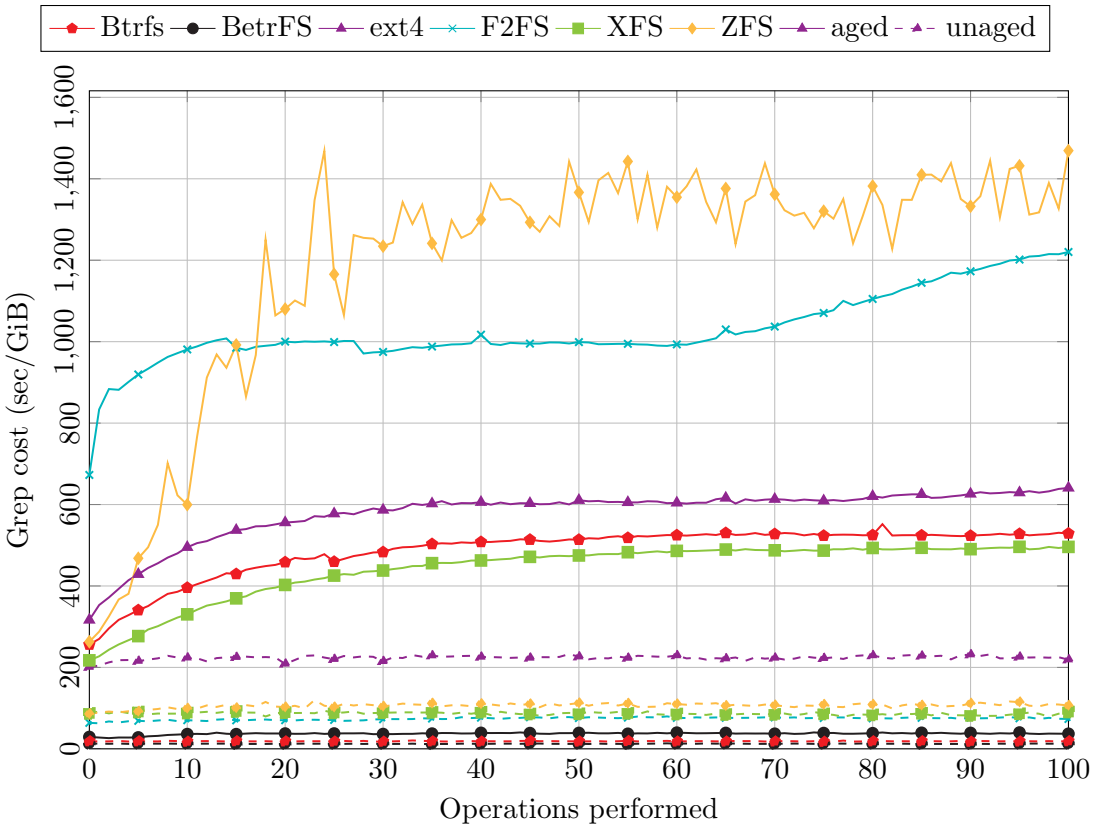


(a) Grep cost at different node sizes (lower is better).

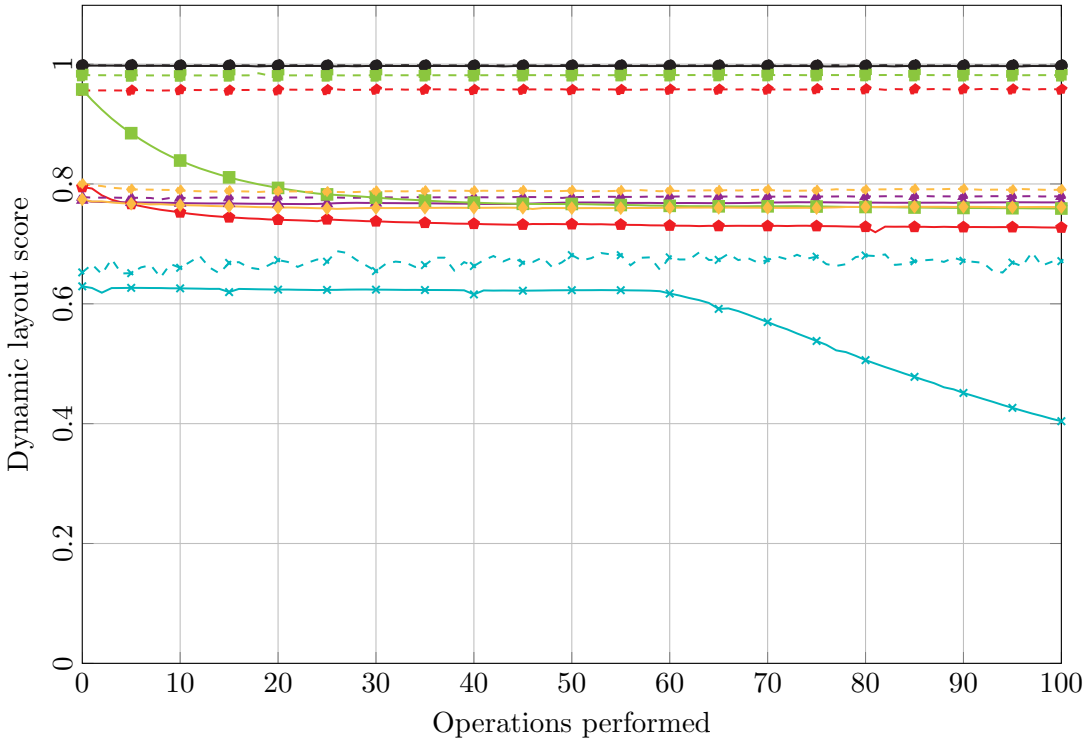


(b) Write amplification at different node sizes (lower is better).

Figure 4.10: Aging and write amplification on Btrfs, with varying node sizes, under the git aging benchmark.



(a) Grep cost during mailserver workload (lower is better).



(b) Mailserver layout (higher is better).

Figure 4.11: Mailserver performance and layout scores.

4.9 Conclusion

The experiments above suggest that the conventional wisdom on fragmentation, aging, allocation and file systems is inadequate in several ways.

First, while it may seem intuitive to write data as few times as possible, writing data only once creates a tension between the logical ordering of the file system's current state and the potential to make modifications without disrupting the future order. Rewriting data multiple times allows the file system to maintain locality. The overhead of these multiple writes can be managed by rewriting data in batches, as is done in write-optimized dictionaries.

For example, in BetrFS, data might be written as many as a logarithmic number of times, whereas in ext4, it will be written once, yet BetrFS is able in general to perform as well as or better than an unaged ext4 file system and significantly outperforms aged ext4 file systems.

Second, today's file system heuristics are not able to maintain enough locality to enable reads to be performed at the disk's natural transfer size. And since the natural transfer size on a rotating disk is a function of the seek time and bandwidth, it will tend to increase with time. Thus we expect this problem to possibly become worse with newer hardware, not better.

We experimentally confirmed our expectation that non-write-optimized file systems would age, but we were surprised by how quickly and dramatically aging impacts performance. This rapid aging is important: a user's experience with unaged file systems is likely so fleeting that they do not notice performance degradation. Instead, the performance costs of aging are built into their expectations of file system performance.

Finally, because representative aging is a difficult goal, simulating multi-year workloads, many research papers benchmark on unaged file systems. Our results indicate that it is relatively easy to quickly drive a file system into an aged state—even if this state is not precisely the state of the file system after, say, three years of typical use—and this degraded state can be easily measured.

Chapter 5

Optimal Ball Recycling

5.1 Introduction

Balls-and-bins games have been a successful tool for modeling load balancing problems [80, 3, 2, 9, 36, 37, 41, 78, 79, 105, 27, 88, 106, 49, 39]. For example, they can be used to study the average and worst-case occupancy of buckets in a hash table [21], the worst-case load on nodes in a distributed cluster [89, 22] and even the amount of time customers wait in line at the grocery store [79]. In all these load-balancing problems, balls-and-bins games are used to study how to distribute load evenly across the resource being allocated.

In this paper we study a new scenario, which we refer to as the ***ball-recycling game***, defined as follows:

> Throw m balls into n bins i.i.d. according to a given probability distribution $\mathbf{p}$. Then, at each time step, pick a non-empty bin and ***recycle*** its balls: take the balls from the selected bin and re-throw them according to $\mathbf{p}$.

We call a bin-picking method a ***recycling strategy*** and define its ***recycling rate*** to be the expected number of balls recycled in the stationary distribution (when it exists).

The ball-recycling game models ***insertion buffers*** and ***update buffers***, which are widely used to speed up insertions in databases by batching updates to blocks on disk. The recycling rate of a recycling strategy corresponds to the speed-up obtained by an insertion buffer, so the goal studied in this paper is how to maximize the recycling rate. This relationship is described in section 5.2, and the experiments in section 5.6 demonstrate that it holds in practice.

In this paper, we present results for ball recycling for both general $\mathbf{p}$ and for the

special case of uniform $\mathbf{p}$, which we denote by $\mathbf{u}$. As we explain in section 5.2, these distributions correspond to update and insertion buffers, respectively.

We focus on three natural recycling strategies:

- FULLEST BIN: A greedy strategy that recycles the bin with the most balls.
- RANDOM BALL: A strategy that picks a ball uniformly at random and recycles its bin.
- GOLDEN GATE: A strategy that picks the bins in round-robin fashion; after a bin is picked, the next bin picked is its non-empty successor.

Let $\|\mathbf{p}\|_{\frac{1}{2}} = (\sum \sqrt{\mathbf{p}_i})^2$ be the half quasi-norm of $\mathbf{p}$. We achieve the following result for general $\mathbf{p}$.

Theorem 6 (section 5.4)**.** *Consider a ball-recycling game with m balls and n bins, where the balls are distributed into the bins i.i.d. according to distribution $\mathbf{p}$. Then* RANDOM BALL *is $\Theta(1)$-optimal.*

It achieves recycling rate $\mathcal{R}^{RB}$:

1. *If $m \geq n$,*
$$\mathcal{R}^{RB} = \Theta\left(\frac{m}{\|\mathbf{p}\|_{\frac{1}{2}}}\right).$$
2. *If $m < n$, let L be the m lowest-weight bins, $q = \sum_{\ell \in L} p_\ell$, and $\mathcal{R}_L^{RB}$ be the recycle rate of* RANDOM BALL *restricted to L. Then,*
$$\mathcal{R}^{RB} = \Theta\left(\min\left(\mathcal{R}_L^{RB}, 1/q\right)\right).$$

In order to establish this result, we first show that no recycling strategy can achieve a higher recycling rate than $(2m+n)/\|\mathbf{p}\|_{\frac{1}{2}}$. This directly establishes optimality when $m = \Omega(n)$. For $m = o(n)$, we show that RANDOM BALL performs as well as another strategy, AGGRESSIVE EMPTY, which takes an optimal strategy on a subset of high-weight bins and turns it into an optimal strategy on all the bins.

Interestingly, the greedy strategy FULLEST BIN is not generally optimal, and in particular:

Observation. *There are distributions for which* FULLEST BIN *is pessimal, that is, it recycles at most 2 balls per round whereas OPT recycles almost m balls per round.*

For example, consider the ***skyscraper distribution***, where $\mathbf{p}_0 = 1 - 1/n + 1/n^2$ and $\mathbf{p}_i = 1/n^2$, for $0 < i \leq n - 1$. Suppose that $m = \sqrt{n}$. Then FULLEST BIN will pick bin 0 every time until it has at most one ball, at which point it will pick another bin, which will almost certainly have 1 ball in it. Thus, the recycling rate of FULLEST BIN drops below 2. Suppose, instead, that we recycle the least-full non-empty bin. In this case, every approximately $\sqrt{n}$ rounds, a ball lands in a low-probability bin and is promptly returned to bin 0. Thus, the recycling rate of this strategy is nearly m. Thus, FULLEST BIN is pessimal for this distribution.

However, the uniform distribution is of particular importance to insertion buffers for databases based on B-trees. This is because (arbitrary) random B-tree insertions are nearly uniformly distributed across the leaves of the B-tree, as we show in section 5.2. On the uniform distribution, FULLEST BIN and GOLDEN GATE are optimal even up to lower order terms:

Theorem 7 (section 5.5)**.** FULLEST BIN *and* GOLDEN GATE *are optimal to within an additive constant for the ball-recycling game with distribution* $\mathbf{u}$ *for any n and m. They each achieve a recycling rate of at least $2m/(n+1)$, whereas no recycling strategy can achieve a recycling rate greater than $2m/n + 1$.*

In this case, RANDOM BALL is only optimal to within a multiplicative constant in the following range:

Theorem 8 (section 5.5)**.** *On the uniform distribution* $\mathbf{u}$*, for sufficiently large m,* RANDOM BALL *is at least $(1/2 + 1/(2^3 3^4))$-optimal and at most $(1 - 3/1000)$-optimal.*

Thus, we establish some surprising results: that FULLEST BIN can perform poorly for arbitrary $\mathbf{p}$ but is optimal for $\mathbf{u}$, up to lower-order terms; and that RANDOM BALL is asymptotically optimal for any $\mathbf{p}$ and in particular is more than 1/2-optimal but not quite optimal for $\mathbf{u}$.

In section 5.6, we present experimental results showing that our analytical results for the ball recycling problem closely match performance results in real databases.

We describe the recycling strategies of several commercial and open-source database systems. In particular we focus on InnoDB, a B-tree that uses a variant of Random Ball.

Our results suggest that Fullest Bin or Golden Gate would be a better choice than Random Ball for InnoDB. In particular, Golden Gate requires almost no additional bookkeeping, and can be implemented in InnoDB with a change of only a few lines of code. With this implementation, we measured a 30% improvement in its insertion-buffer flushing rate, which is in line with our theoretical results.

We conclude that ball recycling is a natural hitherto unexplored balls-and-bins game that closely models a widely deployed method for improving the performance of databases. Moreover, this is the first application (to our knowledge) of a balls-and-bins game to the throughput of a system. This is in contrast to past balls-and-bins analyses, which modeled load balancing and latency.

5.2 Ball Recycling and Insertion/Update Buffers

Ball recycling models insertion and update buffers, which are widely used in modern databases [10, 58, 110, 59, 31, 81, 83, 95, 104, 32, 18]. These implementations are discussed in more detail in section 5.6.

For a key-value store, such as a database, an ***insertion buffer*** is a cache of recently inserted items. When the insertion buffer fills, the database selects a disk block and all the cached items going to that block are evicted in bulk. If k elements are flushed in bulk, then there is a speedup of k, compared to writing the elements to the destination block as soon as they arrive. After evicting k items, there is room for k new elements in the buffer. An ***update buffer*** caches changes to existing key-value pairs but is otherwise like an insertion buffer. Although these types of buffers seem quite similar, we show that they have important differences in how they are modeled as a ball-recycling game.

The mapping to ball recycling is direct: disk blocks are bins and elements in the

insertion/update buffer are balls. The probability distribution $\mathbf{p}$ is based on the distribution of items inserted or updated. Evicting all the items going to a disk block corresponds to emptying the bin associated with that disk block. After an eviction of k items, we have room for k new insertions/updates, i.e., we have k new balls to throw. The policy for selecting the target disk block of an eviction corresponds to the policy for selecting a bin to recycle, and the speedup induced by an eviction policy is its recycling rate.

For B-trees, insertion buffers and update buffers differ in an important way: updates do not change the structure of leaves of a B-tree. In contrast, insertions can change the range of keys associated with a leaf (due to leaf splits), which yields the following result:

Lemma 13. *If N keys are inserted into a B-tree i.i.d. according to some key distribution $\mathbf{q}$, then provided $B = \Omega(\log N)$, the maximum probability that a leaf has of receiving the next insertion is $O(B/N)$ with probability 1. Thus the corresponding recycling game is asymptotically* **almost uniform***: no bin has probability more than a constant multiple of $\frac{1}{n}$.*

We prove the uniformity bound as follows. Let $F(\kappa)$ be the cumulative density function (CDF) of $\mathbf{q}$, which is the probability that an item sampled from $\mathbf{q}$ is less than κ. If κ is distributed according to $\mathbf{q}$, then $F(\kappa)$ will be uniformly distributed on $[0, 1]$.

If n points are sampled from $[0, 1]$ uniformly and sorted so that $x_1 \leq x_2 \leq \cdots \leq x_n$, then $\max x_{i+B} - x_i$ is known as the ***maximal B-spacing***. It follows that:

Lemma 14. *Having inserted n keys into a B-tree i.i.d. according to a distribution $\mathbf{q}$, the maximum probability that any leaf has of receiving the next insertion is less than the maximum B-spacing of the CDFs of those points.*

It is known that:

Lemma 15 ([45])**.** *If $B = \Theta(\log n)$, then the maximum B-spacing of n points distributed uniformly on the unit interval is $\Theta(B/n)$ with probability 1.*

For $B = \omega(\log n)$, we can subdivide B into intervals of $\log n$ points, each of which will satisfy the lemma. Adding together the resulting bounds, we have that the maximal

B-spacing of n points is $O(B/n)$ with probability 1. Together with lemma 14, this implies lemma 13.

We also note that an (almost-uniform) ball-recycling game is an imperfect model for an insertion buffer, because the ball-recycling game has a fixed number of bins, whereas in the insertion buffer, the number of disk blocks will increase. Furthermore, insertions may not be independently distributed.

Finally, the implementation of these strategies is a point of departure between insertion/update buffers and ball recycling. In ball recycling, it is obvious which bin each ball is in. In insertion/update buffers for a B-tree, elements have a key, but we don't necessarily know what the buckets are, since the mapping from keys to buckets depends on the pivots used to define the B-tree leaves. Fullest Bin needs to know what the buckets are, whereas Random Ball and Golden Gate do not. For Random Ball this is because the key of the randomly selected item can be used to fetch its target B-tree leaf, after which we know the max and min keys in that leaf, and Golden Gate can be implemented by remembering the upper bound of the last leaf to which we flushed and then flushing the item with the successor of that key, along with all the other keys going to that leaf. None of these strategies require knowledge of $\mathbf{p}$.

Our results on general $\mathbf{p}$ have an interesting implication for B^ϵ-trees, which are known to be asymptotically optimal for insertions, in the worst case. B^ϵ-trees can also handle updates by propagating messages to the leaves. For some update distributions, flushing according to Random Ball can achieve an update rate that is B^ε faster than Fullest Bin. We expect to try Random Ball flushing in our B^ϵ-tree-based file system, BεtrFS [62, 112, 38, 61, 48].

5.3 Ball Recycling and Markov Theory

We begin our analysis of ball-recycing games with some preliminary results. In particular, we show that all finite-state ball-recycling strategies have stationary distributions.

5.3.1 Ball-recycling games are Markov decision processes.

This section makes use of the standard theory of Markov chains and Markov decision processes; for an introduction see *e.g.* Kallenberg [66].

In a ball-recycling game, we represent the configuration of the balls as a vector $X = (X_i)$ of length n, where X_i is the number of balls in the ith bin. Since the number of balls is finite, there are only a finite number of bin configurations.

A recycling strategy A takes as input the current bin configuration X together with an internal state S, and selects a non-empty bin to recycle; the next state is obtained by removing all the balls from the selected bin and re-throwing them according to $\mathbf{p}$. The bin selection may be randomized. We write A^iX for the state obtained after i rounds of recycling using strategy A. In each round, the recycling algorithm earns a reward equal to the number of balls recycled in that round.

In this way, the ball-recycling game is a Markov decision process, and we are interested in policies that maximize the expected average recycling rate, defined for a policy A as

$$\mathcal{R}^A = \lim_{T\to\infty} \frac{1}{T} \sum_{t=0}^{T} R(A^t X_0).$$

Note that Markov decision processes are very general. For example, in a Markov decision process the policy may vary over time, and may even take the entire history of the process and its own past decisions into account when deciding on its next action. Thus, for some strategies A, the limit $\mathcal{R}^A$ may not exist. In the literature of Markov decision processes, this is often handled by taking the lim sup instead of the limit. However, for any Markov decision process, any strategy that maximizes the limit also maximizes the lim sup [66]. Therefore, for simplicity, we will focus only on strategies for which the limit is well-defined, and the results will generally hold for the lim sup of arbitrary strategies.

A Markov decision process policy is ***deterministic*** if it decides on its next action based solely on the current state, *i.e.* without looking at history, the number of time steps that have passed or by flipping random coins. A deterministic policy can be represented as a simple table mapping each state to a single action to be taken whenever

the system is in that state.

The specific strategies we analyze are ***finite-state*** strategies, where the internal state has only finitely many configurations. When we prove our lower bounds, we will further restrict ourselves to ***stateless*** strategies, where there is a unique internal state. In order to do so, we make use of the following lemma.

Lemma 16 ([66])**.** *There exists a stateless deterministic recycling strategy OPT that achieves the optimal expected average recycling rate.*

Proof. This follows from Kallenberg's Corollary 5.4. □

5.3.2 Stationary distributions of recycling strategies

A ball-recycling game and a recycling strategy together define a Markov process on the state space; the space of pairs comprising a balls-and-bins configuration and an internal state. If the strategy is stateless, this is a Markov process on the balls and bins space.

There are a finite number of balls-and-bins configurations. Therefore, a ball-recycling game with a finite-state recycling strategy defines a finite Markov process, and so has at least one stationary distribution.

We now show that stateless recycling strategies result in Markov processes with unique stationary distributions. The following lemma shows that, when we look only at the bin configurations, recycling games have properties analogous to irreducibility and aperiodicity in Markov chains.

Lemma 17. *For any ball-recycling game with m balls and n bins there is an $\epsilon > 0$ such that, for all bin configurations X and Y, and for all recycling strategies, the probability that X reaches Y within $\min(m, n)$ steps is at least ϵ. Furthermore, every bin configuration can transition to itself in one time step.*

Proof. We just need to show a sequence of outcomes for ball tosses that transform X into Y, no matter which bins the recycling strategy chooses to empty. So, imagine that, at each step, all the recycling balls land in occupied bins, so that at each step, the number of occupied bins goes down by 1. After at most $\min(m, n) - 1$ steps, all

the balls will be in a single bin. On the next round, the recycling strategy must choose that bin, causing all the balls to be rethrown. There is some non-zero probability that they land in configuration Y.

For the second observation, simply note that all the recycled balls may happen to land in the bin from whence they came. □

Lemma 18.

1. *All ball-recycling games using stateless recycling strategies have unique stationary distributions which are equal to their limiting distributions.*

Proof. Having fixed a stateless recycling strategy, the ball-recycling game is a Markov process on the balls-and-bins configuration. By lemma 17, this process is irreducible and aperiodic, and so has a unique stationary distribution equal to its limiting distribution. □

Together with lemma 16, we have:

Corollary 6. *For any ball-recycling game, there exists an optimal strategy with a unique stationary distribution.*

We now show that two of the three main recycling strategies studied in this paper yield unique stationary distributions. The strategies are:

- Fullest Bin: selects the bin that has the most balls;
- Random Ball: selects a ball uniformly at random and recycles whichever bin it is in;
- Golden Gate: selects the bins in a round robin sequence.

Fullest Bin is a deterministic strategy, Random Ball is a stateless strategy and Golden Gate is a finite-state strategy. By lemma 18 we have:

Lemma 19. Fullest Bin *and* Random Ball *have unique stationary distributions.*

5.4 Random Ball is Optimal

In this section we prove theorem 6, showing that RANDOM BALL is $\Theta(1)$-optimal.

5.4.1 Outline of Proof

We prove theorem 6 in the following steps:

1. No recycling strategy has a recycling rate exceeding $(2m + n - 1)/\|\mathbf{p}\|_{\frac{1}{2}}$. (section 5.4.2)

2. RANDOM BALL matches that bound when $m \geq n$, leading to case (1) of theorem 6. (section 5.4.3)

3. There is an $\Theta(1)$-optimal strategy, AGGRESSIVE EMPTY, when $m < n$. The recycling rate of AGGRESSIVE EMPTY matches case (2) of theorem 6. (section 5.4.4)

4. By comparison to AGGRESSIVE EMPTY, RANDOM BALL is $\Theta(1)$-optimal when $m < n$. (section 5.4.5)

5.4.2 The Upper Bound

We begin by proving an important lemma that will be used throughout, which we refer to as the ***flow equation***. Then we proceed to prove the upper bound.

Let A be a stateless strategy with stationary distribution $\chi^{A,\mathbf{p}}$. Let ϕ_i be the event that A picks bin i to recycle. Let $\mathcal{R}_i^A = E[R^A(\chi^{A,\mathbf{p}})|\phi_i]$ be the number of balls recycled given that the strategy picks bin i, and $f_i = P(\phi_j)$, the probability of picking the bin i in the stationary distribution. We note that $\mathcal{R}_i^A$ and f_i could alternatively be defined as limits of repeated applications of A to any given starting state.

For a given bin i, we can analyze the "flow" of balls into and out of i. When k balls are thrown, $\mathbf{p}_i k$ of them are expected to land in i. For a ball to leave i, i must first be picked to be emptied by A, at which point every ball in i will be evicted. In the stationary distribution, the net flow must be zero. We can generalize to any set of bins and get:

Lemma 20. *Let A be a statelss strategy for a ball-recycling game with n bins with probabilities $\mathbf{p}$. If L is a subset of the bins, $\mathbf{p}_L = \sum_{\ell \in L} \mathbf{p}_\ell$, $f_L = \sum_{\ell \in L} f_\ell$ and $\mathcal{R}_L^A$ the conditional recycle rate given A picks a bin in L, then*

$$\mathbf{p}_L \mathcal{R}^A = f_L \mathcal{R}_L^A. \tag{5.1}$$

We will mostly use the following special case of the lemma 20:

Lemma 21 (The Flow Equation)**.** *Let A be a statless recycling strategy for a ball-recycling game with n bins with probabilities $\mathbf{p}$. Then, for all $0 \leq i < n$,*

$$\mathbf{p}_i \mathcal{R}^A = f_i \mathcal{R}_i^A. \tag{5.2}$$

We now describe the main upper bound on the recycling rate of any recycling strategy. In order to understand the intuition behind lemma 22, consider a given bin i. Intuitively, it makes sense to think that for a reasonable recycling strategy the recycling rate of the other bins in the system will go down as the number of balls X_i in bin i grows. After all, the X_i balls in bin i aren't available for recycling until bin i is selected. If we assume this intuition as fact for the moment, this suggests that the expected number of balls in bin i should be greater than half the recycling rate of bin i, perhaps excluding the last ball to land in the bin.

By the Flow Equation, this would suggest that

$$\mathrm{E}\left[X_i\right] \geq \frac{1}{2}(\mathcal{R}_i^A - 1) = \frac{1}{2}\left(\frac{\mathbf{p}_i}{f_i}\mathcal{R}^A - 1\right),$$

so that after summing over i, we obtain lemma 22.

However, the following strategy does not satisfy this assumption: for a given bin i, have the strategy just pick the least full non-empty bins until i has a few balls, then pick the fullest ones, then pick i and repeat. Showing that better strategies do not do this is non-trivial, and we prove lemma 22 by different means.

Lemma 22. *Consider a ball-recycling game with m balls, n bins and distribution $\mathbf{p}$. If A is a stateless strategy that picks bin i with frequency f_i, then its recycle rate is bounded by*

$$\mathcal{R}^A \leq \frac{2m + n - 1}{\sum_j \frac{\mathbf{p}_j}{f_j}}. \tag{5.3}$$

Given a strategy A, the idea of the proof is to use the invariance of the statistic

$$Z(X) = \sum_{j=1}^{n} \frac{X_j^2}{\mathbf{p}_j}, \tag{5.4}$$

under the action of A on its stationary distribution. The application of A to Z together with the flow equation creates a factor of $\sum_j \mathcal{R}_j$, which when solved for proves the bound. First, we begin with some foundational lemmas, and then proceed to prove the main results.

Lemma 23. *Suppose k balls are thrown into n bins i.i.d. according to distribution $\mathbf{p}$. Let $B(j,k)$ be the binomial random variable denoting how many balls land in the jth bin. The following hold:*

$$\mathrm{E}\left[B(j,k)\right] = \mathbf{p}_j k \tag{5.5}$$

$$\mathrm{E}\left[(B(j,k))^2\right] = \mathbf{p}_j(1-\mathbf{p}_j)k + \mathbf{p}_j^2 k^2 \tag{5.6}$$

Proof. $B(j,k)$ is a binomial random variable with parameters $\mathbf{p}_j$ and k. □

Next, given a state X, we compute the effect of recycling the ℓth bin on the jth component of Z. Note that if A recycles bin ℓ of state X, then $X_\ell = R^A(X)$.

Lemma 24. *In a ball-recycling game with m balls, n bins and probability distribution $\mathbf{p}$, if a strategy A recycles bin ℓ in state X, then for $j \neq \ell$,*

$$\mathrm{E}\left[(AX)_j^2\right] = X_j^2 + 2X_j\mathbf{p}_j R^A(X) + \mathbf{p}_j(1-\mathbf{p}_j)R^A(X) + \mathbf{p}_j^2 R^A(X)^2,$$

where $R^A(X) = X_\ell$ is the number of balls recycled.

Proof.

$$\begin{aligned}
\mathrm{E}\left[(AX)_j^2\right] &= \mathrm{E}\left[(X_j + B(j, R^A(X)))^2\right] \\
&= X_j^2 + 2X_j\mathrm{E}\left[B(j,R^A(X))\right] + \mathrm{E}\left[B(j,R^A(X))^2\right] \\
&= X_j^2 + 2X_j\mathbf{p}_j R^A(X) + \mathbf{p}_j(1-\mathbf{p}_j)R^A(X) + \mathbf{p}_j^2\left(R^A(X)\right)^2
\end{aligned}$$

□

We now can use this result to compute the result of applying A to Z.

Lemma 25. *In a ball-recycling game with m balls, n bins and probability distribution $\mathbf{p}$, if a strategy A recycles bin ℓ in state X, then*

$$\mathrm{E}\left[Z(AX)\right] = Z(X) - \left(1 + \frac{1}{\mathbf{p}_\ell}\right)\left(R^A(X)\right)^2 + (2m + n - 1)R^A(X)$$

Proof.

$$\begin{aligned}
\mathrm{E}\left[Z(AX)\right] &= \sum_{j=1}^{n} \mathrm{E}\left[(AX)_j^2/\mathbf{p}_j\right] \\
&= \frac{\mathrm{E}\left[(AX)_\ell^2\right]}{\mathbf{p}_\ell} + \sum_{j\neq\ell} \frac{\mathrm{E}\left[(AX)_j^2\right]}{\mathbf{p}_j} \\
&= (1-\mathbf{p}_\ell)R^A(X) + \mathbf{p}_\ell\left(R^A(X)\right)^2 \\
&\qquad + \sum_{j\neq\ell}\left(X_j^2/\mathbf{p}_j + 2X_jR^A(X) + (1-\mathbf{p}_j)R^A(X) + \mathbf{p}_j\left(R^A(X)\right)^2\right) \\
&= Z(X) - \left(2 + \frac{1}{\mathbf{p}_\ell}\right)\left(R^A(X)\right)^2 + 2mR^A(X) \\
&\qquad + \sum_{j}\left((1-\mathbf{p}_j)R^A(X) + \mathbf{p}_j\left(R^A(X)\right)^2\right) \\
&= Z(X) - \left(1 + \frac{1}{\mathbf{p}_\ell}\right)\left(R^A(X)\right)^2 + (2m + n - 1)R^A(X)
\end{aligned}$$

□

Now, we can prove lemma 22.

Proof of lemma 22. Let χ^A be the stationary distribution relative to A. Let ϕ_j be the event that A recycles the jth bin of χ^A, R_j^A the random variable of how many balls are recycled given the jth bin is chosen by A, $\mathcal{R}_j^A = \mathrm{E}\left[R_j^A\right]$ and f_j the probability that A recycles that bin. Because $\chi^A = A\chi^A$ by definition, we must have $\mathrm{E}\left[Z(A\chi^A)\right] = \mathrm{E}\left[Z(\chi^A)\right]$. Therefore:

$$\begin{aligned}
\mathrm{E}\left[Z(\chi^A)\right] &= \mathrm{E}\left[Z(A\chi^A)\right] \\
&= \sum_j f_j \mathrm{E}\left[Z(A\chi^A)|\phi_j\right] \\
&= \sum_j f_j \left(\mathrm{E}\left[Z(\chi^A)|\phi_j\right] - \left(1+\frac{1}{\mathbf{p}_j}\right)\mathrm{E}\left[\left(R_j^A\right)^2\right] + (2m+n-1)\mathcal{R}_j^A\right) \\
&\leq \mathrm{E}\left[Z(\chi^A)\right] + (2m+n-1)\mathcal{R}^A - \sum_j f_j \left(1+\frac{1}{\mathbf{p}_j}\right)\left(\mathcal{R}_j^A\right)^2 \\
&= \mathrm{E}\left[Z(\chi^A)\right] + (2m+n-1)\mathcal{R}^A - \sum_j \frac{1}{f_j}\left(\mathbf{p}_j^2+\mathbf{p}_j\right)\left(\mathcal{R}^A\right)^2,
\end{aligned}$$

where the inequality is due to the Cauchy-Schwartz Inequality and the last line is because of the Flow Equation.

Thus we have:

$$\mathcal{R}^A \leq \frac{2m+n-1}{\sum_j \frac{1}{f_j}\left(\mathbf{p}_j^2+\mathbf{p}_j\right)} \leq \frac{2m+n-1}{\sum_j \frac{\mathbf{p}_j}{f_j}}.$$

□

Lemma 22 applies to the optimal deterministic strategy OPT promised by corollary 6, and we know that $\mathcal{R}^A \leq \mathcal{R}^{\mathrm{OPT}}$ for *any* recycling strategy A. Thus, by maximizing the RHS of lemma 22, we can get an upper bound on the recycling rate of any recycling strategy.

Lemma 26. *Consider a ball-recycling game with m balls, n bins and distribution $\mathbf{p}$. For any recycling strategy A,*

$$\mathcal{R}^A \leq \frac{2m+n-1}{\|\mathbf{p}\|_{\frac{1}{2}}}.$$

Proof. This follows immediately from the Cauchy-Schwartz Inequality. □

5.4.3 Random Ball with $m \geq n$

We show the following lower bound, which with lemma 26, shows optimality when $m = \Omega(n)$.

Lemma 27. Random Ball *recycles at least $\frac{m}{\|\mathbf{p}\|_{\frac{1}{2}}}$ balls per round in expectation.*

Proof. Let $\chi^{\mathrm{RB}} = (\chi_i^{\mathrm{RB}})$ be the random variable of the number of balls in each bin in the stationary distribution of Random Ball. Random Ball recycles bin i with probability $\frac{\chi_i^{\mathrm{RB}}}{m}$, and therefore the expected number of balls recycled from bin i per round is $\frac{\mathrm{E}\left[\left(\chi_i^{\mathrm{RB}}\right)^2\right]}{m}$. The number of balls that land in bin i per round is $\mathbf{p}_i \sum_{j=1}^{n} \frac{\mathrm{E}\left[\left(\chi_j^{\mathrm{RB}}\right)^2\right]}{m}$. Since X is distributed stationarily, we must have

$$\mathbf{p}_i \sum_{j=1}^{n} \frac{\mathrm{E}\left[\left(\chi_j^{\mathrm{RB}}\right)^2\right]}{m} = \frac{\mathrm{E}\left[\left(\chi_i^{\mathrm{RB}}\right)^2\right]}{m} \geq \frac{\mathrm{E}\left[\chi_i^{\mathrm{RB}}\right]^2}{m},$$

using Jensen's Inequality. Clearing denominators, taking square roots and summing across i, we have

$$\left(\sum_{j=1}^{n} \mathrm{E}\left[\left(\chi_j^{\mathrm{RB}}\right)^2\right]\right)^{\frac{1}{2}} \sum_{i=1}^{n} \mathbf{p}_i^{\frac{1}{2}} = \sum_{i=1}^{n} \left(\mathbf{p}_i \sum_{j=1}^{n} \mathrm{E}\left[\left(\chi_j^{\mathrm{RB}}\right)^2\right]\right)^{\frac{1}{2}} \geq \sum_{i=1}^{n} \mathrm{E}\left[\chi_i^{\mathrm{RB}}\right] = m.$$

Therefore the expected recycle rate is

$$\sum_{j=1}^{n} \frac{\mathrm{E}\left[\left(\chi_j^{\mathrm{RB}}\right)^2\right]}{m} \geq \frac{m}{\left(\sum_{i=1}^{n} \sqrt{\mathbf{p}_i}\right)^2} = \frac{m}{\|\mathbf{p}\|_{\frac{1}{2}}}.$$

□

Corollary 7. *Consider a ball-recycling game with m balls and n bins. If $m = \Omega(n)$, then* Random Ball *is asymptotically optimal among recycling strategies.*

5.4.4 Aggressive Empty is Optimal

In this section, we investigate Aggressive Empty strategies, which aggressively recycle balls outside a given subset of bins. An Aggressive Empty strategy runs one strategy on a fixed subset of bins, but always chooses to recycle a bin outside of this set if there exists one which has any balls. Specifically, we show that a $\Theta(1)$-optimal strategy on a particular $O(m)$ subset of the bins can be extended to an $\Theta(1)$-optimal strategy on the full ball-recycling game by aggressively emptying the rest.

Consider a ball-recycling game with m balls, n bins, and ball distribution $\mathbf{p}$. Let L be some subset of bins and S be a strategy on the ***induced ball-recycling game*** of L,

which is the ball-recycling game with m balls, $|L|$ bins, and ball distribution $\mathbf{q}$, where

$$\mathbf{q}_i = \frac{\mathbf{p}_i}{\sum_{\ell \in L} \mathbf{p}_\ell}.$$

Therefore, $\mathbf{q}$ is $\mathbf{p}$'s conditional probability distribution on L. We define L, S-Aggressive Empty to be the strategy which empties the lowest weight non-empty bin in the complement of L if one exists and otherwise performs S on L. Note that all the balls will be in L whenever S is performed, so this is well-defined.

We begin by showing that there exists an L and S such that $|L| = O(m)$, L contains all bins with weight at least $\frac{1}{m}$, and L, S-Aggressive Empty is asymptotically optimal. Note that when $m = \Omega(n)$, this is trivial, because we can take L to be all the bins and S to be a $\Theta(1)$-optimal strategy; however, this section provides stronger bounds when $m = o(n)$. Intuitively, the idea is that very low weight bins won't be able to effectively accumulate balls, so strategies do better to recover any balls in them than to wait for more balls to land there.

Lemma 28. *There exists an L and S such that $|L| = O(m)$, L contains all bins of weight at least $\frac{1}{m}$ and L, S-Aggressive Empty is asymptotically optimal.*

Proof. By lemma 16, there exists an optimal deterministic strategy OPT. Using the flow equation, lemma 22 can be rewritten as:

$$\sum_{i=1}^{n} \mathcal{R}_i^{\mathrm{OPT}} \leq 2m + n.$$

Because OPT will never recycle an empty bin, each $\mathcal{R}_i^{\mathrm{OPT}} \geq 1$. Therefore, there can be at most m bins with average recycle rates at least 3. Let L be this set of bins, together with any bins of weight at least $\frac{1}{m}$, and we will construct a strategy S that aggressively empties the remaining bins into L.

S aggressively empties the complement of L, but also keeps a virtual configuration of where OPT thinks the balls are, as well as a log of where S has moved them. So when S aggressively empties a bin, it also updates the log of each ball it throws, indicating where it landed. When L^c is empty, it asks OPT which bin to recycle based on the virtual configuration. If it says to recycle a bin in L^c, we use the logs to update where

those balls will land in the virtual configuration. If it says to recycle a bin in L, we recycle those balls that are there in the virtual configuration, leaving any others behind in that same bin. Thus S performs OPT but rushes ahead to recycle those balls outside of L.

Now, consider t rounds of OPT. For large enough t, OPT will recycle on average at most 3 balls at a time from L^c. S recycles at least 1 ball at a time from L^c and exactly as many balls at a time from L. Therefore for large t, t rounds of OPT will correspond to at most $3t$ rounds of S, and during this period S will recycle the same number of balls. Thus S is 1/3-optimal. □

Next we compute the recycle rate of L, S-Aggressive Empty as a function of the recycle rate of S on the induced ball-recycling game on L.

Lemma 29. *If $\mathcal{R}^S$ is the recycle rate of S (on L), and q is the probability of a ball landing in L^c, then the recycle rate of L, S-Aggressive Empty is*

$$\mathcal{R}^{AE} = \Theta\left(\frac{1}{(1-q)/\mathcal{R}^S + q}\right)$$

Proof. Consider a collection of recycling rounds of L, S-Aggressive Empty where t of those times L, S-Aggressive Empty recycles a bin from L. Say b balls are thrown from bins in L and a balls land in L^c. Now, if m balls are thrown into bins of size at most $\frac{1}{m}$, then the expected number of empty bins is at most

$$m\left(1 - \frac{1}{m}\right)^m \leq \frac{m}{e}.$$

Because fewer thrown balls will have fewer collisions, this means the expected number of non-empty bins when $k \leq m$ balls are thrown into L^c is at least $\left(1 - \frac{1}{e}\right)k$, requiring at least as many time steps to aggressively empty. Thus, for large t, the expected number of turns required to empty the a balls out of L^c is at least $\left(1 - \frac{1}{e}\right)\frac{a}{1-q}$. Whereas even if the balls were recycled from L^c one at a time this expected number of turns is at most $\frac{a}{1-q}$ turns. The number of balls recycled during this period is $b + \frac{a}{1-q}$, and we have shown the number of rounds ρ satisfies:

$$\rho = \Theta\left(t + \frac{a}{1-q}\right).$$

For large enough t, $b = \Theta\left(t\mathcal{R}^S\right)$ and $a = \Theta\left(tq\mathcal{R}^S\right)$, so the overall recycle rate $\mathcal{R}^{\mathrm{AE}}$ therefore satisfies

$$\mathcal{R}^{\mathrm{AE}} = \Theta\left(\frac{t\mathcal{R}^S + tq\mathcal{R}^S/(1-q)}{t + tq\mathcal{R}^S/(1-q)}\right) = \Theta\left(\frac{1}{(1-q)/\mathcal{R}^S + q}\right).$$

□

5.4.5 Random Ball is Optimal

In this section we will further examine the performance of Random Ball and show that it is asymptotically optimal. We first describe a sufficient condition for optimality of a strategy based on its recycle rate on L, then show that Random Ball satisfies this criterion.

Lemma 30. *Let L be a set of $O(m)$ bins for which there exists a strategy T such that L, T-Aggressive Empty is asymptotically optimal. Let $\mathcal{R}^{OPT_L}$ be the recycle rate of the optimal strategy on the induced ball-recycling game of L. For a given strategy S, let $\mathcal{R}_L^S$ be the conditional recycle rate of S in the stationary distribution given that a ball in L is selected, and q be the probability that a ball lands in L^c, i.e. $q = \sum_{k\in L^c} \mathbf{p}_k$. If either*

$$\mathrm{E}\left[\mathcal{R}_L^S\right] = \Omega(\mathcal{R}^{OPT_L}) \quad \text{or} \quad \mathrm{E}\left[\mathcal{R}_L^S\right] = \Omega\left(\frac{1}{q}\right),$$

then S is asymptotically optimal.

Proof. By applying lemma 20, the subset variant of the flow equation, to L,

$$f_L\mathcal{R}_L^S = (1-q)(f_L\mathcal{R}_L^S + (1-f_L)\mathcal{R}_{L^c}^S),$$

where f_L is the stationary probability of S picking a bin in L. Solving for f_L,

$$f_L = \frac{\mathcal{R}_{L^c}^S}{q\mathcal{R}_L^S + \mathcal{R}_{L^c}^S}. \tag{5.7}$$

Suppose $\mathcal{R}_L^S = o\left(\frac{1}{q}\right)$ and $\mathcal{R}_L^S$ is $\Theta(1)$-optimal on L. If $\mathcal{R}_L^S \le \frac{1}{q}$, then $f_L \ge \frac{1}{2}$, and so because $\mathcal{R}^S = f_L\mathcal{R}_L^S + (1-f_L)\mathcal{R}_{L^c}^S$, we must have $\mathcal{R}^S = \Omega(\mathcal{R}_L^S)$.

Now, using lemma 28, let L and T be such that L, T-Aggressive Empty is asymptotically optimal, and let $\mathcal{R}^{\mathrm{AE}}$ be its expected recycle rate. By lemma 29,

$$\mathcal{R}^{\mathrm{AE}} = \Theta\left(\frac{1}{(1-q)/\mathcal{R}^T + q}\right) = O\left(\mathcal{R}^T\right) = O\left(\mathcal{R}_L^S\right) = O\left(\mathcal{R}^S\right),$$

so S must be asymptotically optimal.

If $\mathcal{R}_L^S = \Omega\left(\frac{1}{q}\right)$, then $\mathcal{R}_L^S > \frac{\alpha}{q}$ for some α. Rearranging Equation (5.7) and multiplying by $\mathcal{R}_L^S$ yields

$$f_L \mathcal{R}_L^S = \frac{1}{\frac{q}{\mathcal{R}_{L^c}^S} + \frac{1}{\mathcal{R}_L^S}}.$$

Here $\frac{1}{\mathcal{R}_L^S} \leq \frac{q}{\mathcal{R}_{L^c}^S}$, so $f_L \mathcal{R}_L^S = \Omega\left(\frac{1}{q}\right)$, and thus $\mathcal{R}^S = \Omega\left(\frac{1}{q}\right)$ as well. Now we can compare to L,T-Aggressive Empty as above:

$$\mathcal{R}^{\mathrm{AE}} = \Theta\left(\frac{1}{(1-q)/\mathcal{R}^T + q}\right) = O\left(\frac{1}{q}\right) = O\left(\mathcal{R}^S\right)$$

so in this case S is asymptotically optimal as well. □

We can now prove theorem 6.

Proof of theorem 6. If $m = \Omega(n)$, then by lemmas 26 and 27 we are done.

Otherwise, let L be a set of $O(m)$ bins for which there exists a strategy T such that L,T-Aggressive Empty is asymptotically optimal. We will prove the result for a slightly modified Random Ball that only recycles 1 ball outside of L even if more are available; that is, it moves only one of the balls in the bin. Since this strategy is worse than Random Ball, this will be sufficient. We number the bins so that the first $|L|$ bins comprise L.

If $\mathcal{R}_L \geq \frac{1-q}{q}$, then we are done by lemma 30. Otherwise, in the stationary distribution, when a bin in L is recycled, the expected number of balls which land in L^c is $q\mathcal{R}_L < 1-q$. When a bin in L^c is recycled, the expected number of balls which land in L is $1-q$. Thus Random Ball must pick a bin in L more than half the time, and so the expected number of balls in L must be more than $\frac{m}{2}$.

Now analogously to the proof of lemma 27, we have:

$$\mathbf{p}_i \left(\sum_{j=1}^{|L|} \mathrm{E}\left[\left(\chi_j^{\mathrm{RB}}\right)^2\right] + \sum_{j=|L|+1}^{n} \mathrm{E}\left[\chi_j^{\mathrm{RB}}\right] \right) = \mathrm{E}\left[\left(\chi_i^{\mathrm{RB}}\right)^2\right] \geq \mathrm{E}\left[\chi_i^{\mathrm{RB}}\right]^2.$$

Thus,

$$\mathrm{E}\left[\chi_i^{\mathrm{RB}}\right] \leq \sqrt{\mathbf{p}_i} \left(\sum_{j=1}^{|L|} \mathrm{E}\left[\left(\chi_j^{\mathrm{RB}}\right)^2\right] + \frac{m}{2} \right)^{\frac{1}{2}}.$$

Summing over $i \leq |L|$ yields

$$\mathrm{E}\left[\chi_L^{\mathrm{RB}}\right] \leq \left(\sum_{i=1}^{|L|} \sqrt{\mathbf{p}_i}\right)\left(\sum_{j=1}^{|L|} \mathrm{E}\left[\left(\chi_j^{\mathrm{RB}}\right)^2\right] + \frac{m}{2}\right)^{\frac{1}{2}},$$

where χ_L^{RB} is the expected number of balls in L. Now,

$$\mathcal{R}_L^{\mathrm{RB}} \geq \frac{1}{m}\sum_{j=1}^{|L|} \mathrm{E}\left[\left(\chi_j^{\mathrm{RB}}\right)^2\right] \geq \frac{\mathrm{E}\left[\chi_L^{\mathrm{RB}}\right]^2}{m\left(\sum_{i=1}^{|L|}\sqrt{\mathbf{p}_i}\right)^2} - \frac{1}{2} > \frac{m}{4\|\mathbf{p}_L\|_{\frac{1}{2}}} - \frac{1}{2},$$

where $\mathbf{p}_L$ is the conditional probability distribution on L obtained from $\mathbf{p}$. The last inequality holds because there are at least $\frac{m}{2}$ balls in L in expectation.

Thus by lemma 26, Random Ball is asymptotically optimal on the induced system of L, and therefore Random Ball is asymptotically optimal by lemma 30. □

5.5 The Uniform Case

The results of section 5.4 hold for any distribution of the balls into the bins. In this section we consider the special case where they are uniformly distributed, which models insertion buffers as discussed in section 5.2. We then show that Golden Gate and Fullest Bin are optimal, up to lower-order terms, in this setting, whereas Random Ball is at least 1/2- and at most $(1-\epsilon)$-optimal, for some constant $\epsilon > 0$.

For a ball-recycling game with uniformly distributed balls, lemma 26 implies:

Corollary 8. *Consider a ball-recycling game with m balls, n bins and uniform distribution $\mathbf{u}$. For any recycling strategy A,*

$$\mathcal{R}^A \leq \frac{2m+n-1}{n} < 2\frac{m}{n} + 1.$$

The average number of balls in a bin is m/n, so corollary 8 suggests that any "reasonable" strategy will be at least 1/2-optimal in the uniform case.

We now show that Golden Gate and Fullest Bin are within an additive constant of optimal on strictly uniform distributions.

Lemma 31. Golden Gate *and* Fullest Bin *each recycle at least $2m/(n+1)$ balls per round in expectation.*

Proof. Let S be the random variable denoting the number of balls thrown in a given round with Golden Gate. Golden Gate will recycle the bins in order starting from the next one and cycling around. Therefore, we can consider the collection of bins to be a queue. After throwing the balls, the average place in the queue in which a ball lands is the $[(n-1)/s]$th bin, due to uniformity. Each ball thrown will therefore sit for an average of at most $(n-1)/2$ rounds before it is thrown again. Therefore, $m - \mathrm{E}[S] \leq \mathrm{E}[S](n-1)/2$, and we have the result after solving for $\mathrm{E}[S]$.

Let T be the random variable denoting the number of balls thrown in a given round with Fullest Bin. If after removing the balls in the Fullest Bin, we list the bins in order of fullness, we can again think of the bins as a sort of queue. When we throw the balls, the average place in the queue which a ball lands is the $[(n-1)/2]$th bin as above, due to uniformity. Now, we reorder the bins back into fullness order. During the reordering more balls are moved up the queue than down, thus each ball thrown into the system will sit for an average of less than $(n-1)/2$ rounds before it is thrown again. Therefore, as above, $m - \mathrm{E}[S] \leq \mathrm{E}[S](n-1)/2$, and we are done. □

Corollary 8 and lemma 31 together prove theorem 7. Despite these strong performance bounds, recall that Fullest Bin can perform arbitrarily badly on non-uniform $\mathbf{p}$. Random Ball on the other hand is always $\Theta(1)$-optimal.

5.5.1 Random Ball in the Uniform Case

However, Random Ball does not achieve this level of optimality on uniform distributions. In this section we will show in theorem 8 that Random Ball recycles at most $1 + (2-\epsilon)m/n$ balls per round in expectation, for some $\epsilon > 0$. The upper bound is given in lemma 33 and corollary 9, and the lower bound is given in lemma 34.

We begin with the following lemma:

Lemma 32. *Let χ^{RB} be the stationary distribution relative to* Random Ball*, $R^{RB}(X)$ the random variable of how many balls* Random Ball *recycles from ball configuration*

X, and $\mathcal{R}^{RB} = \mathrm{E}\left[R^{RB}\left(\chi^{RB}\right)\right]$ the expected recycle rate of Random Ball. *Then,*

$$\frac{\mathrm{E}\left[R^{RB}\left(\chi^{RB}\right)^2\right]}{\mathcal{R}^{RB}} = \frac{2m+n-1}{n+1} \leq 1 + \frac{2m}{n}.$$

Proof. Consider the random variable of the number of distinct unordered pairs of balls which are in the same bin in χ^{RB}. In expectation, a round of Random Ball eliminates

$$\binom{\mathcal{R}^{\mathrm{RB}}}{2}$$

and creates

$$\sum_{k=0}^{\mathcal{R}-1} \frac{m - \mathcal{R}^{\mathrm{RB}} + k}{n}$$

such pairs. In the stationary distribution, these must be equal, so

$$\frac{\mathrm{E}\left[R^{\mathrm{RB}}\left(\chi^{\mathrm{RB}}\right)^2\right]}{2} - \frac{\mathcal{R}^{\mathrm{RB}}}{2} = \frac{(2m-1)\mathcal{R}^{\mathrm{RB}}}{2n} - \frac{\mathrm{E}\left[R^{\mathrm{RB}}\left(\chi^{\mathrm{RB}}\right)^2\right]}{2n}.$$

After rearranging we have the result. □

Lemma 33. *There exists a constant $\alpha > 0$ such that* Random Ball *is at most $(1-\alpha)$-optimal.*

Proof. Let χ^{RB} be the stationary distribution relative to Random Ball, $R^{\mathrm{RB}}(X)$ the random variable of how many balls Random Ball recycles from ball configuration X, and $\mathcal{R}^{\mathrm{RB}} = \mathrm{E}\left[R^{\mathrm{RB}}\left(\chi^{\mathrm{RB}}\right)\right]$ the expected recycle rate of Random Ball. We will prove the result by contradiction, so assume that for all constant $\epsilon > 0$,

$$\mathcal{R}^{\mathrm{RB}} \geq 1 + \frac{(2-\epsilon)m}{n}.$$

Let $c \in (1,2)$ be a constant to be determined later. We say a bin is ***light*** if it contains at most cm/n balls. Let L be the random variable of the number of balls in light bins in the stationary distribution. Then the probability q_L that Random Ball recycles a light bin in the stationary distribution is $\mathrm{E}[L]/m$. We proceed by cases.

Case 1. Suppose $\mathrm{E}[L] \geq \delta m$ for some constant $\delta > 0$. Then $q_L \geq \delta$ and for

$c \leq 2 - 2\epsilon$ and $\epsilon < 1/2$,

$$\begin{aligned}
\operatorname{Var}\left[R^{\mathrm{RB}}\left(\chi^{\mathrm{RB}}\right)\right] &= \mathrm{E}\left[\left(R^{\mathrm{RB}}\left(\chi^{\mathrm{RB}}\right) - \mathcal{R}^{\mathrm{RB}}\right)^2\right] \\
&\geq q_L\left(1 + \frac{(2-\epsilon)m}{n} - \frac{cm}{n}\right)^2 \\
&\geq \frac{\epsilon^2\delta}{4}\left(\frac{4m^2}{n^2} + \frac{4m}{n} + 1\right) \\
&\geq \frac{\epsilon^2\delta}{4}\left(\mathcal{R}^{\mathrm{RB}}\right)^2.
\end{aligned}$$

Thus by the definition of variance, we have

$$\mathrm{E}\left[R^{\mathrm{RB}}\left(\chi^{\mathrm{RB}}\right)^2\right] \geq \left(1 + \frac{\epsilon^2\delta}{4}\right)\left(\mathcal{R}^{\mathrm{RB}}\right)^2.$$

Now by lemma 32,

$$\mathcal{R}^{\mathrm{RB}} \leq \left(1 + \frac{\epsilon^2\delta}{4}\right)^{-1}\left(1 + \frac{2m}{n}\right).$$

Since ϵ, δ are constants greater than 0, we have our contradiction for the first case.

Case 2. Otherwise, $\mathrm{E}[L] < \delta m$. Since $L \in [0, m]$, $\mathrm{E}\left[L^2\right] < \delta m^2$. Lemma 32 implies $\mathrm{E}\left[R^{\mathrm{RB}}\left(\chi^{\mathrm{RB}}\right)^2\right] \leq (1 + 2m/n)^2$. Together Hölder's inequality we have

$$\begin{aligned}
\mathrm{E}\left[LR^{\mathrm{RB}}\left(\chi^{\mathrm{RB}}\right)\right] &\leq \left(\mathrm{E}\left[L^2\right]\mathrm{E}\left[R^{\mathrm{RB}}\left(\chi^{\mathrm{RB}}\right)^2\right]\right)^{1/2} \\
&< \left(\delta m^2\left(1 + \frac{2m}{n}\right)^2\right)^{1/2} \\
&= \sqrt{\delta} m\left(1 + \frac{2m}{n}\right) \qquad (5.8)
\end{aligned}$$

Let Y be the random variable of the number of balls in the stationary distribution which start in a light bin, but end up begin among the first $1 + cm/n$ balls in a heavy bin after an application of Random Ball. Let Φ be the random variable of the number of distinct unordered pairs of balls that are in the same light bin in the stationary distribution. Applying Random Ball in expectation creates at most

$$\mathrm{E}\left[\sum_{k=0}^{\mathcal{R}^{\mathrm{RB}}-1}\frac{L+k}{n}\right] = \mathrm{E}\left[\frac{2LR^{\mathrm{RB}}\left(\chi^{\mathrm{RB}}\right) + R^{\mathrm{RB}}\left(\chi^{\mathrm{RB}}\right)^2 - R^{\mathrm{RB}}\left(\chi^{\mathrm{RB}}\right)}{2n}\right]$$

such pairs, and eliminates at least

$$\mathrm{E}\left[\frac{Y}{1 + \frac{cm}{n}}\binom{1 + \frac{cm}{n}}{2}\right].$$

In the stationary distribution these quantities must be equal, so rearranging together with Equation (5.8), we have

$$\begin{aligned}\mathrm{E}[Y] &\leq \frac{2\mathrm{E}\left[LR^{\mathrm{RB}}\left(\chi^{\mathrm{RB}}\right)\right] + \mathrm{E}\left[R^{\mathrm{RB}}\left(\chi^{\mathrm{RB}}\right)^2\right] - \mathcal{R}^{\mathrm{RB}}}{cm} \\ &< \frac{2\sqrt{\delta}}{c}\left(1+\frac{2m}{n}\right) + \frac{\mathrm{E}\left[R^{\mathrm{RB}}\left(\chi^{\mathrm{RB}}\right)^2\right] - \mathcal{R}^{\mathrm{RB}}}{cm} \\ &< \left(1+\frac{2m}{n}\right)\left(\frac{2\sqrt{\delta}}{c}+\frac{2}{cn}\right),\end{aligned}$$

where we have used lemma 32 for the last inequality.

We now compute the effect on $\mathrm{E}[L]$ of applying Random Ball to the stationary distribution. By Markov's inequality, there must be more than $(1-1/c)n$ light bins, and so the probability that a ball is thrown into a light bin is more than $1-1/c$. Therefore, at least $(1-1/c)\mathcal{R}^{\mathrm{RB}}$ balls land in light bins in expectation. We expect at most $\mathrm{E}[Y]$ balls to be in light bins which turn into heavy bins. Finally, we recycle at most cm/n balls from a light bin $\mathrm{E}[L]/m$ of the time. Since the net change to L must be 0 in expectation,

$$\left(1-\frac{1}{c}\right)\mathcal{R}^{\mathrm{RB}} < \frac{c}{n}\mathrm{E}[L] + \mathrm{E}[Y].$$

However, this is a contradiction. Indeed, the LHS is at least

$$\left(1-\frac{1}{c}\right)\left(1+\frac{(2-\epsilon)m}{n}\right),$$

but the RHS is less than

$$\delta c\frac{m}{n} + \left(1+\frac{2m}{n}\right)\left(\frac{2\sqrt{\delta}}{c}+\frac{2}{cn}\right).$$

Thus, if we pick a sufficiently small $\delta > 0$, $\epsilon = 0.01$, $c = 1.98$ and $n \geq 3$, we have a contradiction. For $n \leq 2$, the contradiction follows immediately from lemma 32. □

Corollary 9. *Setting*

$$(\epsilon, c, \delta) = (0.001, 1.456, 0.042)$$

in the proof of theorem 8, we obtain

$$\mathcal{R}^{RB} < 1 + 1.994\frac{m}{n}.$$

Lemma 34. *For all $c > 0$, there exists a c' such that if $m \geq c'n \log n$, the uniform random ball policy has expected recycle rate at least*

$$\left(1 + \frac{1}{6^4} - c\right) \frac{m}{n}.$$

Proof. Let $X_{t,k}$ be the random variable denoting the number of balls in the kth bin at the beginning of the tth round. Because of symmetry, $X_{t,k}$ follows the same distribution as $X_{t,\ell}$ for any $k \neq \ell$. For simplicity, we let X_t be a random variable that follows the same distribution as $X_{t,k}$ for all k.

We pick t to be sufficiently large so that the system enters its stationary state after t rounds. Thus, X_t and $X_{t'}$ follows the same distribution for any $t' > t$.

Let Y_t be the random variable denoting the number of balls recycled in the tth round. By definition, we have

$$\mathrm{E}\left[Y_t\right] = \sum_{1 \leq k \leq n} \frac{\mathrm{E}\left[X_{t,k}^2\right]}{m} = \frac{n}{m}\left(\mathrm{E}\left[X_t\right]^2 + \mathrm{Var}\left[X_t\right]\right).$$

Note that $\mathrm{E}\left[X_t\right] = m/n$, and so $\mathrm{E}\left[Y_t\right] \geq m/n$.

To show $\mathrm{E}\left[Y_t\right]$ deviates from m/n, we derive a lower bound for $\mathrm{Var}\left[X_t\right]$.

If $\mathrm{P}\left(X_t \leq (1-\epsilon)m/n\right) \geq \delta$, then $\mathrm{E}\left[Y_t\right] \geq (1 + \epsilon^2\delta)m/n$.

Otherwise $\mathrm{P}\left(X_t > (1-\epsilon)m/n\right) > 1 - \delta$. We will show that if δ is small enough, then this case does not exist.

We say a bin is *heavy* if it has more than $(1-\epsilon)m/n$ balls. Let Z_t be the random variable denoting the number of heavy bins at the beginning of the tth round. We have

$$\mathrm{E}\left[Z_t\right] = \sum_{1 \leq k \leq n} \mathrm{E}\left[\mathrm{I}\left[X_{t,k} > (1-\epsilon)\frac{m}{n}\right]\right] > (1-\delta)n.$$

Z_t is a non-negative variable in $[0, n]$ and has expected value more than $(1-\delta)n$. By Markov's inequality,

$$\Pr[Z_t \leq (1-2\delta)n] < \frac{1}{2} \text{ and } \Pr[Z_t > (1-2\delta)n] > \frac{1}{2}.$$

We compute $\mathrm{E}\left[Z_{t+n/2}\right]$ from the Z_t. If $Z_t > (1-2\delta)n$ for some constant $\delta < 1/4$, the following hold during P, the time period between the tth round and the $(t+n/2)$th round:

1. At least $(1/2 - 2\delta)n$ bins in H_t are recycled,

2. At least $(1/2 - 2\delta)(1 - \epsilon)m$ balls are recycled,

3. At least $(1/2 - 2\delta)n$ bins in H_t are not recycled,

where H_t denotes the set of heavy bins at the beginning of the tth round.

Given (c), we can find a subset $S_t \subset H_t$ that is composed of $(1/2 - 2\delta)n$ bins in H_t not recycled during P. Note that which bins are recycled and which are not depends on the random choices made by the system. Hence, S_t varies.

Next, we derive a lower bound on the expected number of balls in any S_t. The balls which stay in S_t come from two different sources. There are those that stay in S_t at the beginning of the tth round, of which there are at least $|S_t|(1 - \epsilon)m/n$. There are also those which are recycled during P, of which there are at least $|S_t|(1 - \epsilon')|B|/n$ by lemma 35, to follow. Combining the two sources, the expected number of balls in S_t is at least

$$\Gamma = (1 - \epsilon)\left(\left(\frac{1}{2} - 2\delta\right) + \left(\frac{1}{2} - 2\delta\right)^2 (1 - \epsilon')\right) m.$$

Lemma 35. *Let B be the multiset of the first $(1/2 - 2\delta)(1 - \epsilon)m$ balls recycled during P. B is well-defined thanks to 2. above. Let L_i be the random variable denoting the number of balls in B that land on the ith bin. For all $\epsilon' > 0$, there exists a c' such that if $m \geq c' n \log n$*

$$\mathrm{E}\left[\min\{L_1, L_2, \ldots, L_n\}\right] \geq (1 - \epsilon')|B|/n.$$

Proof. For $i \in [1, n]$, $\mathrm{E}[L_i] = |B|/n$. By Chernoff bounds,

$$\Pr[|L_i - \mathrm{E}[L_i]| \geq (\epsilon'/2)\mathrm{E}[L_i]] \leq \frac{1}{n^2}$$

for some sufficiently large c'. Consequently, by the union bound,

$$\Pr[\min\{L_1, L_2, \ldots, L_n\} \leq (1 - \epsilon'/2)|B|/n] \leq \frac{1}{n}.$$

Because the L_i's are non-negative, we are done. □

Given Γ, we obtain the following bound:

$$\begin{aligned}\mathrm{E}\left[Z_{t+n/2} \,\middle|\, Z_t > (1-2\delta)n\right] &\leq |S_t| + \frac{m-\Gamma}{(1-\epsilon)m/n} \\ &= \left(\frac{1}{1-\epsilon} - \left(\frac{1}{2}-\delta\right)^2 (1-\epsilon')\right) n \\ &\approx \left(\frac{3}{4}+\epsilon+\delta\right) n\end{aligned}$$

Together with the trivial bound $\mathrm{E}\left[Z_{t+n/2} \,\middle|\, Z_t \leq (1-2\delta)n\right] \leq n$, $\mathrm{E}\left[Z_{t+n/2}\right]$ equals

$$\begin{aligned}&\mathrm{P}\left(Z_t \leq (1-2\delta)\right) \mathrm{E}\left[Z_{t+n/2} \,\middle|\, Z_t \leq (1-2\delta)\right] \\ &\quad + \mathrm{P}\left(Z_t > (1-2\delta)\right) \mathrm{E}\left[Z_{t+n/2} \,\middle|\, Z_t > (1-2\delta)\right] \\ &\qquad < \frac{1}{2}\left(1 + \left(\frac{3}{4}+\epsilon+\delta\right)\right) n \\ &\qquad \approx \left(\frac{7}{8} + \frac{\epsilon+\delta}{2}\right) n\end{aligned}$$

This leads to a contradiction if $\epsilon + \delta$ is small enough. This is because we have $\mathrm{E}\left[Z_t\right] > (1-\delta)n$ and $\mathrm{E}\left[Z_{t+n/2}\right] = \mathrm{E}\left[Z_t\right]$, because the system is stationary. As a result, we have a contradiction if $\frac{\epsilon+3\delta}{2} < \frac{1}{8}$.

Combining the results for the two cases, we wish to maximize $1 + \epsilon^2\delta$ subject to $\epsilon + 3\delta < \frac{1}{4}$. Picking $\epsilon = 1/6$ yields the result. □

Theorem 8 follows from corollaries 8 and 9 and lemma 34.

5.6 Database Experiments

In this section, we consider insertion buffers as they are used in practice. We demonstrate, through simulations as well as experiments on real-world systems, that the theoretical results in the prior sections hold and can be used to improve performance.

5.6.1 Insertion Buffers in Database Systems

Many databases cache recently inserted items in RAM so that they can write items to disk in batches. Examples include Azure [10], DB2 [58], Hbase [110], Informix [59], InnoDB [31], NuDB [81], Oracle [83], SAP [95], and Vertica [104]. They are also

used to accelerate inserts in several research prototypes, such as the buffered Bloom filter [32] and buffered quotient filter [18]. By batching updates to disk, these insertion buffers reduce the amortized number of I/Os per insert, which can substantially improve insertion throughput. Facebook claims that the insertion buffer in InnoDB speeds up some production workloads by a factor of 5 to 16, and accelerates some synthetic benchmarks by up to a factor of 80 [31].

A motivating factor for the use of insertion buffers is that they can significantly mitigate the precipitous performance drop that databases can experience when the data set grows too large to fit in RAM. section 5.6.1 shows the time per 1,000 insertions into a MySQL database using the InnoDB backend, with and without InnoDB's insertion buffer enabled. For the first $200,000$ insertions, the entire database fits in RAM, and so insertions are fast, even without the insertion buffer.

Once the database grows larger than RAM, insertion performance without the insertion buffer falls off a cliff. In fact, once the database reaches 1M rows, it can perform only about 200 insertions per second, suggesting that the throughput is limited by the random-I/O performance of the underlying disk. In the benchmark with the insertion buffer enabled, on the other hand, performance degrades by only a small amount.

Based on the performance of the first 1M insertions, it appears that InnoDB's insertion buffer effectively eliminates the performance cliff that can occur when the database grows larger than RAM. This improvement explains the popularity of insertion buffers in database design.

However, in our experiment, as the database continues to grow, the efficacy of the insertion buffer declines. Figure 5.1 shows the time per 10,000 insertions as the database grows to 50M rows. Although the performance without the insertion buffer drops more quickly early on, it remains relatively stable thereafter. Performance with the insertion buffer, on the other hand, slowly declines over the course of the benchmark until it is only about a third faster than without the insertion buffer. This is well below the $5 - 80\times$ speedups reported above.

As these experiments show, it can be difficult to extrapolate from small examples the performance gains that insertion buffers can provide for large databases. Therefore,

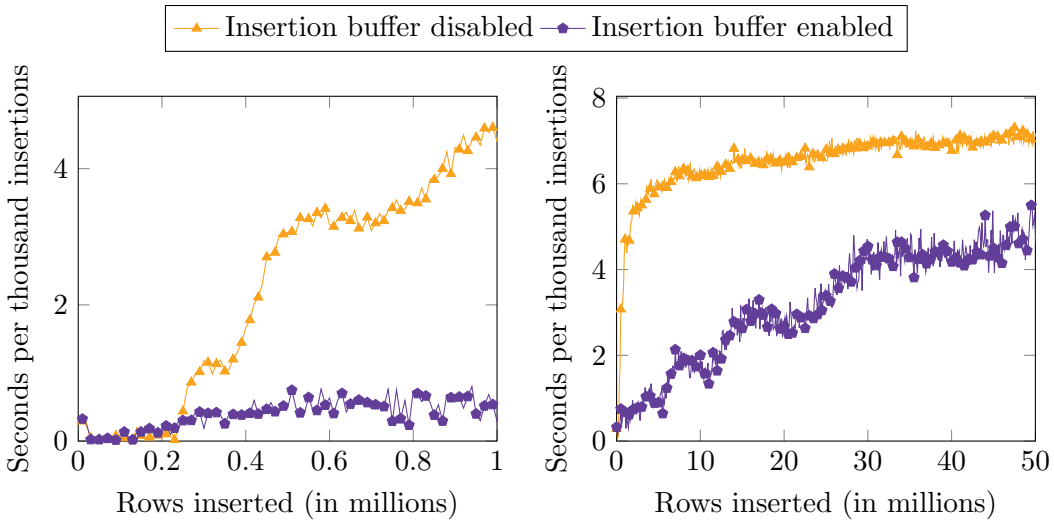


Figure 5.1: The cost of inserting batches of rows into an empty table in InnoDB with and without the insertion buffer. The rows are inserted in batches of 10,000 to avoid slowdown in parsing, and the keys are distributed uniformly. After 1M insertions, the buffered version takes 12.3% as long as the unbuffered version (measured over 50,000 insertions); after 50M insertions, the advantage is reduced so that the buffered version takes 68.3% of the time of the unbuffered. (Lower is better)

it is no wonder that reported speedups from insertion buffers vary wildly from as little as 2× to as high as 80× [31]. Some have even suggested that insertion buffers may provide many of the benefits of write-optimization [30], i.e., that insertion buffers can bring the performance of B-trees up to that of LSM-trees [82], COLAs [15], Fractal Trees [101], xDicts [28], or B^ϵ-trees [29].

5.6.2 Experimental Validation

Here we validate our theoretical study of insertion buffers by showing that our analysis above can have a material impact on the performance of databases with insertions buffers. We simulated workloads of random insertions to a B-tree, with varying distributions on the inserted keys. We found that, as predicted, the performance was independent of the input distribution and closely matched the performance predicted by our theorems.

We then ran workloads of random insertions into InnoDB and measured the average batch size of flushes from its insertion buffer. InnoDB implements a variant of the random-item flushing strategy. We modified it to implement the golden-gate flushing

strategy. Despite the additional complexities of InnoDB's insertion buffer implementation, we found that performance closely tracked our theoretical predictions and was independent of the distribution of inserted keys. We also found that the golden-gate flushing strategy improved InnoDB's flushing rate by about 30% over the course of our benchmark.

Our analysis explains why insertion buffers can provide dramatic speedups for small databases, but only small gains are available as the database grows. Our results also provide useful guidance to implementers about which flushing strategy will provide the most performance improvement.

Our results also show that insertion buffers cannot deliver the same asymptotic performance improvements that are possible with write-optimized data structures, such as LSM-trees and B^ϵ-trees.

5.6.3 Insertion-Buffer Background

This section describes insertion buffers that are actually implemented and used in deployed systems and recent research prototypes.

SAP:

The SAP IQ database supports an in-memory row-level versioning (RLV) store, and insertions are performed to the RLV store and later merged into the main on-disk store [95].

NuDB:

The NuDB SSD-based key-value store buffers all insertions in memory, and later flushes it to SSD [81]. Flushes occur at least once per second, or more often if insertion activity causes the in-memory buffer to fill.

Buffered Bloom and quotient filters:

Bloom filters are known to have poor locality for both inserts and lookups. The buffered Bloom filter [32] improves the performance of insertions to a Bloom filter on SSD by buffering the updates in RAM. The on-disk Bloom filter is divided into pages, and each page has a buffer of updates in RAM. When a page's buffer fills, the buffered changes are written to the page.

The buffered quotient filter stores newly inserted items in an in-memory quotient filter [18, 16]. When the in-memory quotient filter fills, its entire contents are flushed to the on-disk quotient filter.

InnoDB:

The InnoDB [84] B-tree implementation used in the MySQL [85] and MariaDB [51] relational database systems includes an insertion buffer.

Our experiments in this paper focus on InnoDB as an archetypal and open-source implementation of an insertion buffer, so we describe it in detail.

InnoDB structures its insertion buffer as a B-tree. When the insertion buffer becomes full, it selects the items to be flushed by performing a random walk from the root to a leaf. The random walk is performed by selecting, at each step, uniformly randomly from among the children of the current node. Once it gets to a leaf, it picks a single item to insert into the on-disk B-tree. This item, along with any other items in the insertion buffer that belong in that leaf, are inserted into the leaf and removed from the insertion buffer.

InnoDB's insertion buffer is complicated in several ways. First, the size of the insertion buffer changes over time, as InnoDB allocates more or less space to other buffers and caches.

InnoDB also has a leaf cache. Whenever a leaf is brought into cache for any reason, all inserts to that leaf that are currently in the insertion buffer are immediately applied to the leaf, and any future inserts to that leaf also skip the insertion buffer as long as the leaf remains in cache.

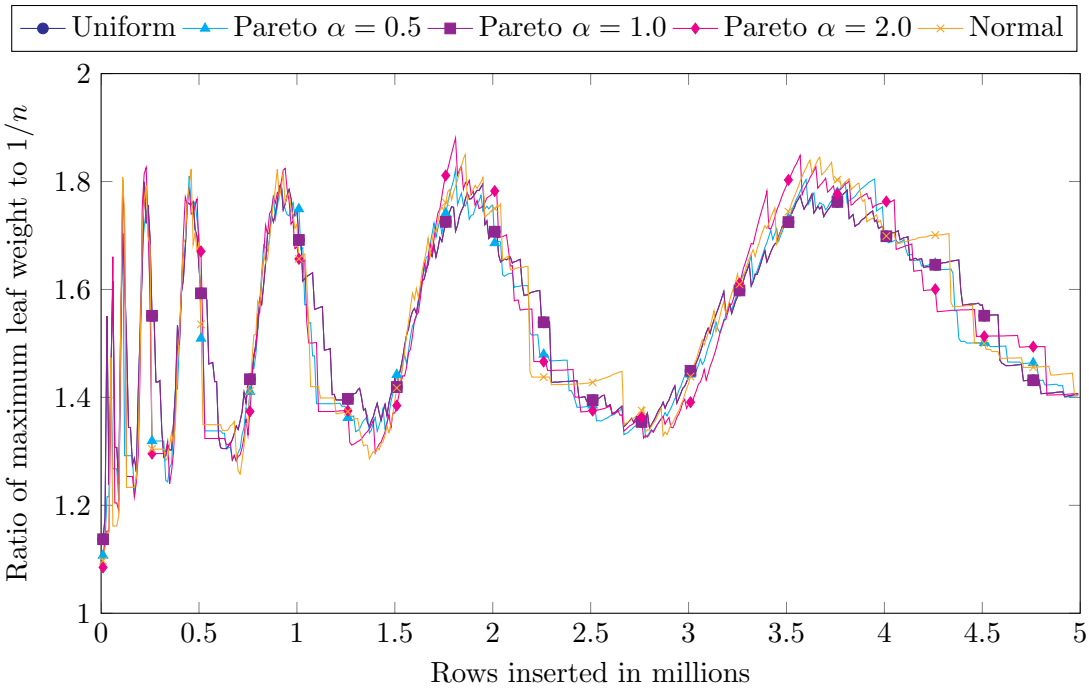


Figure 5.2: Deviation of the maximum weight leaf from uniform in simulation

Finally, it performs some flushing when the buffer is not full. Roughly every second, InnoDB performs a small amount of background flushing. Moreover, it prematurely flushes its buffer to a leaf when it calculates that such a flush will cause the leaf to split. We hypothesize that this feature exists to simplify the transactional system.

5.6.4 Leaf Probabilities in B-trees

In section 5.5, we established that, on insertion, the leaf probabilities are nearly uniform. We empirically verify this uniformity property by simulating insertions into the leaves of a B-tree. We insert real-valued keys i.i.d. according to uniform, Pareto (real-valued Zipfian) and normal distributions; the leaves of the B-tree split when they are full, and we measure the ratio of the maximal weight leaf to $1/n$. Lemma 13 tells us that this ratio should be asymptotically at most constant, but as fig. 5.2 shows, our experimental analysis shows further that this constant is generally less than 2. Because leaves generally split in 2, this makes some intuitive sense.

We also verify the these results using the InnoDB storage engine. We insert 5 million rows into a database using uniform, Pareto and normal distributions on the keys. the

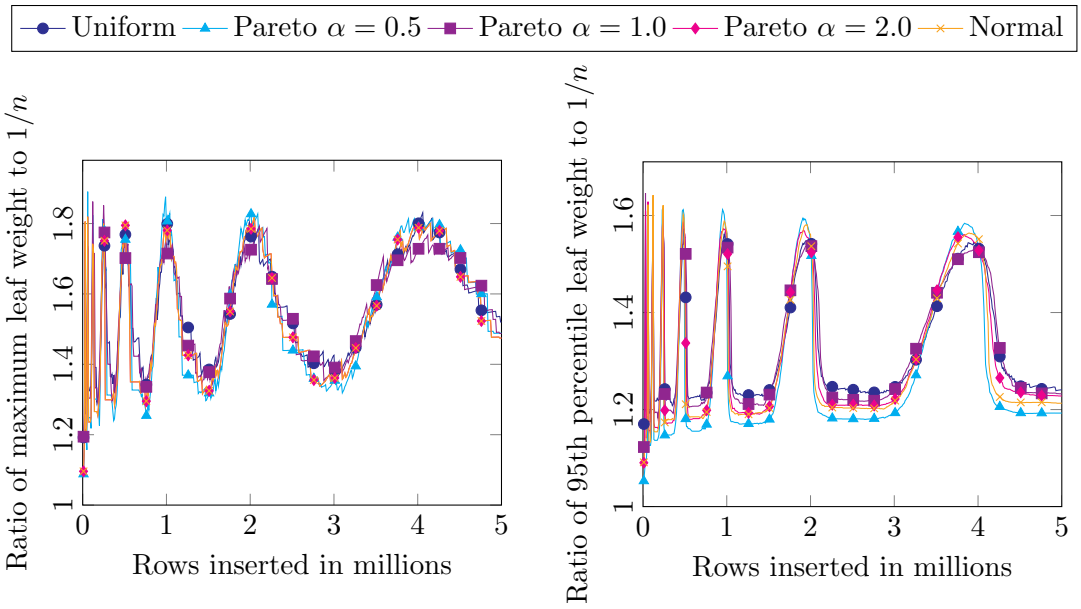


Figure 5.3: Deviation of the maximum and 95th percentile weight leaves from uniform as observed in InnoDB.

results are summarized in section 5.6.4. The maximum ratio does not exceed 2.3, and the 95th percentile ratio does not exceed 1.6. Thus the distribution of the keys to the leaves is in fact almost uniform.

5.6.5 Simulating Insertion Buffers

The ball-and-bins models described above are based on a static leaf structure. However, in practice inserting into a database causes the leaf structure (the number and probability distribution of bins in the model) to change. However, we can still perform the same strategies, and by simulating an insertion buffer in front of a database, we can compare their efficiency as well as verify that much of the static analysis empirically applies to the dynamic system.

We insert real-valued keys into the simulation according to one of several distributions of varying skewness: uniform on $[0, 1000]$, Pareto with parameter $\alpha = \{0.5, 1.0, 2.0\}$, and uniform centered at 0, with standard deviation 1000. We have a buffer which stores 2,500 keys; when it fills we choose a leaf according to the chosen strategy and flush all the buffered keys destined to it. Initially we have one leaf, and the leaves split when they exceed 160 keys, as uniformly as possible.

As shown in fig. 5.4, the key distribution doesn't affect the recycle rate of the insertion buffer, and as the number of leaves gets larger, the recycle rate decreases. Generally fullest bin does better than golden gate, and golden gate does better than random ball. Demonstrated with the normal distribution (all distributions perform very similarly), fig. 5.4f shows that golden gate initially outperforms random ball by about 30%, which then decreases as the number of bins grows.

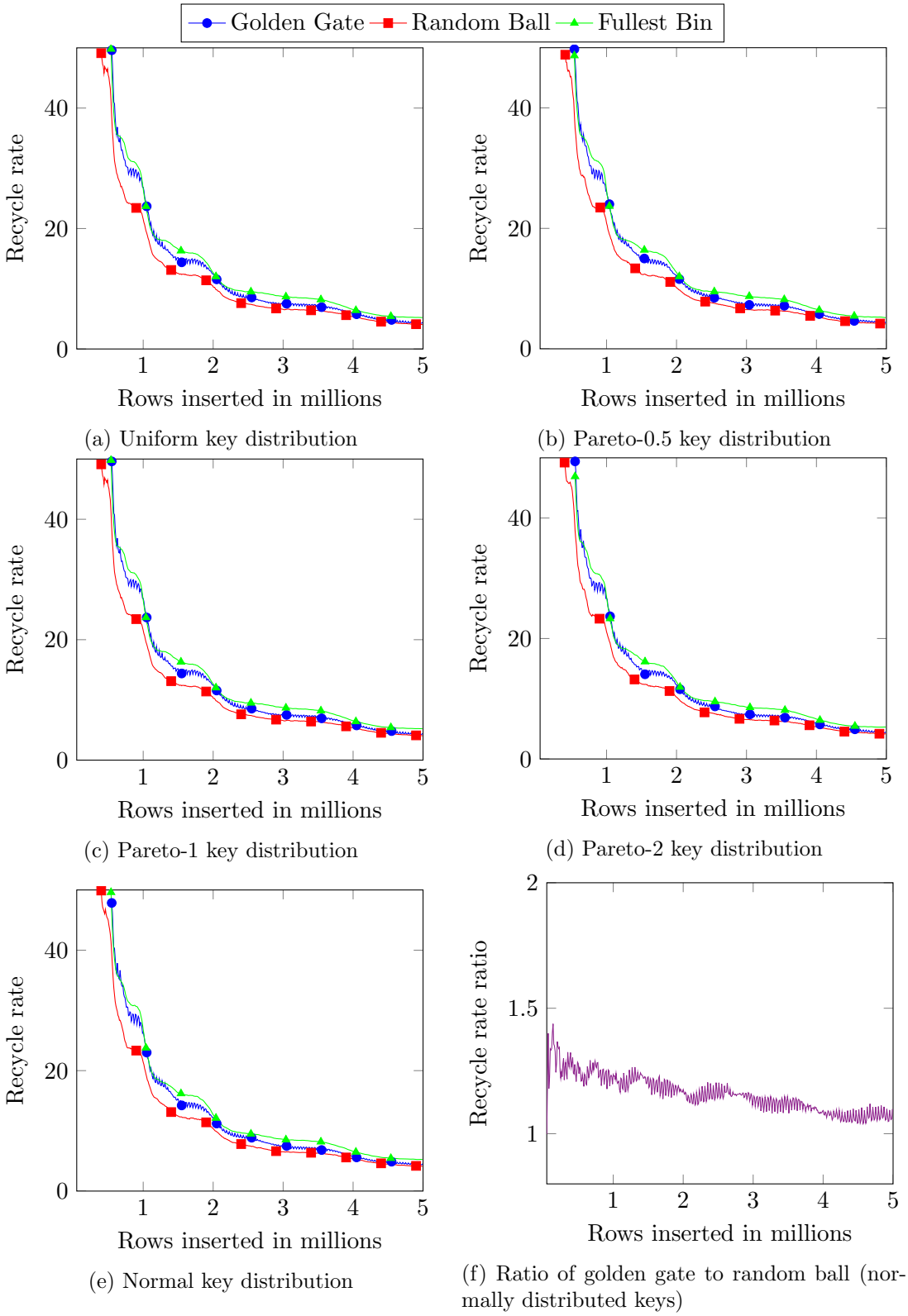
5.6.6 Real-World Performance (InnoDB)

In this section, we empirically evaluate the performance of insertion buffers in InnoDB, the default storage engine in MySQL.

Analogously to the experiments in section 5.6.5, we insert rows into the MySQL database, and after every 10000 insertions, we check the "merge ratio" reported by InnoDB. This is the number of rows merged into the database from the buffer during each buffer flush, and corresponds to the recycling rate in the balls and bins model. We also check the reported memory allocated to the buffer, which allows us to control for memory usage.

The keys of the rows are i.i.d. according to the same real-valued probability distributions as in section 5.6.5: uniform on $[0, 1000000]$, Pareto with parameter $\alpha = \{0.5, 1, 2\}$, and normal centered at 0 with standard deviation 1000. The results for the different distributions are shown in figs. 5.5a to 5.5e. The structure of the plot generally does not depend on the key distribution, and while there is more noise, the overall picture is similar to the plots in fig. 5.4.

If we were to hold the number of leaves roughly constant and change the buffer size, lemma 31 suggests that the relationship with recycle rate would be roughly linear. To test this, we ran the above experiment with buffer sizes from 8mb to 128mb in 2mb increments. We performed 11 million insertions with uniformly distributed keys each time, and then took the average recycle rate for the last million rows. As demonstrated in fig. 5.5f, the resulting plot is approximately linear.



(a) Uniform key distribution

(b) Pareto-0.5 key distribution

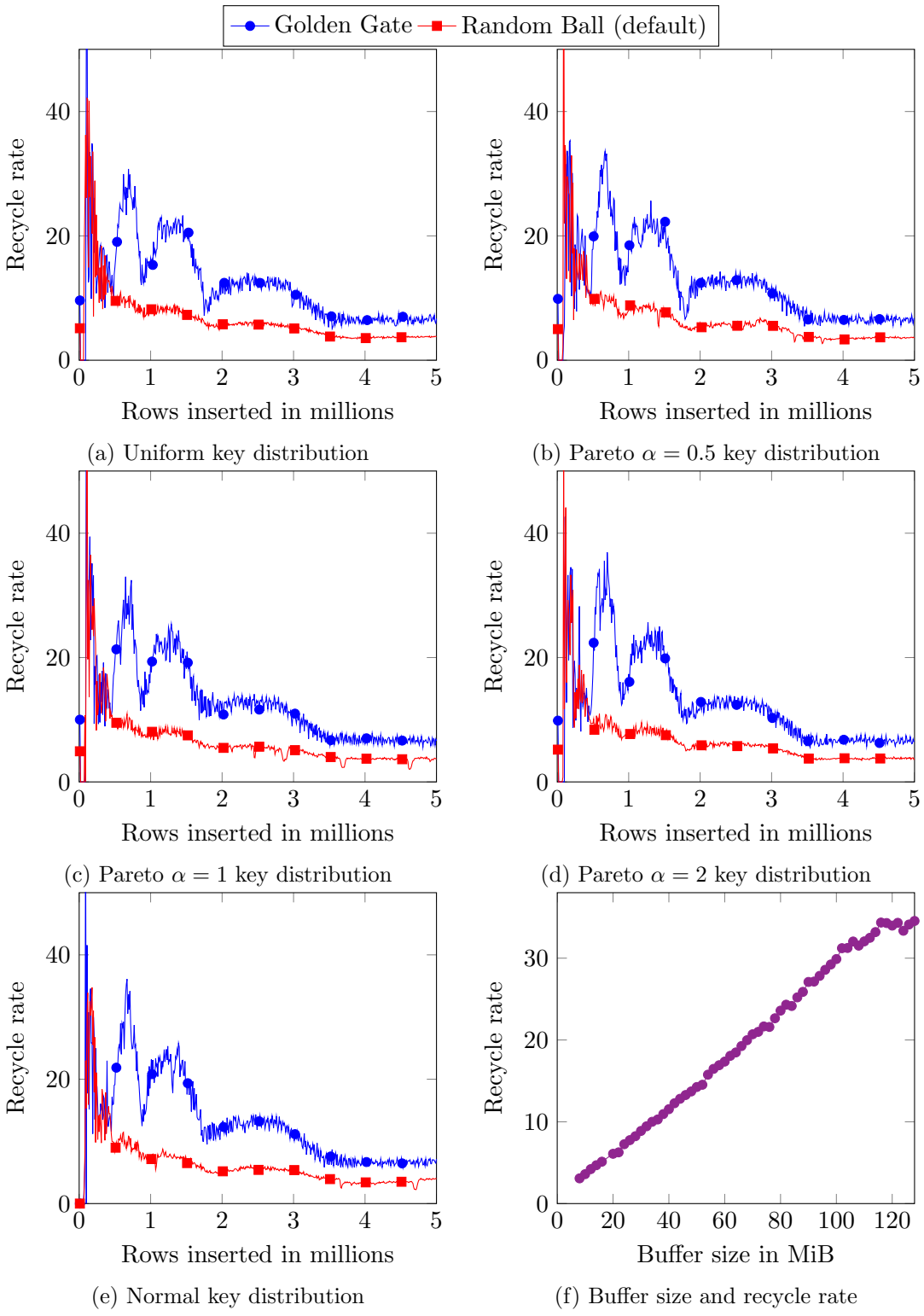
(c) Pareto-1 key distribution

(d) Pareto-2 key distribution

(e) Normal key distribution

(f) Ratio of golden gate to random ball (normally distributed keys)

Figure 5.4: Simulated results with various key distributions and recycling strategies. Recycle rates are taken over the latest 50,000 insertions. (Higher is better)



(a) Uniform key distribution

(b) Pareto $\alpha = 0.5$ key distribution

(c) Pareto $\alpha = 1$ key distribution

(d) Pareto $\alpha = 2$ key distribution

(e) Normal key distribution

(f) Buffer size and recycle rate

Figure 5.5: InnoDB Insertion buffer recycle rates for various key distributions and memory sizes. Recycling rate taken over last 10,000 insertions, except for fig. 5.5f, where it is taken over last 1 million insertions. (Higher is better)

References

[1] Fragging wonderful: The truth about defragging your ssd. Accessed 25 September 2016. URL: `http://www.pcworld.com/article/2047513/fragging-wonderful-the-truth-about-defragging-your-ssd.html`.

[2] Micah Adler, Petra Berenbrink, and Klaus Schröder. Analyzing an infinite parallel job allocation process. In Gianfranco Bilardi, Giuseppe F. Italiano, Andrea Pietracaprina, and Geppino Pucci, editors, *Algorithms - ESA '98, 6th Annual European Symposium, Venice, Italy, August 24-26, 1998, Proceedings*, volume 1461 of *Lecture Notes in Computer Science*, pages 417–428. Springer, 1998. `doi:10.1007/3-540-68530-8\_35`.

[3] Micah Adler, Soumen Chakrabarti, Michael Mitzenmacher, and Lars Eilstrup Rasmussen. Parallel randomized load balancing (preliminary version). In Frank Thomson Leighton and Allan Borodin, editors, *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, 29 May-1 June 1995, Las Vegas, Nevada, USA*, pages 238–247. ACM, 1995. `doi:10.1145/225058.225131`.

[4] Peyman Afshani, Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Mayank Goswami, and Meng-Tsung Tsai. Cross-referenced dictionaries and the limits of write optimization. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 1523–1532. SIAM, 2017. `doi:10.1137/1.9781611974782.99`.

[5] Alok Aggarwal and Jeffrey Scott Vitter. The I/O complexity of sorting and related problems (extended abstract). In Thomas Ottmann, editor, *Automata, Languages and Programming, 14th International Colloquium, ICALP87, Karlsruhe, Germany, July 13-17, 1987, Proceedings*, volume 267 of *Lecture Notes in Computer Science*, pages 467–478. Springer, 1987. `doi:10.1007/3-540-18088-5\_40`.

[6] Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Generating realistic *impressions* for file-system benchmarking. *TOS*, 5(4):16:1–16:30, 2009. `doi:10.1145/1629080.1629086`.

[7] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A five-year study of file-system metadata. *TOS*, 3(3):9:1–9:32, 2007. `doi:10.1145/1288783.1288788`.

[8] Woo Hyun Ahn, Kyungbaek Kim, Yongjin Choi, and Daeyeon Park. DFS: A defragmented file system. In *10th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2002),*

11-16 October 2002, Fort Worth, Texas, USA, pages 71–80. IEEE Computer Society, 2002. doi:10.1109/MASCOT.2002.1167062.

[9] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200, 1999. doi:10.1137/S0097539795288490.

[10] Microsoft Azure. 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, 2016.

[11] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: creating synergies between memory, disk and log in log structured key-value stores. In Dilma Da Silva and Bryan Ford, editors, *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, pages 363–375. USENIX Association, 2017. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/balmau.

[12] Michael A. Bender, Jonathan W. Berry, Rob Johnson, Thomas M. Kroeger, Samuel McCauley, Cynthia A. Phillips, Bertrand Simon, Shikha Singh, and David Zage. Anti-persistence on persistent storage: History-independent sparse tables and dictionaries. In Tova Milo and Wang-Chiew Tan, editors, *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 289–302. ACM, 2016. doi:10.1145/2902251.2902276.

[13] Michael A. Bender, Richard Cole, Erik D. Demaine, and Martin Farach-Colton. Scanning and traversing: Maintaining data for traversals in a memory hierarchy. In Rolf H. Möhring and Rajeev Raman, editors, *Algorithms - ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17-21, 2002, Proceedings*, volume 2461 of *Lecture Notes in Computer Science*, pages 139–151. Springer, 2002. doi:10.1007/3-540-45749-6_16.

[14] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious b-trees. *SIAM J. Comput.*, 35(2):341–358, 2005. doi:10.1137/S0097539701389956.

[15] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming b-trees. In Phillip B. Gibbons and Christian Scheideler, editors, *SPAA 2007: Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Diego, California, USA, June 9-11, 2007*, pages 81–92. ACM, 2007. URL: http://doi.acm.org/10.1145/1248377.1248393, doi:10.1145/1248377.1248393.

[16] Michael A. Bender, Martin Farach-Colton, Mayank Goswami, Rob Johnson, Samuel McCauley, and Shikha Singh. Bloom filters, adaptivity, and the dictionary problem. In Mikkel Thorup, editor, *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, pages 182–193. IEEE Computer Society, 2018. doi:10.1109/FOCS.2018.00026.

[17] Michael A. Bender, Martin Farach-Colton, Mayank Goswami, Dzejla Medjedovic, Pablo Montes, and Meng-Tsung Tsai. The batched predecessor problem in external memory. In Andreas S. Schulz and Dorothea Wagner, editors, *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, volume 8737 of *Lecture Notes in Computer Science*, pages 112–124. Springer, 2014. doi:10.1007/978-3-662-44777-2_10.

[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. Don't thrash: How to cache your hash on flash. *PVLDB*, 5(11):1627–1637, 2012. URL: http://vldb.org/pvldb/vol5/p1627_michaelabender_vldb2012.pdf.

[19] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. Don't thrash: How to cache your hash on flash. In Irfan Ahmad, editor, *3rd USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage'11, Portland, OR, USA, June 14, 2011*. USENIX Association, 2011. URL: https://www.usenix.org/conference/hotstorage11/dont-thrash-how-cache-your-hash-flash.

[20] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Simon Mauras, Tyler Mayer, Cynthia A. Phillips, and Helen Xu. Write-optimized skip lists. In Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 69–78. ACM, 2017. URL: http://doi.acm.org/10.1145/3034786.3056117, doi:10.1145/3034786.3056117.

[21] Itai Benjamini and Yury Makarychev. Balanced allocation: Memory performance tradeoffs. *CoRR*, abs/0901.1155, 2009. URL: http://arxiv.org/abs/0901.1155, arXiv:0901.1155.

[22] Petra Berenbrink, Tom Friedetzky, Peter Kling, Frederik Mallmann-Trenn, Lars Nagel, and Christopher Wastell. Self-stabilizing balls & bins in batches: The power of leaky bins [extended abstract]. In George Giakkoupis, editor, *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 83–92. ACM, 2016. doi:10.1145/2933057.2933092.

[23] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, and Julian Shun. Sorting with asymmetric read and write costs. In Guy E. Blelloch and Kunal Agrawal, editors, *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2015, Portland, OR, USA, June 13-15, 2015*, pages 1–12. ACM, 2015. doi:10.1145/2755573.2755604.

[24] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970. URL: http://doi.acm.org/10.1145/362686.362692, doi:10.1145/362686.362692.

[25] Jeff Bonwick. ZFS. In Paul Anderson, editor, *Proceedings of the 21th Large Installation System Administration Conference, LISA 2007, Dallas, Texas, USA, November 11-16, 2007*. USENIX, 2007. URL: http://www.usenix.org/events/lisa07/tech/bonwick_guru.pdf.

[26] Dhruba Borthakur. Rocksdb github wiki – performance benchmarks, 2013. URL: https://github.com/facebook/rocksdb/wiki/Performance-Benchmarks.

[27] Karl Bringmann, Thomas Sauerwald, Alexandre Stauffer, and He Sun. Balls into bins via local search: Cover time and maximum load. *Random Struct. Algorithms*, 48(4):681–702, 2016. doi:10.1002/rsa.20602.

[28] Gerth Stølting Brodal, Erik D. Demaine, Jeremy T. Fineman, John Iacono, Stefan Langerman, and J. Ian Munro. Cache-oblivious dynamic dictionaries with update/query tradeoffs. In Moses Charikar, editor, *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 1448–1456. SIAM, 2010. doi:10.1137/1.9781611973075.117.

[29] Gerth Stølting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA.*, pages 546–554. ACM/SIAM, 2003. URL: http://dl.acm.org/citation.cfm?id=644108.644201.

[30] Mark Callaghan. An obscure performance problem with the insert buffer. https://www.facebook.com/notes/mysql-at-facebook/an-obscure-performance-problem-with-the-insert-buffer/479735920932/, 2010.

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

[32] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Christian A. Lang, and Kenneth A. Ross. Buffered bloom filters on solid state storage. In Rajesh Bordawekar and Christian A. Lang, editors, *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2010, Singapore, September 13, 2010*, pages 1–8, 2010. URL: http://www.vldb.org/archives/workshop/2010/proceedings/files/vldb_2010_workshop/ADMS_2010/adms10-canim.pdf.

[33] Rémy Card, Theodore Ts'o, and Stephen Tweedie. Design and implementation of the Second Extended Filesystem. In *Proceedings of the First Dutch International Symposium on Linux*, pages 1–6, Amsterdam, NL, December 8–9 1994. URL: http://e2fsprogs.sourceforge.net/ext2intro.html.

[34] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. Hashkv: Enabling efficient updates in KV storage via hashing. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 1007–1019.

USENIX Association, 2018. URL: https://www.usenix.org/conference/atc18/presentation/chan.

[35] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In John R. Douceur, Albert G. Greenberg, Thomas Bonald, and Jason Nieh, editors, *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems, SIGMETRICS/Performance 2009, Seattle, WA, USA, June 15-19, 2009*, pages 181–192. ACM, 2009. doi:10.1145/1555349.1555371.

[36] Richard Cole, Alan M. Frieze, Bruce M. Maggs, Michael Mitzenmacher, Andréa W. Richa, Ramesh K. Sitaraman, and Eli Upfal. On balls and bins with deletions. In Michael Luby, José D. P. Rolim, and Maria J. Serna, editors, *Randomization and Approximation Techniques in Computer Science, Second International Workshop, RANDOM'98, Barcelona, Spain, October 8-10, 1998, Proceedings*, volume 1518 of *Lecture Notes in Computer Science*, pages 145–158. Springer, 1998. doi:10.1007/3-540-49543-6_12.

[37] Richard Cole, Bruce M. Maggs, Friedhelm Meyer auf der Heide, Michael Mitzenmacher, Andréa W. Richa, Klaus Schröder, Ramesh K. Sitaraman, and Berthold Vöcking. Randomized protocols for low congestion circuit routing in multistage interconnection networks. In Jeffrey Scott Vitter, editor, *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pages 378–388. ACM, 1998. doi:10.1145/276698.276790.

[38] 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. File systems fated for senescence? nonsense, says science! In Geoff Kuenning and Carl A. Waldspurger, editors, *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*, pages 45–58. USENIX Association, 2017. URL: https://www.usenix.org/conference/fast17/technical-sessions/presentation/conway.

[39] Alexander Conway, Martin Farach-Colton, and Philip Shilane. Optimal hashing in external memory. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, volume 107 of *LIPIcs*, pages 39:1–39:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.ICALP.2018.39.

[40] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum, editors, *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154. ACM, 2010. doi:10.1145/1807128.1807152.

[41] Artur Czumaj and Volker Stemann. Randomized allocation processes. *Random Struct. Algorithms*, 18(4):297–331, 2001. doi:10.1002/rsa.1011.

[42] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 79–94. ACM, 2017. doi:10.1145/3035918.3064054.

[43] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 505–520. ACM, 2018. doi:10.1145/3183713.3196927.

[44] Niv Dayan and Stratos Idreos. The log-structured merge-bush & the wacky continuum. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 449–466. ACM, 2019. doi:10.1145/3299869.3319903.

[45] Paul Deheuvels and Luc Devroye. Strong laws for the maximal k-spacing when $k \leq c \log n$. *Zeitschrift für Wahrscheinlichkeitstheorie und Verwandte Gebiete*, 66(3):315–334, 1984.

[46] Allen B. Downey. The structural cause of file size distributions. In *9th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2001), 15-18 August 2001, Cincinnati, OH, USA*, pages 361–370. IEEE Computer Society, 2001. doi:10.1109/MASCOT.2001.948888.

[47] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim M. Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing DRAM footprint with NVM in facebook. In Rui Oliveira, Pascal Felber, and Y. Charlie Hu, editors, *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 42:1–42:13. ACM, 2018. doi:10.1145/3190508.3190524.

[48] John Esmet, Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. The tokufs streaming file system. In Raju Rangaswami, editor, *4th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage'12, Boston, MA, USA, June 13-14, 2012*. USENIX Association, 2012. URL: https://www.usenix.org/conference/hotstorage12/workshop-program/presentation/esmet.

[49] Martin Farach-Colton, Rohan J. Fernandes, and Miguel A. Mosteiro. Bootstrapping a hop-optimal network in the weak sensor model. *ACM Trans. Algorithms*, 5(4):37:1–37:30, 2009. doi:10.1145/1597036.1597040.

[50] Apache Software Foundation. Apache Cassandra, 2019. URL: http://cassandra.apache.org.

[51] MariaDB Foundation. MariaDB Foundation, 2017. https://mariadb.org. URL: https://mariadb.org.

[52] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 285–298. IEEE Computer Society, 1999. doi:10.1109/SFFCS.1999.814600.

[53] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Scaling concurrent log-structured data stores. In Laurent Réveillère, Tim Harris, and Maurice Herlihy, editors, *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, pages 32:1–32:14. ACM, 2015. doi:10.1145/2741948.2741973.

[54] Inc. Google. Leveldb, 2019. URL: https://github.com/google/leveldb.

[55] Wilson C. Hsieh and William E. Weihl. Scalable reader-writer locks for parallel systems. In Viktor K. Prasanna and Larry H. Canter, editors, *Proceedings of the 6th International Parallel Processing Symposium, Beverly Hills, CA, USA, March 1992*, pages 656–659. IEEE Computer Society, 1992. doi:10.1109/IPPS.1992.222989.

[56] Yihe Huang, Matej Pavlovic, Virendra J. Marathe, Margo Seltzer, Tim Harris, and Steve Byan. Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 967–979. USENIX Association, 2018. URL: https://www.usenix.org/conference/atc18/presentation/huang.

[57] John Iacono and Mihai Patrascu. Using hashing to solve the dictionary problem. In Yuval Rabani, editor, *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 570–582. SIAM, 2012. URL: http://portal.acm.org/citation.cfm?id=2095164&CFID=63838676&CFTOKEN=79617016, doi:10.1137/1.9781611973099.

[58] IBM. 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, 2017.

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

[60] William Jannen, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Lazy analytics: Let other queries do the work for you. In Nitin Agrawal and Sam H. Noh, editors, *8th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2016, Denver, CO, USA, June 20-21, 2016*. USENIX Association, 2016. URL: https://www.usenix.org/conference/hotstorage16/workshop-program/presentation/jannen.

[61] 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. Betrfs: A right-optimized write-optimized file system. In Jiri Schindler and Erez Zadok, editors, *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*, pages 301–315. USENIX Association, 2015. URL: `https://www.usenix.org/conference/fast15/technical-sessions/presentation/jannen`.

[62] 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. Betrfs: Write-optimization in a kernel file system. *TOS*, 11(4):18:1–18:29, 2015. `doi:10.1145/2798729`.

[63] Cheng Ji, Li-Pin Chang, Liang Shi, Chao Wu, Qiao Li, and Chun Jason Xue. An empirical study of file-system fragmentation in mobile storage systems. In Nitin Agrawal and Sam H. Noh, editors, *8th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2016, Denver, CO, USA, June 20-21, 2016*. USENIX Association, 2016. URL: `https://www.usenix.org/conference/hotstorage16/workshop-program/presentation/ji`.

[64] Myoungsoo Jung and Mahmut T. Kandemir. Revisiting widely held SSD expectations and rethinking system-level implications. In Mor Harchol-Balter, John R. Douceur, and Jun Xu, editors, *ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '13, Pittsburgh, PA, USA, June 17-21, 2013*, pages 203–216. ACM, 2013. `doi:10.1145/2465529.2465548`.

[65] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-ri Choi. SLM-DB: single-level key-value store with persistent memory. In Arif Merchant and Hakim Weatherspoon, editors, *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*, pages 191–205. USENIX Association, 2019. URL: `https://www.usenix.org/conference/fast19/presentation/kaiyrakhmet`.

[66] Lodewijk Kallenberg. *Markov Decision Processes - version 2016.* 10 2016.

[67] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. Reaping the performance of fast NVM storage with udepot. In Arif Merchant and Hakim Weatherspoon, editors, *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*, pages 1–15. USENIX Association, 2019. URL: `https://www.usenix.org/conference/fast19/presentation/kourtis`.

[68] Bredley Kuszmaul. Tokutek White Paper: A Comparison Of Log-Structured Merge (LSM) And Fractal Tree Indexing, 2014. URL: `http://highscalability.com/blog/2014/8/6/tokutek-white-paper-a-comparison-of-log-structured-merge-lsm.html`.

[69] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In Jiri Schindler and Erez Zadok, editors, *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*, pages 273–286. USENIX Association, 2015. URL: https://www.usenix.org/conference/fast15/technical-sessions/presentation/lee.

[70] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 447–461. ACM, 2019. doi:10.1145/3341301.3359628.

[71] Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. Elasticbf: Elastic bloom filter with hotness awareness for boosting read performance in large key-value stores. In Dahlia Malkhi and Dan Tsafrir, editors, *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 739–752. USENIX Association, 2019. URL: https://www.usenix.org/conference/atc19/presentation/li-yongkun.

[72] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *TOS*, 13(1):5:1–5:28, 2017. doi:10.1145/3033273.

[73] Dongzhe Ma, Jianhua Feng, and Guoliang Li. A survey of address translation technologies for flash memories. *ACM Comput. Surv.*, 46(3):36:1–36:39, 2014. doi:10.1145/2512961.

[74] Leonardo Mármol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. NVMKV: A scalable, lightweight, ftl-aware key-value store. In Shan Lu and Erik Riedel, editors, *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, pages 207–219. USENIX Association, 2015. URL: https://www.usenix.org/conference/atc15/technical-session/presentation/marmol.

[75] Avantika Mathur, Mingming Cao, and Andreas Dilger. Ext4: The next generation of the ext3 file system. *login Usenix Mag.*, 32(3), 2007. URL: https://www.usenix.org/publications/login/june-2007-volume-32-number-3/ext4-next-generation-ext3-file-system.

[76] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Trans. Comput. Syst.*, 2(3):181–197, 1984. doi:10.1145/989.990.

[77] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: random write considered harmful in solid state drives. In William J. Bolosky and Jason Flinn, editors, *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012*, page 12. USENIX

Association, 2012. URL: https://www.usenix.org/conference/fast12/sfs-random-write-considered-harmful-solid-state-drives.

[78] Michael Mitzenmacher. Load balancing and density dependent jump markov processes (extended abstract). In *37th Annual Symposium on Foundations of Computer Science, FOCS '96, Burlington, Vermont, USA, 14-16 October, 1996*, pages 213–222. IEEE Computer Society, 1996. doi:10.1109/SFCS.1996.548480.

[79] Michael Mitzenmacher. On the analysis of randomized load balancing schemes. *Theory Comput. Syst.*, 32(3):361–386, 1999. doi:10.1007/s002240000122.

[80] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1094–1104, 2001. doi:10.1109/71.963420.

[81] NuDB. Nudb: A fast key/value insert-only database for ssd drives in c++11. https://github.com/vinniefalco/NuDB, 2016.

[82] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996. doi:10.1007/s002360050048.

[83] Oracle. Tuning the database buffer cache. https://docs.oracle.com/database/121/TGDBA/tune_buffer_cache.htm#TGDBA294, 2017.

[84] Oracle, Inc. Introduction to innodb, 2017. https://dev.mysql.com/doc/refman/5.6/en/innodb-introduction.html. URL: https://dev.mysql.com/doc/refman/5.6/en/innodb-introduction.html.

[85] Oracle, Inc. MySQL, 2017. https://www.mysql.com. URL: https://www.mysql.com.

[86] Prashant Pandey, Michael A. Bender, Rob Johnson, and Robert Patro. A general-purpose counting filter: Making every bit count. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 775–787. ACM, 2017. doi:10.1145/3035918.3035963.

[87] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In Ajay Gulati and Hakim Weatherspoon, editors, *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, pages 537–550. USENIX Association, 2016. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/papagiannis.

[88] Gahyun Park. A generalization of multiple choice balls-into-bins: Tight bounds. *Algorithmica*, 77(4):1159–1193, 2017. doi:10.1007/s00453-016-0141-z.

[89] Marina Petrova, Natalia Olano, and Petri Mähönen. Balls and bins distributed load balancing algorithm for channel allocation. In *WONS '10*.

[90] Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash-, and space-efficient bloom filters. *ACM Journal of Experimental Algorithmics*, 14, 2009. doi:10.1145/1498698.1594230.

[91] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 497–514. ACM, 2017. doi:10.1145/3132747.3132765.

[92] Ohad Rodeh. B-trees, shadowing, and clones. *TOS*, 3(4):2:1–2:27, 2008. doi:10.1145/1326542.1326544.

[93] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: the linux b-tree filesystem. *TOS*, 9(3):9:1–9:32, 2013. doi:10.1145/2501620.2501623.

[94] Drew S. Roselli, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. In *Proceedings of the General Track: 2000 USENIX Annual Technical Conference, June 18-23, 2000, San Diego, CA, USA*, pages 41–54. USENIX, 2000. URL: http://www.usenix.org/publications/library/proceedings/usenix2000/general/roselli.html.

[95] SAP. Rlv data store for write-optimized storage. http://help-legacy.sap.com/saphelp_iq1611_iqnfs/helpdata/en/a3/13783784f21015bf03c9b06ad16fc0/content.htm, 2017.

[96] Jeanette P. Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff-hoeffding bounds for applications with limited independence. *SIAM J. Discrete Math.*, 8(2):223–250, 1995. doi:10.1137/S089548019223872X.

[97] Inc. Scylla. ScyllaDB: The real-time big data database, 2019. URL: https://www.scylladb.com.

[98] Pradeep Shetty, Richard P. Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building workload-independent storage with vt-trees. In Keith A. Smith and Yuanyuan Zhou, editors, *Proceedings of the 11th USENIX conference on File and Storage Technologies, FAST 2013, San Jose, CA, USA, February 12-15, 2013*, pages 17–30. USENIX, 2013. URL: https://www.usenix.org/conference/fast13/technical-sessions/presentation/shetty.

[99] Keith A. Smith and Margo I. Seltzer. File system aging - increasing the relevance of file system benchmarks. In John Zahorjan, Albert G. Greenberg, and Scott T. Leutenegger, editors, *Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, Seattle, Washington, USA, June 15-18, 1997*, pages 203–213. ACM, 1997. doi:10.1145/258612.258689.

[100] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS file system. In *Proceedings of the USENIX Annual Technical Conference, San Diego, California, USA, January 22-26, 1996*, pages 1–14. USENIX Association, 1996. URL: http://www.usenix.org/publications/library/proceedings/sd96/sweeney.html.

[101] Tokutek, Inc. TokuDB and TokuMX, 2014. http://www.tokutek.com. URL: http://www.tokutek.com.

[102] Stephen Tweedie. EXT3, journaling filesystem. In *Ottowa Linux Symposium*, Ottowa, ON, Canada, July 20 2000.

[103] Vijay Vasudevan, Michael Kaminsky, and David G. Andersen. Using vector interfaces to deliver millions of IOPS from a networked key-value storage server. In Michael J. Carey and Steven Hand, editors, *ACM Symposium on Cloud Computing, SOCC '12, San Jose, CA, USA, October 14-17, 2012*, page 8. ACM, 2012. doi:10.1145/2391229.2391237.

[104] Vertica. Wos (write optimized store). https://my.vertica.com/docs/7.1.x/HTML/Content/Authoring/Glossary/WOSWriteOptimizedStore.htm, 2017.

[105] Berthold Vöcking. How asymmetry helps load balancing. *J. ACM*, 50(4):568–589, 2003. doi:10.1145/792538.792546.

[106] R. West, P. Zaroo, C.A. Waldspurger, X. Zhang, and H. Zheng. Online computation of cache occupancy and performance, July 19 2016. US Patent 9,396,024.

[107] Wikipedia. Thomae's function — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Thomae's%20function&oldid=837510765, 2018. [Online; accessed 28-April-2018].

[108] Lars Wirzenius, Joanna Oja, Stephen Stafford, and Alex Weeks. *Linux System Administrator's Guide*. The Linux Documentation Project, 2004. Version 0.9. URL: http://www.tldp.org/LDP/sag/sag.pdf.

[109] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items. In Shan Lu and Erik Riedel, editors, *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, pages 71–82. USENIX Association, 2015. URL: https://www.usenix.org/conference/atc15/technical-session/presentation/wu.

[110] Jimmy Xiang. Apache hbase write path. http://blog.cloudera.com/blog/2012/06/hbase-write-path/, 2012.

[111] Ting Yao, Jiguang Wan, Ping Huang, Yiwen Zhang, Zhiwen Liu, Changsheng Xie, and Xubin He. Geardb: A gc-free key-value store on HM-SMR drives with gear compaction. In Arif Merchant and Hakim Weatherspoon, editors, *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*, pages 159–171. USENIX Association, 2019. URL: https://www.usenix.org/conference/fast19/presentation/yao.

[112] 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. Optimizing every operation in a write-optimized file system. In Angela Demke Brown and Florentina I. Popovici, editors, *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*, pages 1–14. USENIX Association, 2016. URL: https://www.usenix.org/conference/fast16/technical-sessions/presentation/yuan.

[113] Ningning Zhu, Jiawu Chen, and Tzi-cker Chiueh. TBBT: scalable and accurate trace replay for file server evaluation. In Garth Gibson, editor, *Proceedings of the FAST '05 Conference on File and Storage Technologies, December 13-16, 2005, San Francisco, California, USA*. USENIX, 2005. URL: `http://www.usenix.org/events/fast05/tech/zhu.html`.

[114] Pengfei Zuo, Yu Hua, and Jie Wu. Level hashing: A high-performance and flexible-resizing persistent hashing index structure. *TOS*, 15(2):13:1–13:30, 2019. `doi:10.1145/3322096`.