A Practical Concurrent Index for Solid-State Drives

Risi Thonangi
Duke University
rvt@cs.duke.edu

Shivnath Babu
Duke University
shivnath@cs.duke.edu

Jun Yang
Duke University
junyang@cs.duke.edu

ABSTRACT

Solid-state drives are becoming a viable alternative to magnetic disks in database systems, but their performance characteristics, particularly those caused by their erase-before-write behavior, make conventional database indexes a poor fit. There have been various proposals of indexes specialized for these devices, but to make such indexes practical, we must address the issue of concurrency control. Good concurrency control is especially critical to indexes on solid-state drives, because they typically rely on batch updates, which may take long and block concurrent index accesses. We design, implement, and evaluate an index structure called *FD+tree* and an associated concurrency control scheme called *FD+FC*. Our evaluation confirms significant performance advantages of our approach over less sophisticated ones, and brings out insights on data structure design and OLTP performance tuning on solid-state drives.

1 Introduction

Solid-State Drives (*SSDs*) have become a viable alternative to magnetic disks in database systems [9, 12, 4]. SSDs perform random reads one to two orders of magnitude faster than magnetic disks. However, a write may necessitate first erasing a large region of data (called an *erase block*). This erase-before-write nature makes random writes one to two orders of magnitude slower than reads. SSDs' fast random reads benefit tree indexes like the B+tree used extensively in database systems. However, the conventional B+tree performs random in-place writes, making it a poor fit for SSDs.

The unique characteristics of SSDs have led database researchers to new tree indexes such as *BFTL* [22], *LA-tree* [2], *FD-tree* [13], and *SkimpyStash* [7]. A foundational idea behind these new indexes is to convert the small random writes caused by index modifications into large sequential writes, by somehow buffering modifications and then updating the index with a batch reorganization. Such reorganizations may take long, as illustrated by Figure 1. While they make efficient use of SSD characteristics, there is a serious issue: if the index does not employ proper concurrency control techniques, an ongoing reorganization can prevent concurrent index accesses, hence causing large variance in access latency. For example, without proper concurrency control, the completion times shown in Figure 1 would translate into response times experienced by concurrent accesses: some accesses may complete in milliseconds or less, but others blocked by a long-running reorganization can take seconds. Thus, there is a particularly pressing need for efficient concurrency control for tree indexes on SSDs, which stems from potentially long-running index reorganizations—an issue not found in traditional indexes designed for magnetic disks.

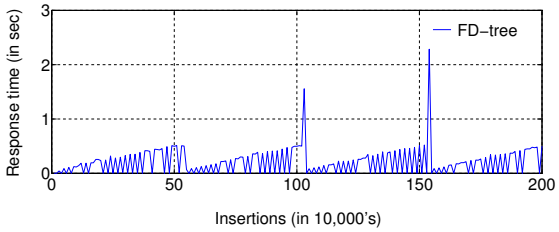

Figure 1: Completion time of an insertion request—including any index reorganization triggered—for (an improved version of) FD-tree [13], over the course of two million insertions, starting with an empty index. The insertions are grouped into buckets each containing 10,000 requests; we plot the longest observed time per request in each bucket.

An index with good concurrency control should do better on three crucial requirements: a) low average access latency, b) low variance across latencies, and c) low worst-case latency. While most database performance research and specifically those on SSD indexes focus on (a), requirements (b) and (c) are equally (perhaps more) important in practice. Users feel variance in performance more than they feel the average [15]. Engineers at Web-based companies like Facebook and Amazon are concerned about minimizing variance and ensuring that the "edge cases" are not bad, even at the potential cost of higher average latency [1, 8].

Contributions First, we identify concurrency as a critical issue in making SSD indexes practical. We show that straightforward concurrency control schemes are inadequate. A global readers-writer lock incurs unacceptably high variances and worst-case access latencies. An alternative is for each index reorganization to write a new version of the updated portion of the index and "switch it in" at the end of the organization; hence, readers can access the old copy of the index without being blocked. However, this scheme doubles the space requirement, making it unattractive for SSDs, which continue to be much smaller and more expensive than magnetic disks. Furthermore, as we shall see later in the paper, because crucial resources held by the old copy cannot be devoted to incoming modifications until the current organization finishes, this scheme continues to suffer from worst-case modification latency as high as using a global readers-writer lock.

We propose *FD+FC*, a novel indexing and concurrency control scheme for SSDs. FD+FC allows concurrent accesses by both

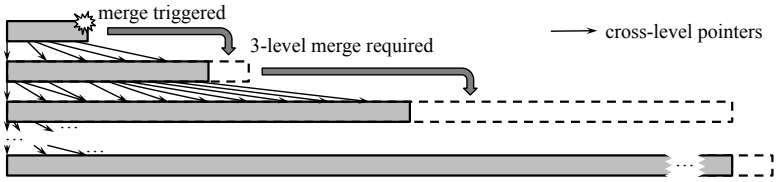

Figure 2: Illustration of high-level ideas in FD+tree. Maximum capacity of each level is delineated by dashed lines.

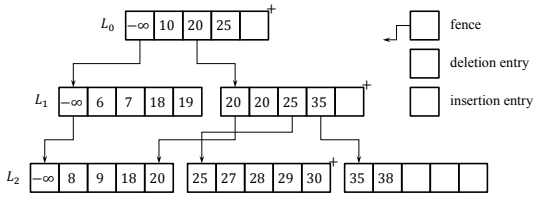

Figure 3: An example FD+tree. Blocks visited by the lookup for key 25 are marked with "+".

readers and writers during ongoing index reorganizations, improving both response time and throughput of the index. Furthermore, FD+FC does so without the extra space requirement of a space-doubling scheme. Achieving these features requires careful design of the data structure and algorithms. At a high level, FD+FC employs a reorganization procedure that sweeps a wavefront across a portion of the index, progressively converting it while ensuring that the converted and unconverted parts remain connected as a coherent structure supporting concurrent accesses. Most parts of the index have a single sequential writer and multiple random readers; this special access pattern is exploited by FD+FC's efficient concurrency control protocol. We have implemented and empirically evaluated FD+FC against alternative schemes. To our knowledge, we are the first to evaluate concurrency control for SSD indexes. We include a full-fledged empirical comparison of FD+FC, its alternatives, and *Berkeley DB* [18], an highly optimized industry-strength B+tree implementation with concurrency control.

Finally, the basis for FD+FC is an index called the *FD+tree*, which modifies and extends the FD-tree proposed by Li et al. [13]. FD+tree is a contribution in its own right because of several new features aimed at making it practical: *one-pass merge* makes index reorganization more efficient and simpler for concurrency control; *level skipping* speeds up reads by skipping small, unnecessary levels; *level tightening* makes it possible for the tree to shrink in height in the presence of deletions; and *underflow-triggered merges* provide performance guarantees for workloads involving deletions.

2 FD+Tree without Concurrency Control

Overview and Intuition Before presenting our full indexing and concurrency control scheme, we need to describe its underlying index, FD+tree. As mentioned earlier, FD+tree modifies and extends FD-tree [13]. Conceptually, both indexes employ the techniques of the *logarithmic method* [3] and *fractional cascading* [6].

The logarithmic method turns in-place random writes into batch sequential writes. Data reside across layers of sequential files ("levels") whose maximum capacity increases geometrically from top to bottom. Modifications are not applied in place, but are instead added to the top level. Upon reaching its maximum capacity, a level is merged into a lower one; sometimes multiple levels will need to be merged so that the result level is within its maximum capacity. Figure 2 illustrates the idea.

Fractional cascading speeds up search across levels. Each level is sorted by key. By strategically placing, in each level, pointers to locations within the next level, we leverage the effort of searching a level in searching the next level, as illustrated in Figure 2. Without fractional cascading, each level would be searched from scratch.

Beyond conceptual similarities, FD+tree differs from FD-tree in significant ways, which we highlight in Section 2.4. Our overarching goal is to make the index more practical, with performance guarantees for workloads involving deletions, simpler and more efficient index reorganization, etc. To this end, FD+tree maintains a richer set of data structure invariants, and performs more careful bookkeeping, pre-reorganization planning, and post-reorganization adjustment. A representative example of the differences between FD+tree and FD-tree is how they handle the case when multiple levels need to be merged. Not knowing in advance which level the merge would stop at, FD-tree merges records down one level at a time until the result fits. Though *conceptually* simple, this approach is *executionally* complex and suboptimal—it writes upper levels multiple times and generates multiple intermediate index states violating invariants—which complicates concurrency control. In contrast, FD+tree determines the number of levels involved in the merge upfront, and merges multiple levels in a single pass. Though planning is conceptually more complex, it is computationally trivial, and leads to more efficient and much simpler execution that is better suited for concurrency.

2.1 Data Structure

For simplicity of presentation, we assume a unique index; i.e., there is at most one record with any given key value. Extension to handle duplicates is straightforward.

An FD+tree (illustrated in Figure 3) consists of a sequence of *levels* denoted $L_0, L_1, \ldots, L_{h-1}$, where h is the height of the tree. Levels below L_0 are linked lists of disk blocks[1] compactly storing sorted runs of entries. L_0 is a standard data structure (e.g., a B+tree) that supports lookup, in-place insert and delete, and ordered scan. L_0 is small enough that we keep it in main memory; persistence can be achieved by separate logging.[2]

An FD+tree level can be *skipped* or *materialized* (and alternate between these states over time). The top and bottom levels (L_0 and L_{h-1}) are always materialized. Intuitively, lookups bypass skipped levels, thereby saving disk accesses. In this paper, when we refer to the level above (or below) L_i, we mean the next *materialized* level above (or below) L_i, denoted $L_{\mathsf{prec}(i)}$ (or $L_{\mathsf{succ}(i)}$, respectively).

The entries in an FD+tree have two types: *data* and *fence*. Every entry has a *key* and a *payload*. A data entry can be an *insert* or *delete* entry. For a data entry, the payload contains the record pointer or value being indexed. For a fence entry f, the payload contains a pointer to a block in the next materialized level; all entries in that block have keys no less than f's key. The bottom level, L_{h-1}, has no fences or delete entries.

In the presence of deletions, the tree may contain multiple data entries with the same key; e.g., a delete entry can effectively "cancel out" an insert entry with the same key in a lower level. Therefore, the tree may be "bloated" in the sense that it stores more entries than necessary. To limit bloating, we maintain two counters, N_Δ and N_∇, which respectively track the total number (across all levels) of insert data entries and that of delete data entries currently in the tree. While the total number of data entries in the tree is $N_\Delta + N_\nabla$, the *true* number of elements indexed is $N_\Delta - N_\nabla$ because every delete entry cancels out exactly one insert entry.

[1] In this paper, a "block" refer to a page or "block" in the traditional sense of the word, i.e., a unit of disk transfer. It does *not* refer to an erase block.
[2] Alternatively, L_0 can be stored in a so-called *locality area* in an SSD where random writes have similar performance as sequential writes [5, 13].

Let β denote the *block size*, as measured by the number of entries that fit within one block. Let γ denote the *size ratio* parameter that controls how fast the maximum size grows between adjacent levels (see (I3) below). Let $B(L_i)$ denote the actual size of level L_i in the number of blocks. An FD+tree maintains the following invariants (FD-tree maintains only the first three):

- **(I1)** *All-blocks-fenced*: For every block of L_i (for all $i > 0$) there is a fence in the level above pointing to that block.
- **(I2)** *Fence-first*: The first entry in every block of L_i (for all $i \geq 0$) is always a fence.[3]
- **(I3)** *Max-size*: Let κ_i denote the maximum size allowed for L_i measured in blocks. $B(L_i) \leq \kappa_i$ for all $i \geq 0$; and $\kappa_i = \gamma\kappa_{i-1} = \gamma^i\kappa_0$ for all $i > 0$.
- **(I4)** *Min-size*: If $h > 2$, $B(L_{h-1}) > \kappa_{h-2}$; i.e., the bottom level has so much data that it cannot be stored as a higher level.
- **(I5)** *Skip-limit*: For any level L_i ($i < h-1$), $B(L_{\mathsf{succ}(i)}) \leq \kappa_{i+1}$; i.e., we can skip level(s) below L_i only if L_i is not required to store more than the maximum number of fences that it is supposed to store for L_{i+1}.
- **(I6)** *No-underflow*: $\frac{N_\nabla}{N_\Delta} \leq \frac{1}{3}$, or, equivalently, $\frac{N_\Delta - N_\nabla}{N_\Delta + N_\nabla} \geq \frac{1}{2}$, which guarantees that the number of data entries stored by the tree is at most twice the true number of elements indexed.

2.2 Modification and Lookup

For an insertion, we simply add an insert data entry to L_0. For a deletion, we check whether L_0 contains an insert entry with the same key: if yes, we delete that entry and decrement N_Δ; otherwise, the entry being deleted is below L_0, so we add a delete entry to L_0 and increment N_∇. An update is handled as a deletion followed by an insertion.

If the new entries we add to L_0 overflow it, we trigger a *merge*, as described later in Section 2.3. If $\frac{N_\nabla}{N_\Delta} > \frac{1}{3}$ (i.e., (I6) is violated), we also trigger a merge. We call these two types of merges *overflow-triggered* and *underflow-triggered*, respectively.

Lookup for a given key proceeds top-down through the levels, starting with L_0. On a level L_i, we look for all data entries with key. The modification procedure for L_0 and the merge procedure guarantee that there are only four cases: L_i has 1) no data entries with key; 2) a single insert entry with key; 3) a single delete entry with key; or 4) one delete entry with key followed by one insert entry with key. In Case 3, we report that key is not found and stop. In Cases 2 and 4, we return the insert entry and stop. In Case 1, we look in L_i for the fence f with the largest key no greater than key.[4] The search continues to the block in $L_{\mathsf{succ}(i)}$ pointed to by f. If we encounter Case 1 in L_{h-1}, we report that key is not found and stop. Figure 3 illustrates the process of lookup.

2.3 Merge

Merge reads and replaces the top $m+1$ levels ($L_0, \ldots, L_m$) for some $m \geq 1$. Levels below L_m that exist before the merge are not disturbed. Some new levels may be skipped; let $L_{\overline{m}}$ be the last new level that is materialized. $L_{\overline{m}}$ consolidates and stores all data entries from the old $m+1$ levels; $L_{\overline{m}}$ also stores fences to $L_{\mathsf{succ}(\overline{m})}$ (if any). When the merge ends, new levels above $L_{\overline{m}}$ store only fences. Figure 4 illustrates how merge works. Here, $m = \overline{m} = 3$, and note that level L_1 is skipped. In practice, $\overline{m}$ can be less than m when many input entries cancel each other.

[3]To simplify the discussion of how to maintain this invariant, assume we index a dummy data entry with key $-\infty$, which precedes all real keys.

[4]It is easy to see that this fence must be found on the same block as the data entries with key, because the entries are sorted by key and the first entry in a block is always a fence (I2).

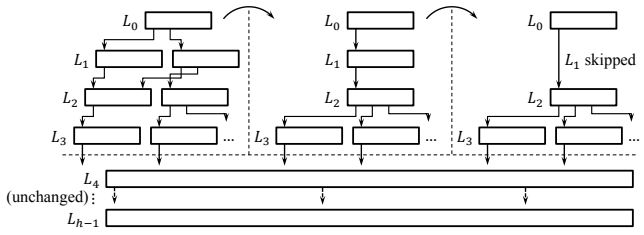

Figure 4: Illustration of FD+tree merge and motivation for level skipping. From right to left, we show the old levels, the new levels that would have been produced by merge without level skipping, and the actual new levels produced by merge (with level skipping). FD+FC directly produces the new levels on the right without going through the state in the middle.

Suppose the tree height is h before the merge. A merge that reads all levels is called a *full merge*. A full merge with $\overline{m} = h$ grows the tree by one level. A full merge with $\overline{m} < h-1$ shrinks the tree (possibly by more than one level).

Merge has three steps detailed below: *merge-prepare* determines which levels are involved; *merge-execute* combines the old levels into a new one, and creates fence-only upper levels as needed; finally, *merge-finalize* adjusts the merge result by skipping unnecessary fence-only levels and/or shrinking the tree when possible.

Merge-Prepare We first determine m, i.e., which levels are to be replaced. For an underflow-triggered merge, we always do a full merge, and $m = h-1$. For an overflow-triggered merge, we calculate m as follows. An upper bound on the size (in blocks) of the run obtained by merging $L_0, \ldots, L_i$ is:

$$\widehat{U}(i) = \begin{cases} \left\lceil \sum_{j=0}^{i} B(L_j) - \frac{1}{\beta} \sum_{j=1}^{i} B(L_j) \right\rceil & \text{for } i < h-1; \\ \lceil (N_\Delta - N_\nabla)/\beta \rceil & \text{for } i = h-1. \end{cases} \quad (1)$$

The second summation in the case for $i < h-1$ is a lower bound on the total number of fences in levels above L_i, which will be ignored by the merge. In the case of $i = h-1$, we in fact know the exact size of the run with the help of counters N_Δ and N_∇. We set

$$m = \begin{cases} h & \text{if } \widehat{U}(h-1) > \kappa_{h-1}; \\ \arg\min_i(\widehat{U}(i) \leq \kappa_i) & \text{otherwise.} \end{cases} \quad (2)$$

In other words, we try to merge as few levels as possible provided that the last level has enough space to accommodate the result.[5]

There are two points worth noting about merge-prepare. First, calculating $\widehat{U}(i)$ and m is computationally trivial. Second, while accurate for insertion-only workloads, $\widehat{U}(i)$ can be a conservative estimate in the presence of deletions, because a sequence of insert and delete entries with the same key can be merged into as few as zero entries. It turns out that we can easily extend FD+tree with some additional bookkeeping such that we can always accurately determine the merge result size and the optimal choice of m, but we find the simpler approach here suffices practically. See Remark A.1 in appendix for a detailed discussion.

Merge-Execute To execute the merge, we read all materialized levels among $L_0, \ldots, L_m$ in key order in parallel and output the new levels $L_0^{\mathsf{new}}, \ldots, L_m^{\mathsf{new}}$.[6]

[5]In the worst case, $\widehat{U}(h-1) < \kappa_h$, so a full merge of all h levels into a new L_h (thereby growing the tree by one level) will surely work.

[6]Note that we could reclaim space in $L_0, \ldots, L_m$ as we process their entries, so the total space taken by old and new levels is roughly no more than that taken by the old levels when the merge started. We ignore the details here, but will revisit this point in Sections 3 and 4.

During the merge, we maintain p_{last}, a pointer to the last block in $L_{\mathsf{succ}(m)}$ for which we have added a fence to L_m^{new}. This information is needed to ensure (I2) and, specifically, that the first entry in every L_m^{new} block is a fence.

Consider all entries in the old $L_0, \ldots, L_m$ with the smallest key key yet to be processed. If among them is a fence f pointing to $L_{\mathsf{succ}(m)}$, we will add f to L_m^{new}. Fences pointing to L_m or above are simply ignored. Let $S_0, \ldots, S_m$ denote the sets of data entries with key from $L_0, \ldots, L_m$. We *coalesce* these data entries into a final set S to add to L_m^{new}. S captures the net effect of applying the insert and delete operations in $S_m, S_{m-1}, \ldots, S_0$ in order. It is easy to see that $0 \leq |S| \leq 2$ and there are just four cases as described in Section 2.2. We update N_Δ and N_∇ to reflect the effect of replacing $S_m, S_{m-1}, \ldots, S_0$ with S.

If the last block of L_m^{new} has enough space to accommodate f (when applicable) and S, we simply add them and move on to the next key. If f was added to L_m^{new}, we also update p_{last}.

If L_m^{new} has insufficient space, we begin a new block of L_m^{new} to add f (when applicable) and S. To maintain (I2), the first entry for this block needs to be a fence. If we happen to have f to add, we are fine; otherwise, we add a fence with key and p_{last} as the first entry of the new block.

When we begin a new block of L_m^{new}, we need to add to $L_{\mathsf{prec}(m)}^{\mathsf{new}}$ a fence to this block. If $L_{\mathsf{prec}(m)}^{\mathsf{new}}$'s last block has no space left, we begin a new block for $L_{\mathsf{prec}(m)}^{\mathsf{new}}$ to put the fence in (which automatically satisfies (I2)). A fence to this new block must then be added to $L_{\mathsf{prec}(\mathsf{prec}(m))}^{\mathsf{new}}$. Such fence additions may propagate all the way up to L_0^{new}; merge-prepare ensures that L_0^{new} has enough space.

After all entries from the old $L_0, \ldots, L_m$ have been processed, $L_0^{\mathsf{new}}, \ldots, L_m^{\mathsf{new}}$ become the new $L_0, \ldots, L_m$.

Merge-Finalize The result of merge-execute needs further tweaking for several reasons. The first reason is that we need a way for an FD+tree to shrink in height, when there have been enough deletions and a merge produces a bottom level violating (I4).

The second reason is more subtle. Performance becomes suboptimal when large merges generate long chains of single-block levels. After merge-execute, the new levels above L_m store only fences. The sizes of these levels decrease rapidly at the rate of $1/\beta$, so typically, all but a few levels above L_m would have one block each, as illustrated in Figure 4. Hence, lookups perform poorly. This inefficiency may be further aggravated by merge-prepare's overestimation of the merge result size, as discussed earlier.

Thus, to allow an FD+tree to shrink, and to guard against inefficiency, we take the third step of merge, merge-finalize. Based on the actual sizes of the new levels produced by merge-execute, merge-finalize skips new levels that are unnecessary, and adjusts level numbers for the remaining materialized new levels. It may modify L_0, but no other levels' contents.

For a new level L_i produced by merge-execute, we compute o_i, a new level number for L_i, as follows:

$$o_i = \arg\min_j \left(B(L_i) \leq \kappa_j \wedge B(L_{\mathsf{succ}(i)}) \leq \kappa_{j+1} \right). \quad (3)$$

The two operands of the conjunction above correspond to (I3) and (I5), respectively. Intuitively, we "tighten" the level number for a new level as much as possible while preserving all invariants.

Let $z = \arg\max_i (o_i = 0)$; i.e., L_z is the last new level that can be relabeled L_0. Merge-finalize proceeds as follows. 1) If $z > 0$, replace L_0's content with that of L_z, and skip levels $L_1, L_2, \ldots, L_z$. 2) For each materialized level L_i with $z < i \leq m$, relabel it as L_{o_i} unless $i = m$ and L_m is not the bottom level.[7]

[7] For a technical reason that we discuss in Remark A.3, we do not adjust the last new level's number unless we have a full merge.

Figure 4 illustrates how merge-finalize improves lookup performance by reducing the effective tree height.

2.4 Discussion

FD+tree's performance guarantees are summarized below. See Remark A.15 for the proof.

Theorem 1. *Let N denote the true number of elements indexed. The space consumption is $O(N/\beta)$ blocks. The worst-case I/O cost of a lookup is $O(\log_\gamma \frac{N}{\kappa_0 \beta})$ and the amortized I/O cost of an insertion or deletion is $O(\frac{\gamma}{\beta - \gamma} \log_\gamma \frac{N}{\kappa_0 \beta})$.*

As mentioned earlier in this section, FD+tree differs from FD-tree [13] in significant ways. Out of FD+tree's six invariants presented in Section 2.1, FD-tree maintains only the first three: (I1), (I2) and (I3). More importantly, FD+tree improves practicality by addressing the following three issues.

First, although FD-tree supports deletion, it does not tighten levels, and its merges are triggered by overflows only. Therefore, FD-tree provides no performance guarantee for workloads involving deletions; the complexity bounds and proofs in [13] assume insertion-only workloads. Remark A.4 gives examples where the lack of level tightening or underflow-triggered merges can lead to sustained poor performance. In contrast, our Theorem 1 applies to any workload, and its bounds are stated in terms of the *true* number of elements indexed, not in terms of the number of entries stored in the tree (which would have been a weaker guarantee).

Second, FD-tree does not have level skipping; all levels are materialized and fences always point to the immediate next level. Therefore, unlike FD+tree, FD-tree is susceptible to performance degradation by chains of single-block levels, as described in the discussion of merge-finalize.

Third, as discussed at the beginning of Section 2, compared with FD+tree's one-pass multi-level merge, an FD-tree merge proceeds in multiple passes, combining only two levels at a time. When L_0 overflows, FD-tree first merges L_0 and L_1, producing a new L_1 (and a new fence-only L_0). In general, if the result of merging L_{i-1} and L_i can be accommodated by L_i, FD-tree stops the merge; otherwise, FD-tree proceeds to merge L_i and L_{i+1}, rewriting $L_0, \ldots, L_i$ with new fences in the process. Although FD-tree and FD+tree merges cost asymptotically the same, FD+tree's one-pass multi-level merge in most cases is the clear winner both in terms of actual cost and in terms of number of writes (which is important to SSDs because of write wearing). Also, concurrency control for the FD-tree merge is more difficult because multi-pass two-level merges have more complex read/write patterns; such a merge may rewrite a level multiple times, while a one-pass multiple-level merge writes each level once.

3 Towards Concurrency

Section 1 has motivated the need for concurrency control in tree indexes for SSDs. Before presenting our full solution in Section 4, we first present two other approaches to concurrency control in FD+tree, which further motivate our design choices.

FD+XM (FD+Tree with Exclusive Merge) FD+XM uses a single readers-writer lock for the entire FD+tree. The tree supports either multiple concurrent lookups, which must acquire shared locks (s-locks), or a single insertion or deletion request, which must require an exclusive lock (x-lock). If a modification triggers a merge, the x-lock is released only after the merge completes.

FD+XM has low CPU overhead because each request makes a single lock call, and merges run uninterrupted with exclusive access to the tree. For workloads consisting of nearly all lookups or

those whose modifications are issued far apart in time from other requests, we expect FD+XM to work well.

However, FD+XM offers no concurrency between lookups and modifications. As a merge x-locks the entire tree, lookups must wait until the merge completes. Since merges are long, such waits severely lengthen lookup response times to the point of impractical.

FD+DS (FD-Tree with Concurrency by Doubling Space) The key idea is to trade space for concurrency. During a merge, instead of emptying the old levels as we go, we simply leave them intact while producing the new levels on the side. Meanwhile, lookups can still proceed through the old levels. When the new levels are ready, they replace the old levels in an atomic step, and the space taken by the old levels can then be reclaimed. Thus, FD+DS improves lookup response times because readers of the old levels are not competing with any writer; merges run faster too, because the single writer of the new levels is not competing with any reader.

While FD+DS is conceptually simple, implementing it still requires some care, as we have discovered from our experience. First, some concurrency control is still needed. For example, before reclaiming the space taken by the old levels, we must ensure any ongoing lookups through them have completed; we implement this check using a counting semaphore. Second, to support concurrent modifications while a merge is in progress, we need to write these modifications somewhere, and have lookups search through them in addition to the old levels, again necessitating concurrency control. Our implementation of FD+DS adds these modifications to the new top level being created by merge; when searching the new top level, lookups do not follow fences (while lookups through the old levels still do).[8]

One main drawback of FD+DS is that it doubles the index space while merges are ongoing. This higher space requirement is especially costly for SSDs, which are still much smaller and more expensive than magnetic disks.

There is another subtle yet significant disadvantage to FD+DS's simple rule of not modifying the old levels. Recall that the top level occupies premium memory space (the argument also holds if the top level resides in the locality area of SSDs, because this area is small). FD+DS cannot free space in the old top level until after a merge completes, and therefore cannot use that space to accommodate modifications that arrive during the merge. Thus, given limited space, FD+DS can accommodate fewer new modifications during a merge before stalling for its completion, so its worst-case modification response time suffers, as we will see in Section 5. Getting around this issue would require non-trivial patches to FD+DS; we discuss a number of them in detail in Remark A.17. However, these patches either introduce other performance issues or complicate FD+DS to the point where it becomes no simpler than our proposed FD+FC scheme (to be described in Section 4) and yet still uses more space.

Discussion Our main quest is to achieve the same or higher level of concurrency offered by FD+DS without its space overhead. Since merges have a regular, sequential access pattern, it should be possible, with careful updates to the index, to direct lookups to the unprocessed parts of the old levels as appropriate, while space from the processed parts continues to be reclaimed. The idea of trading space for concurrency is still applied, but we can limit redundant storage to data near the "wavefront" of the ongoing merge.

[8] An alternative is to write these modifications to a dedicated memory buffer, but they require special handling when the current merge completes, which would complicate FD+DS even more.

Moreover, better space utilization makes it possible to improve modification concurrency. As a merge progresses, it consumes entries from the top level. By aggressively reclaiming space taken by these entries, it should be possible to process new modifications at the same rate as the merge consumes the old top level.

Next, we show how to realize these possibilities with FD+FC.

4 FD+FC: FD+Tree with Full Concurrency

FD+FC builds on the idea of weaving both unprocessed and processed parts of the index during a merge into a single coherent structure to support concurrent accesses. While this idea is conceptually simple, its realization is far from trivial. For example, we must consider the overhead introduced by maintaining a single coherent index during merges. Moreover, with fractional cascading, cross-level pointers in FD+tree may form a graph, where one index block may be reached by multiple paths, which makes the index trickier to handle than traditional ones such as B-tree where pointers form a tree. In the following, we will start with the conceptual overview of FD+FC, and gradually introduce implementation details and challenges such as those mentioned above.

4.1 Data Structure and Conceptual Overview

When there is no ongoing merge, FD+FC's data structure is the same as FD+Tree. However, when there is an ongoing merge $\mathfrak{M}$ involving $L_0, \ldots, L_m$, the data structure consists of the following parts (as illustrated in Figure 5):

- *New top level* (L_0^{new}). Initially empty at the beginning of $\mathfrak{M}$, this part holds the new modifications that have arrived since; it also holds fences being produced by $\mathfrak{M}$. When $\mathfrak{M}$ completes, L_0^{new} becomes the new L_0.
- *Old top level* (L_0^{old}). At the beginning of $\mathfrak{M}$, this part contains all of L_0. As $\mathfrak{M}$ progresses, this part is gradually emptied from left to right.
- *New upper levels* ($L_1^{\mathsf{new}}, \ldots, L_m^{\mathsf{new}}$). Initially empty at the beginning of $\mathfrak{M}$, they are populated by $\mathfrak{M}$ as it progresses. When $\mathfrak{M}$ completes, L_m^{new} will store all (consolidated) data entries from $L_0, \ldots, L_m$, while the other levels will store only fences.
- *Old upper levels* ($L_1^{\mathsf{old}}, \ldots, L_m^{\mathsf{old}}$). At the beginning of $\mathfrak{M}$, they are $L_1, \ldots, L_m$. As $\mathfrak{M}$ progresses, they are gradually emptied from left to right.
- *Below-merge levels* ($L_{m+1}, L_{m+2}, \ldots$, if $\mathfrak{M}$ is not a full merge). They do not participate in $\mathfrak{M}$ and will not change. As $\mathfrak{M}$ progress, it will gradually move fences for $L_{\mathsf{succ}(m)}$ from L_m^{old} to L_m^{new}.

During $\mathfrak{M}$, the first entry in L_0^{old} is always a fence which we call the *wavefront fence*. This fence serves the special purpose of delineating the old and new parts of the tree. It always points to the current head block of $L_{\mathsf{succ}(0)}^{\mathsf{old}}$. Its key indicates how far the merge has progressed, with the following invariant:

- **(I7)** *Wavefront*: Consider all data entries and fences for $L_{\mathsf{succ}(m)}$ in $L_0, \ldots, L_m$ being merged by $\mathfrak{M}$. Those with keys no greater than the wavefront key have been processed by $\mathfrak{M}$ and can be found by searching from L_0^{new}. Those with keys strictly greater than the wavefront key can be found by searching from L_0^{old}.

Conceptually, the wavefront fence partitions the top and upper levels of the FD+tree into old and new parts. $\mathfrak{M}$ progressively pushes the wavefront forward—emptying the old top and upper levels, consolidating the entries, and populating the new top and upper levels. Meanwhile, modifications go directly to the new top level; lookups check the wavefront fence to determine which parts of the tree need to be searched. $\mathfrak{M}$ empties the old upper levels in a

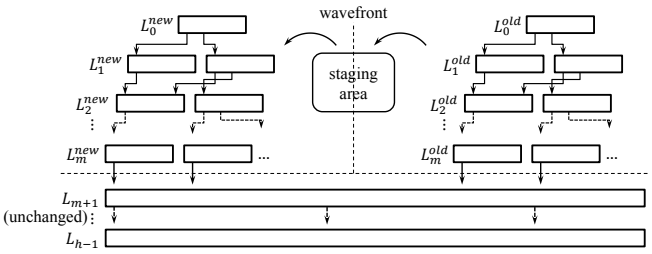

Figure 5: Components of FD+FC during a merge of $L_0, \ldots, L_m$.

careful way such that an old block is reclaimed as soon as no new lookups can ever go through it.

4.2 From Concept to Implementation

Data Movement from Old to New Levels A straightforward implementation of merge moves one "tuple" (or more precisely, all entries with the same key) at a time from $L_0^{\text{old}}, \ldots, L_m^{\text{old}}$ to L_m^{new}. When removing an entry from an old level, we cannot afford to remove it on the SSD, because that would cause an expensive in-place write per entry. Caching the block in memory for searches and updates solves this problem, but there are other performance issues with this *tuple-wise data movement*. Every removal from a block in an old level requires not only updating the block's in-memory data structure, but also x-locking appropriate parts of the tree to avoid conflicts with concurrent lookups that might be reading the old levels. As our performance evaluation reveals, the CPU overhead of these operations are high. Thus, we choose instead to implement *block-wise data movement*. We would never modify a block in an old level, either on SSD or in memory; we only reclaim a complete block when its contents are not needed. The memory requirement is only $O(h\beta)$. More details are given in Section 4.4.2.

Preemptive Level Skipping As discussed in Section 2.3, FD+tree performs level skipping in merge-finalize. Unfortunately, the timing is too late for FD+FC, because concurrent lookups that arrive during $\mathfrak{M}$ would miss this optimization and still see suboptimal tree shapes. Therefore, FD+FC performs preemptive level skipping during merge-prepare, so that even the intermediate tree state produced by merge-execute skips unnecessary levels. The details are in Section 4.4.1.

Dynamic Memory Sharing between Top Levels Both L_0^{old} and L_0^{new} are memory-resident. A simple approach would be to allocate a fixed amount of memory to each, but we can do better at memory utilization by allowing them to share memory dynamically as $\mathfrak{M}$ progresses. We implement L_0^{old} and L_0^{new} using a common buffer of the size allotted for L_0. An overflow-triggered merge $\mathfrak{M}$ begins when L_0 is close to (but below) full capacity and becomes L_0^{old}; we always reserve κ_1 slots for L_0^{new} to accommodate fences to be produced by $\mathfrak{M}$. As $\mathfrak{M}$ progresses, space taken by entries removed from L_0^{old} is given to L_0^{new} for new modifications; a modification may have to wait for $\mathfrak{M}$ to make new space. At the end of $\mathfrak{M}$, L_0^{old} becomes empty, and L_0^{new} becomes L_0.

Locks For disk-resident levels, we use a readers-writer lock for each block. Since L_0 (or L_0^{new} and L_0^{old}) is in memory, we use a single mutex to control its access. As the wavefront fence is stored in L_0^{old}, its access is controlled by the same mutex.

4.3 Modification and Lookup

A modification goes to L_0^{new} if there is an ongoing merge; otherwise it goes to L_0. It waits if no space is available in L_0^{new}. Then, it locks L_0^{new} or L_0 and proceeds exactly as in Section 2.2. By design, FD+FC triggers a merge after the modification completes.

If there is no ongoing merge, we process a lookup exactly as in Section 2.2. If a merge is underway, we compare the lookup key against the wavefront key:

- If key is strictly greater, we first search L_0^{new} (but without going below L_0^{new}). We can stop if we find a data entry with key here. If not, we search L_0^{old}, and through it, the old upper levels and below-merge levels.
- Otherwise, we search L_0^{new}, and through it, the new upper levels and below-merge levels.

In either case, lookup uses a standard tree-based locking protocol. It starts by locking L_0^{old} and L_0^{new}, and it always s-locks a block in the next level before unlocking the current.

4.4 Merge

As with the non-concurrent FD+tree, the merge procedure has three steps. However, as motivated in Section 4.2, FD+FC's merge-prepare performs preemptive level skipping.

4.4.1 Merge-Prepare

We first determine m, i.e., the levels $L_0^{\text{old}}, \ldots, L_m^{\text{old}}$ that need replaced, as in Section 2.3. In addition, we make a conservative, best-effort guess of which result levels to skip, so merge-execute can avoid materializing unnecessary levels.

Because the sizes of the would-be result levels are generally unavailable at this point, we estimate them as follows, starting with $\widehat{U}(m)$, which upper-bounds the number of blocks in L_m^{new} (Eq. (1)). Let $\widehat{B}_i$ $(0 \le i \le m)$ denote our estimate (and also an upper bound) for the size of L_i^{new} in blocks. We have the following recurrence:

$$\widehat{B}_m = \widehat{U}(m); \quad \widehat{B}_{i-1} = \lceil \widehat{B}_i / \beta \rceil.$$

For notational convenience, we also write $\widehat{B}_{m+1} = B(L_{\text{succ}(m)})$, whose actual value we readily know. Using calculations similar to those in FD+ree *merge-finalize* in Section 2.3 (Eq. 3), we compute $\widehat{o}_i$ $(0 \le i \le m)$, an "optimized" level number for what would be L_i^{new} had we simply generated all levels $L_0^{\text{new}}, \ldots, L_m^{\text{new}}$:

$$\widehat{o}_i = \arg\min_j \left(\widehat{B}_i \le \kappa_j \wedge \widehat{B}_{i+1} \le \kappa_{i+1} \right).$$

Again, we try to reduce the level number for each new level as much as possible while preserving all invariants. The set $\widehat{O} = \{\widehat{o}_i \mid 0 \le i \le m\}$ is a subset of the level numbers in $[0, m]$. It can be shown that $0 \in \widehat{O}$ and $m \in \widehat{O}$. Instead of generating all $m+1$ levels, the next step of the merge procedure, *merge-execute*, would generate only $|\widehat{O}|$ levels (always including L_0^{new} and L_m^{new} in particular) bearing the level numbers in $\widehat{O}$.

4.4.2 Merge-Execute

Starting Merge-Execute At this point, the wavefront fence (the first fence in L_0^{old}) has key $-\infty$ and points to the first block of $L_{succ(0)}^{\text{old}}$. The new upper levels do not have any blocks yet; *m-insert*, described below, will create them on demand.

Merge-Execute Main Loop We repeat the following three steps until all entries from the old levels are processed.

- *M-stage* reads entries from the old levels and puts them in key order into an in-memory *staging area* $\mathcal{S}$.
- *M-insert* moves entries from $\mathcal{S}$ to L_m^{new} and then adds fences to levels above as needed.
- *M-delete* updates the wavefront fence and conceptually "deletes" from the old levels the entries in $\mathcal{S}$, which have been added to the new levels by m-insert. Entries in L_0^{old} are really deleted, but for disk-resident levels, m-delete never updates in-place; it simply reclaims whole blocks.

M-Stage M-stage buffers in memory the contents of every block it reads from the old levels, so no block will be read more than once. The memory required for buffering is at most $h\beta$ entries. The first time it runs, m-stage reads one block from each of $L_0^{\mathsf{old}}, \ldots, L_m^{\mathsf{old}}$. In each iteration, m-stage starts with a empty staging area $\mathcal{S}$, and keeps adding buffered entries in key order to $\mathcal{S}$ until some old level runs out of buffered entries to add. M-stage also ensures that $\mathcal{S}$ contains either all entries with the same key, or none at all. When it stops, m-stage hands off $\mathcal{S}$ to m-insert, and reads in the next block for any level that is out of buffered entries. While the actual number of entries in $\mathcal{S}$ varies, it is easy to see that the maximum is $h\beta$.

M-stage acquires no locks, because when it is running there are no other writes.

M-Insert M-insert processes entries in $\mathcal{S}$ one group at a time, where each group contains all entries with the same key. Processing of each group proceeds exactly as in FD+tree's merge-execute (Section 2.3). M-insert buffers new blocks in memory until they are full or it finishes writing; the memory required is $h\beta$ entries.

Note that for each group, m-insert writes new levels bottom-up. Writing to a new block b requires no locking, because at this point no lookup can access b—there are no fences to b yet since we write levels bottom-up. On the other hand, to write to an existing block b, m-insert x-locks b, and unlocks b when it finishes writing and *before* it writes a block in the level above. The timing of release is important to avoid deadlocks with lookups, who might be traversing down the new upper levels with the tree-based locking protocol.

M-Delete Let $\mathcal{S}_0$ denote the subset of entries in $\mathcal{S}$ from L_0^{old}. First, m-delete updates the wavefront fence in L_0^{old}: the key is set to the last key in $\mathcal{S}$ (not $\mathcal{S}_0$), and the pointer is set to that of the last fence in $\mathcal{S}_0$ (not $\mathcal{S}$), if any. Then, m-delete deletes all entries in $\mathcal{S}_0$ from L_0^{old}.

Next, m-delete reclaims blocks in old upper levels that are no longer needed. Knowing when which blocks are safe to reclaim is tricky. It turns out that we cannot simply reclaim a block once all its entries have been processed by m-insert; this block may still be needed to direct lookups. Therefore, m-delete uses the following rule: a block can be reclaimed only when m-insert has processed the first key on the following block on its level. Remark A.5 in appendix explains the intricacies and why this rule works correctly. M-delete applies the rule to the old upper levels top-down. For the head block b of each level, if $\mathcal{S}$ contains the first key in b's following block, b is reclaimed.

M-delete locks L_0^{old} while modifying it. To reclaim a block, m-delete x-locks it and unlocks it when done; there is no need to x-lock the next block below before unlocking the current.

Ending Merge-Execute After the merge-execute main loop completes, we still have a chain consisting of the last blocks from the old levels. Recall m-delete's rule of not reclaiming a block unless the first key on the following block has been processed; the last blocks do not have following blocks. Thus, we reclaim the chain explicitly. Starting from L_0^{old}, we finally delete the wavefront fence and make L_0^{new} the new L_0; after this point, the remaining old upper levels are no longer accessible by lookups. Then, we proceed top-down to reclaim the blocks in old upper levels.

Modifying L_0^{old} requires locking. Then, we follow the standard tree-based locking protocol to reclaim the blocks, always x-locking a block in the next level before unlocking the current one. Tree-based locking is necessary to avoid conflict with any ongoing lookup that might still be searching the old levels top-down for a (nonexistent) key greater than all existing keys; such a lookup traverses on the very chain to be reclaimed.

4.4.3 Merge-Finalize

The merge-prepare step may have overestimated the sizes of the new levels, because it did not account for cancellations between insert and delete entries. Thus, merge-finalize further tightens the levels. Recall that merge-execute produces $|\widehat{O}|$ materialized new levels, as determined by merge-prepare. For each such level L_i, we compute o_i, a new level number for L_i, in a manner similar to FD+tree merge-finalize in Section 2.3 (Eq. (3)):

$$o_i = \arg\min_j \begin{pmatrix} \big(j > 0 \wedge B(L_i) \le \kappa_j \wedge B(L_{\mathsf{succ}(i)}) \le \kappa_{j+1}\big) \\ \vee\ (j = 0 \wedge |L_i| \le \kappa_1) \end{pmatrix}.$$

The first term of the disjunction above checks whether we can reassign L_i to some level L_j below L_0; it is identical to the condition in Eq. (3). The second term of the disjunction, which checks whether L_i can become L_0,[9] requires different treatment. Here, $|L_i|$ denotes the number of entries in L_i. Although L_0 can accommodate $\kappa_0\beta > \kappa_1$ entries, keep in mind that concurrent insertions can claim all unreserved slots in L_0 during merge-execute. Since merge-execute has reserved only κ_1 slots in L_0, κ_1 limits the number of entries a level can have if we want to make it L_0.

After calculating o_i's, merge-finalize proceeds in the same way as FD+-tree merge-finalize described in Section 2.3. The only exception is that L_z, where $z = \arg\max_i(o_i = 0)$, does not completely replace L_0 because L_0 contains entries added while the merge was running; instead, we remove all fences from L_0 and add L_z's content to L_0.

To ensure the correctness of concurrent lookups, merge-finalize x-locks L_0, removes fences in L_0, adds L_z's fences to it, and unlocks L_0. Then, merge-finalize deletes levels $L_1, \ldots, L_z$ in a top-down fashion, x-locking each block before deleting it. This top-down locking and deletion order ensures that no ongoing lookup encounters any deleted block.

4.5 Discussion

Instead of employing standard locking protocols on FD+tree in a straightforward manner, FD+FC carefully considers FD+tree's special access patterns in designing correct and efficient protocols. For example, for the new upper levels, the top-down read pattern of lookup coexists with the bottom-up write pattern of m-insert, which means the standard top-down tree-based locking cannot be applied to both. As another example, for the old upper levels, since both lookup and m-delete have top-down access patterns, the standard tree-based locking would work, but FD+FC instead allows m-delete to deviate by releasing its locks early (before acquiring child locks). This optimization hinges on the observation that there is a single writer of the tree levels at any time—the merge procedure. Without this observation specific to FD+tree, early lock release would lead to deadlocks between two writers.

In conclusion, FD+FC serializes lookups and modification by the order in which they lock the top level, and is free of deadlocks. A discussion of the correctness of FD+FC can be found in Remark A.16.

In terms of space and I/O complexities, bounds established for FD+tree in Theorem 1 still hold for FD+FC. Because of block-wise data movement, a merge may use $O(h)$ (logarithmic in N/β) additional blocks (without affecting the asymptotic space complexity). In comparison, the space-doubling FD+DS (Section 3) uses up to $\Theta(N/\beta)$ additional blocks during a merge.

[9] This term does not need the condition $B(L_{\mathsf{succ}(i)}) \le \kappa_{j+1}$ because it is implied by $|L_i| \le \kappa_1$.

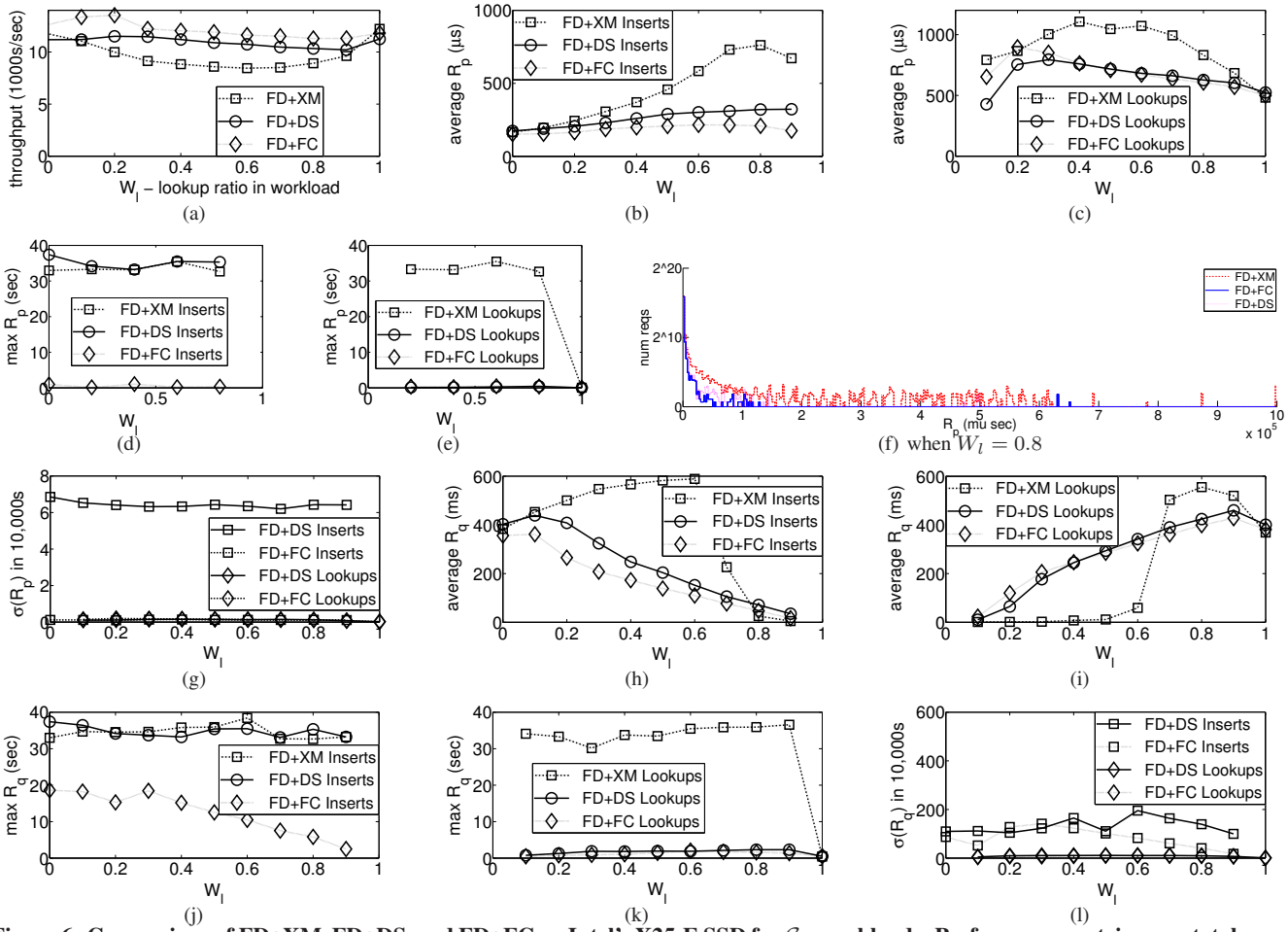

Figure 6: Comparison of FD+XM, FD+DS, and FD+FC on Intel's X25-E SSD for $\mathcal{G}_R$ workloads. Performance metrics are: total completion time (a), average insertion R_p (b), average lookup R_p (c), worst-case insertion R_p (d), worst-case lookup R_p (e), distribution of lookup R_p's (f), standard deviation in R_p (g), comparison on R_q (h)-(l).

5 Experimental Evaluation

We implemented FD+XM, FD+DS, and FD+FC in C++. We use two SSDs in our evaluation: Intel X25-E SLC 32GB SSD and Intel 320 Series MLC 80GB SSD, hereon referred to simply as *X25-E* and *320S*. At the time when we ran our experiments, 12.5TB had been written to X25-E, and 1.5TB to 320S. Here we report only results for X25-E because 320S showed similar trends; for details see Remark A.11. The indexes are stored on the SSD connected through SATA to a workstation with an Intel i7 8-core 2.8GHz CPU, 8GB main memory, and Linux 2.6.32 kernel running in single-user mode without GUI. We used Linux's `ext2` file-system, which does not have journaling. We set the file-system cache and the SSD's internal cache to write-through mode, as recommended by most database vendors. We also experimented with write-back mode for SSD's internal cache; see Remark A.12 for details.

We implemented two workload generators for the evaluation. The first generator, $\mathcal{G}_R$, generates a stream of lookup (l), insertion (i), and deletion (d) requests with a specified $W_l : W_i : W_d$ proportion. The second generator, $\mathcal{G}_T$, generates the stream of requests by following the TPC-C workload characteristics. Remarks A.7 and A.8 give more details about the two generators. The workload is stored as a file from which a *workload injection thread* reads and populates two request queues: Q_r for index lookup requests (reads), and Q_w for index insertion and deletion requests (writes). We allocate T_r and T_w *worker threads* to process requests from Q_r and Q_w respectively. If the workload injection thread finds Q_r or Q_w full, it blocks until slots become available in that request queue.

We measure performance with the following metrics:

- R_p is the time taken by a worker thread to dequeue a request and process it to completion (including computation, I/O, and lock wait times).
- R_q is R_p plus the time spent by the request waiting in the queue.
- R_o is the overall time between request arrival and completion. See Remark A.9 for details on how we obtain R_o.

As emphasized in Section 1, we measure (a) average, (b) variance, and (c) worst-case for the above metrics over each entire workload. We will focus on the R_p and R_q metrics in this section. Results for R_o are similar and are sampled in the appendix.

When testing with $\mathcal{G}_R$, we preload each index with 10M insertions resulting in an index of 120MB; we then run a workload containing 10M requests (with specified $W_l : W_i : W_d$) and measure performance after a warm-up of 10K requests. When testing with $\mathcal{G}_T$, we use 10 districts per warehouse and 3000 customers per district. The preload step inserts 3 orders per customer.

Defaults for the size of the FD+tree top level (κ_0, but measured in bytes), size ratio (γ), and the main-memory buffer cache size are 256KB, 24, and 15MB respectively. Defaults for the total number of queue slots ($|Q_r| + |Q_w|$) and number of worker threads ($T_r + T_w$) are 5000 and 8 respectively. The queue slots are divided between Q_r and Q_w according to the ratio $W_l : (W_i + W_d)$. For

FD+FC and FD+XM, we set $T_r = 6$ and $T_w = 2$ because these indexes process insertions quickly. FD+FC runs merges in an extra background thread that is woken up by one of the T_w threads whenever merge is triggered (by overflow or underflow).

5.1 Overall Benefits of Full Concurrency

Varying the Lookup Ratio Figure 6 compares FD+FC against FD+XM and FD+DS. We consider a spectrum of $\mathcal{G}_R$ workloads by varying the ratio of index lookup requests W_l in the workload from 0 to 1. The non-lookup requests in each workload are distributed equally between insertions and deletions so as to keep the total number of indexed records roughly constant throughout workload execution. For example, a workload with $W_l = 0.6$ will have 20% each of insertions and deletions. Recall that our default total workload size is 10M requests.

At a high-level, Figure 6 shows that 1) FD+FC significantly outperforms FD+XM on worst-case response times, as we expect by design, but as a bonus, FD+FC also performs better on other metrics; 2) FD+FC delivers comparable or better performance than FD+DS without requiring the double amount of space as FD+DS; 3) Despite of doubling space, FD+DS's worst-case insertion times are in fact as bad as FD+XM. We now delve into details below.

Figure 6(a) shows that FD+FC's throughput is at least as good as FD+DS and is much better than FD+XM. For a very update intensive workload ($W_l \leq 0.2$), FD+FC's throughput is larger than FD+DS by around 15%. For other workloads, it is larger by 8–9%. This is because updates have to wait longer in FD+DS. Over FD+XM, FD+FC's advantage on throughput metric is more pronounced. It is around 27% and 20% larger when $W_l = 0.6$ and 0.8, respectively.

Figures 6(b) and 6(c) show the average R_p for insertions and lookups (deletions are handled just as insertions by FD+trees). When W_l reaches 0.9, FD+FC processes insertions faster than FD+DS by 25% on average. It is faster by 3.8 times compared to FD+XM. Insertions take little time to process by themselves, but for FD+XM and FD+DS, they wait longer when there is an ongoing merge—FD+XM waits for the release of the exclusive lock, while FD+DS waits for the reclamation of the old top level. When we consider average lookup R_p, all three approaches perform equally well when tested with a read-only workload. Also, FD+FC performs equally as well as FD+DS except for update-heavy workloads; lock/unlock calls of frequently triggered merges slow down the lookups. For FD+XM, it is much worse, because lookups need to wait when a merge is ongoing.

Figures 6(d) and 6(e) show the worst observed R_p for insertions and lookups. While FD+FC's worst R_p over insertions is around a second, for FD+DS and FD+XM it is around 33 to 38 seconds—displaying a crucial advantage of the fully concurrent FD+FC. As described in Section 3, FD+DS cannot remove entries from L_0 even after they were added to the new levels. If it does, lookups will fail. It can delete all the entries only after the merge completes. When the merge involves many levels (for ex., a full merge), this scheme is obviously costly. As for lookups, FD+FC and FD+DS show similar worst R_p's (around 500 milliseconds), which are well below FD+XM, because lookups in both FD+FC and FD+DS do not need to wait for an ongoing merge to complete.

Figure 6(f) shows equi-width histograms with a bucket length of 2.5 milliseconds for lookup R_p's of FD+DS and FD+XM. Similar trends were observed for insertions as well. The Y-axis is in log-scale. The X-axis shows the first 400 buckets, i.e., $R_p \leq 1$ second; requests with $R_p > 1$ second are added to the last bucket in Figure 6(f)—hence the (red) blip at the end of FD+XM's histogram. Figure 6(g) complements Figure 6(f) by showing how the standard deviation of R_p values for insertions is much lower for FD+FC than that for FD+DS.

Figures 6(h)-6(l) compare FD+FC and FD+DS on the R_q performance metric (recall that it is R_p plus the time spent by the request in the Q_r or Q_w queue). FD+FC's better performance on the core R_p metric translates into better performance on R_q (and R_o as well—see Remark A.10 in appendix). For the three concurrency schemes, average insertion R_q is large when W_l is small. Average lookup R_q shows an opposite trend. The reason is that, for low W_l, there are so many updates that processing them becomes the bottleneck. Hence, insertion requests wait in the queue longer than lookup requests. As W_l increases, the bottleneck shifts to lookup processing. Figure 6(k) shows FD+DS and FD+DXM have high worst-case insertion R_q. FD+FC's worst-case insertion R_q starts higher, because the workload is too skewed, and there are always many insertions waiting in the queue; but as W_l increases, it falls to 2.5 seconds, where as FD+DS and FD+XM still remain close to 33 seconds. Figure 6(k) shows FD+XM suffers high worst-case lookup R_q. Figure 6(l) shows higher standard deviation for FD+DS insertion R_q distribution than that of FD+FC.

Varying the Initial Index Size Figure 7 shows the performance trends on X25-E as we scale the initial index size to 90M key-value pairs totaling 1080MB. Size of the workload is set to be equal to the initial index size. There are 80% lookups in every workload. Note that the worst-case insertion R_p for FD+DS jumps to 659 seconds when the database size is 90M. However, FD+FC's worst insertion R_p still remains under a second. The benefits of FD+FC's full concurrency are clear when data sizes increase; its lookup and insertion R_p's remain manageable. These results show that FD+FC's concurrency algorithms scale as well as FD+DS.

Workloads based on TPC-C Figure 8 shows the comparison of the three indexes for $\mathcal{G}_T$ workloads with TPC-C characteristics.

As the number of warehouses increases from 20 to 100, initial index size increases from 21MB to 105MB. Note that the insertions constitute 91.3% of the requests in this workload (see Remark A.8); the remaining 8.7% are lookups. For such workload characteristics, Figure 8(a) shows that FD+FC's throughput is higher than both FD+XM as well as FD+DS. It is higher by 14% than FD+DS when number of warehouses is 100. Average insertion R_p is slightly better for FD+FC (Figure 8(b)), around 13% less when the number of warehouses is 100. However, FD+DS has better average lookup R_p because for high-insertion workloads, FD+FC's concurrent block reclamations constantly interfere with lookups. Nonetheless, FD+FC still has higher throughput overall, and as Figure 8(c) shows, has lower worst-case insertion R_p. In fact, there is also a high variance in the distribution of insertion R_p values (see Figure 8(f)). FD+FC's average R_q for update requests is as worse as FD+DS (Figure 8(d)), this is because the workload is very update intensive. In the worst-case R_q comparision, FD+DS has a high worst-case response time limitation for insertions (Figure 8(e)).

5.2 Benefits of Design Choices in FD+FC

Having seen the end-to-end benefits of FD+FC, we now drill down to the benefits provided by individual features.

Dynamic Memory Sharing between L_0^{new} and L_0^{old} FD+FC's memory sharing feature allows space freed from L_0^{old} by a merge to be added to L_0^{new} immediately (Section 4.4). The plot 'FD+FC w/o MS' in Figure 9(a) compares the performance of FD+FC without memory sharing against 'FD+FC w/ MS,' the regular version of FD+FC with memory sharing. We include FD+DS performance data for better illustration of memory sharing feature's advantage. FD+DS's worst insertion R_p is close to that of 'FD+FC w/o MS,'

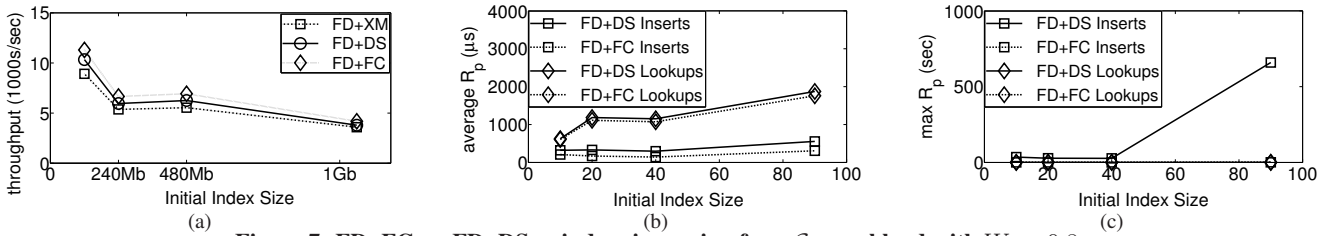

Figure 7: FD+FC vs. FD+DS as index size varies, for a $\mathcal{G}_R$ workload with $W_l = 0.8$.

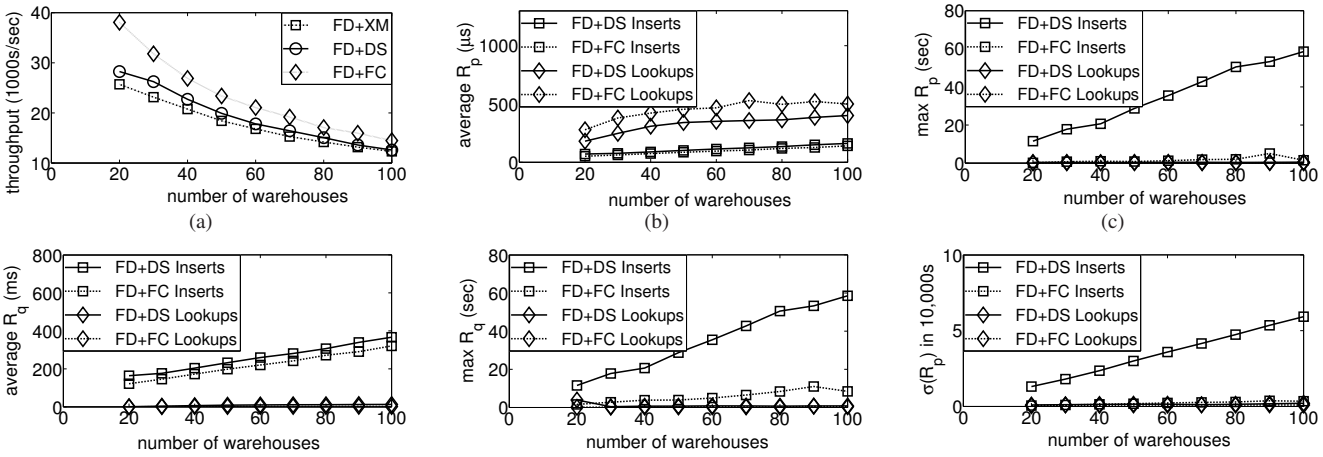

Figure 8: Comparison of FD+XM, FD+DS, and FD+FC for $\mathcal{G}_T$ workloads on X25-E by varying number of warehouses, in terms of: total completion time (a), average insertion and lookup R_p (b), worst-case insertion and lookup R_p (c), average insertion and lookup R_q (d), worst-case insertion and lookup R_q (e), and standard deviation in R_p (f).

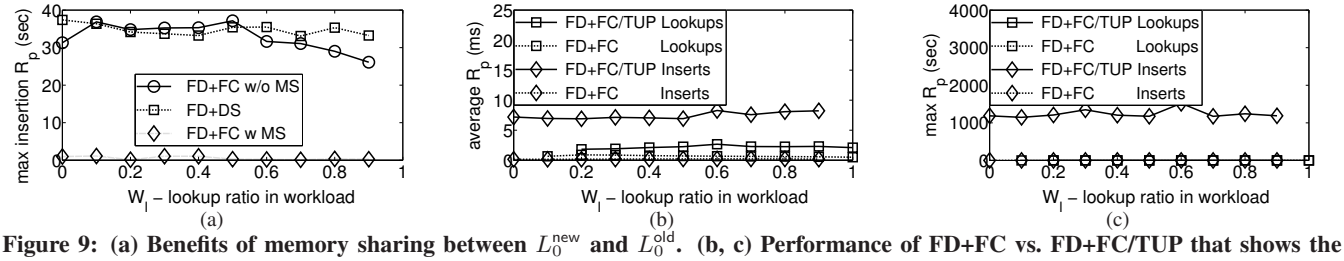

Figure 9: (a) Benefits of memory sharing between L_0^{new} and L_0^{old}. (b, c) Performance of FD+FC vs. FD+FC/TUP that shows the importance of FD+FC's block-wise data movement.

which is around 32–38 seconds. Worst insertion R_p for FD+FC is much smaller at around a second.

Block-wise vs. Tuple-wise Data Movement FD+FC uses block-wise data movement during merges (Section 4.4). To show the benefit of this feature, Figures 9(b) and 9(c) compare FD+FC against FD+FC/TUP, a variant of FD+FC that uses tuple-wise data movement in the merge procedure. FD+FC/TUP leads to heavy CPU usage and long-running merges that impact insertion response times severely. When $W_l = 0.9$, average insertion R_p of FD+FC/TUP is 12 times worse than FD+FC. FD+FC/TUP's worst-case insertion R_p is much worse than even that of FD+XM, highlighting the importance of optimizing CPU usage when using SSDs.

Level Skipping and Tightening FD+FC contains the novel level skipping and tightening features that remove unnecessary levels from the index (Section 4.4). To see the benefits of level skipping, we created an FD+tree with $\gamma = 12$ and $\kappa_0 = 64$KB. We inserted 10M key-value pairs, and ran a $\mathcal{G}_R$ workload of 20M requests with $W_l = 0.8$. In the execution trace, we looked for cases where a merge produced an index tree with one less materialized level than before. As expected, all these cases lowered lookup response times. One specific case, where a merge on an FD+CC with five materialized levels led to an FD+CC with four materialized levels and one skipped level, lowered average lookup R_p by 14%. This improvement is essentially "free" as the level skipping and tightening steps do not increase the running time of merges.

5.3 FD+FC vs. an Industry-Strength B+Tree

When developing our FD+FC prototype, we were mainly concerned with ensuring a fair comparison of FD+FC against FD+DS and FD+XM. Nonetheless, here we report on a comparison between our FD+FC prototype and Berkeley DB's industry-strength and fully concurrent B+tree implementation. For the B+tree, we changed $T_r : T_w$ to match $W_l : (W_i + W_d)$, which suited it better than our settings intended for FD+tree (see the beginning of this section).

One would expect the B+tree to perform better on read-only workloads, while FD+FC is expected to perform better at the other end of the spectrum where $W_l = 0$. That is the behavior we see in Figure 10(a), which shows workload completion times for our default experimental settings. More results are in Remark A.13.

The cross-over point between B+tree and our prototype FD+FC occurs between $W_l = 0.8$ and $W_l = 0.9$ in Figure 10(a). This point can be pushed more to the right (i.e., larger W_l) by further optimizing the implementation of our FD+FC prototype (e.g., minimizing CPU-intensive `memcpy()` calls). The current prototype

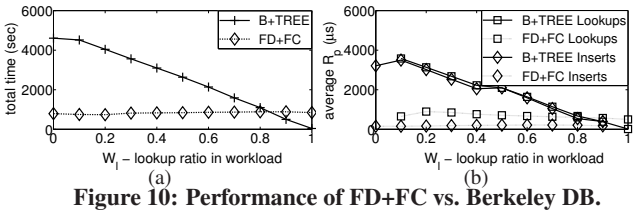

Figure 10: Performance of FD+FC vs. Berkeley DB.

makes heavy use of the C++ Standard Template Library (STL) classes (for internal data structures) and the Boost threading library, which can be avoided to improve overall performance. Even without deep engineering-level optimizations, FD+FC performs better than Berkeley DB in minimizing the average-case R_p (see Figure 10(b)).

6 Related Work

A number of indexes have been proposed recently to optimize for the SSDs' fast random read and slow random write characteristics. *BFTL* [22], *FlashDB* [17], and *LA-tree* [2] are based on B-trees and perform some form of logging in order to postpone in-place updating of B-tree blocks. *SkimpyStash* [7] and *SILT* [14] are exact-match key-value stores based on hashing. FD-tree [13] is a state-of-the-art index designed for SSDs, which we have discussed and compared with in detail in Section 2.4. All of these indexes have reorganizations as essential part of their operations. Their reorganization costs vary, but are consistently higher than those in their counterparts designed for magnetic disks. However, none of these past works address concurrency control. The *PIO B-tree* [20] technique includes a basic concurrency scheme that is very similar to FD+XM (discussed in Section 3). In contrast, the FD+FC scheme we developed allows lookups concurrent access to the index while a merge is ongoing.

Index structures optimized for writes to magnetic disks have also been considered in the database literature. *LSM-tree* [19] maintains multiple B-trees with geometrically increasing sizes. All updates go to the smallest tree. *Rolling merges*, which run concurrently between each pair of neighboring levels, percolate these updates to lower levels. Our work differs from LSM-tree in many ways. First, LSM-tree's design is motivated by an always active insertion workload (e.g., when indexing a growing log file), while we target traditional workloads including OLTP. Second, CPU efficiency is not a concern for LSM-tree; however, since SSDs have orders-of-magnitude faster I/Os than magnetic disks, CPU costs become significant (see Section 5) and we must design for CPU efficiency. Together, these differences in design goals translate into very different choices: 1) To speed up search across levels, FD+tree uses fractional cascading (Section 2), which requires maintaining pointers across levels. LSM-tree does not use fractional cascading because it targets insertion-heavy workloads. 2) LSM-tree uses multiple rolling merges to increase insertion throughput. However, such an approach would consume a lot of OS resources (threads, memory, etc.) and add too much CPU overhead for OLTP workloads running on SSDs; it would also significantly complicate concurrency control in the presence of fractional cascading. In contrast, FD+FC's one-pass multi-level merge is more CPU-efficient and works well with fractional cascading.

The *LHAM-tree* [16] is conceptually similar to LSM-tree, but targets temporal databases. The *bLSM-tree* [21] is similar to LSM-tree, but uses bloom filters to improve lookup performance and carefully designed scheduling policies for synchronizing between rolling merges. The *Stepped-Merge* technique proposed in [10] is similar to LSM-tree, but maintains multiple B-trees at each level. The *T1SM* [11] partitions data into subindexes, each of which is an LSM-tree. For the last two techniques, concurrency control is implemented by allowing the merge to create a new version of the index or subindex, and dropping the previous version after the merge completes. Thus, this approach is analogous to FD+DS discussed in Section 3, which we have evaluated and compared with FD+FC in Section 5. Like LSM-tree, none of LHAM-tree, bLSM-tree, Stepped-Merge, and T1SM supports fractional cascading.

7 Conclusion

New indexes are being designed for database systems that store data on SSDs. We argue that efficient concurrency control schemes are crucial in making these indexes usable for a wide spectrum of workloads. In this paper, we have described the FD+tree index for SSDs and the associated FD+FC concurrency control scheme, which, to our knowledge, is the first of its kind. We demonstrated the performance benefits of FD+FC through extensive experimental evaluation. A promising avenue for further work is to consider crash recovery for FD+FC.

References

[1] Facebook Recommends Minimizing Request Variance. `http://tinyurl.com/389aro4`.

[2] D. Agrawal, D. Ganesan, R. K. Sitaraman, Y. Diao, and S. Singh. Lazy-adaptive tree: An optimized index structure for flash devices. *Proceedings of the VLDB Endowment*, 2(1):361–372, 2009.

[3] J. L. Bentley. Decomposable searching problems. *Inf. Process. Lett.*, 8(5):244–251, 1979.

[4] P. A. Bernstein, C. W. Reid, and S. Das. Hyder - a transactional record manager for shared flash. In *Proceedings of the 2011 CIDR*, pages 9–20, Asilomar, California, USA, January 2011.

[5] L. Bouganim, B. T. Jónsson, and P. Bonnet. uflip: Understanding flash io patterns. In *CIDR*, 2009.

[6] B. Chazelle and L. J. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

[7] B. K. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD*, pages 25–36, Athens, Greece, June 2011.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 2007 ACM SOSP*, pages 205–220, Stevenson, Washington, USA, October 2005.

[9] J. Gray and B. Fitzgerald. Flash disk opportunity for server applications. *ACM Queue*, 6(4):18–23, 2008.

[10] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and Rama Kanneganti. Incremental organization for data recording and warehousing. In *Proceedings of the 1997 VLDB*, pages 16–25, Athens, Greece, August 1997.

[11] C. Jermaine, E. Omiecinski, and W. G. Yee. Out from under the trees. In *Proceedings of the 2002 ICDE*, page 265, San Jose, California, USA, February 2002.

[12] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory SSD in enterprise database applications. In *Proceedings of the 2008 ACM SIGMOD*, pages 1075–1086, Vancouver, Canada, June 2008.

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

[14] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the 2011 ACM SOSP*, pages 1–13, Cascais, Portugal, October 2011.

[15] C. V. Millsap. Thinking clearly about performance, part 2. *Commun. ACM*, 53(10):39–45, 2010.

[16] P. Muth, P. E. O'Neil, A. Pick, and G. Weikum. The LHAM log-structured history data access method. *The VLDB Journal*, 8(3–4): 199–221, 2000.

[17] S. Nath and A. Kansal. FlashDB: Dynamic Self-Tuning Database for NAND Flash. In *IPSN*, pages 410–419, 2007.

[18] M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley DB. In *Proceedings of the 1999 USENIX ATC*, pages 183–191, Monterey, California, USA, June 1999.

[19] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.
[20] H. Roh, S. Park, S. Kim, M. Shin, and S.-W. Lee. B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives. *Proceedings of the VLDB Endowment*, 5(4):286–297, 2011.
[21] R. Sears and R. Ramakrishnan. blsm: A general purpose log-structured merge tree. In *Proceedings of the 2012 ACM SIGMOD*, pages 217–228, Scottsdale, Arizona, USA, June 2012.
[22] C.-H. Wu, L.-P. Chang, and T.-W. Kuo. An efficient b-tree layer for flash-memory storage systems. In *RTCSA*, 2003.

APPENDIX

Remark A.1 (Avoiding overestimation of merge result sizes; Section 2.4) To avoid overestimation of merge result sizes, we simply need to track cancellations between insert and delete entries across levels more carefully. Note that N_Δ and N_∇ are insufficient because they only tell us the number of cancellations for a full merge; we need to know this number for a merge involving any number of levels. To this end, we use $O(h^2)$ counters N_{pq}, where $0 \le p < q < h$; N_{pq} tracks the number of delete entries in L_p that would cancel out some insert entries in L_q. Let $E(i)$ denote the number of data entries in L_i, which is readily available. With the help of the counters, the number of data entries in the run obtained by merging $L_0, \ldots, L_i$ can be accurately calculated as:

$$\textstyle\sum_{j=0}^{i} E(j) - \sum_{0 \le p < q \le i} N_{pq}.$$

From the number of data entries, it is straightforward to calculate the number of blocks.

Upon completion of merge-execute involving $L_0, \ldots, L_m$, we update the counters as follows:

$$N_{pq} \leftarrow \begin{cases} \sum_{j \le m} N_{jq} & \text{for } p = m \text{ and } q > p; \\ 0 & \text{for } p < m \text{ and } q > p. \end{cases} \quad (4)$$

Finally, when processing a deletion, we must first locate the level L_i with the corresponding insert entry; then, we increment N_{0i} by 1. Therefore, a lookup is required as part of deletion processing.

In the version of the FD+tree presented in Sections 2 and 4 and evaluated in Section 5, we choose not to use these counters, and deletions do not involve lookups. Code that uses the FD+tree can easily ensure that there are no deletions of nonexistent elements. Furthermore, gross overestimation of merge result sizes is rare and does not affect correctness; level tightening performed by merge-finalize further protects against performance degradation. Thus, we feel that the approach without the N_{pq} counters suffices in practice.

Remark A.2 (Overestimation of merge result size; Section 2.3) Consider the following worst-case example. Before the merge, L_m has $\kappa_{m-1} + \epsilon$ blocks (i.e., slightly more than the maximum size of L_{m-1}), while $L_0, \ldots, L_{m-1}$, all at full capacity, contain only fences and delete entries. During a merge, these delete entries "cancel out" all but a few insert entries in L_m. Thus, the new L_m is nearly empty while $\widehat{U}(m)$ is roughly $2\kappa_{m-1}$. It would be inefficient to have a chain of m fences pointing to a nearly empty L_m.

Remark A.3 (Tightening of the last level produced by merge; Section 2.3) To preserve (I4) and limit the tree height, we need to tighten the bottom level produced by a full merge. However, when we do not have a full merge, we do not adjust the last new level's number for a technical reason having to do with the proof of Lemma 4 in Appendix. In particular, that proof assumes that after an L_m-merge, we need to fill up all empty (fence-only) levels above L_m with data entries in order to trigger another L_m-merge. This observation allows us to lower-bound the number of modification requests between merges.

Consider the case when an L_m-merge produces a run that is just small enough to be accommodated by L_{m-1}. If we perform level tightening here, this run would become a nearly full L_{m-1} (and L_m would be skipped). It would take a much fewer number of modifications requests to trigger another L_m-merge. It remains open whether this possibility actually breaks our asymptotic bounds; we have simply taken a safe approach in this paper by not tightening the last new level for a merge that is not full.

Remark A.4 (Examples where level tightening and underflow-triggered merges are needed; Section 2.4) Consider Remark A.2; suppose the merge there is a full merge. Without level tightening, the resulting tree, despite being nearly empty, can have an arbitrary number of levels.

We show another example highlighting the need for underflow-triggered merges. Suppose a full merge has just completed, leaving the bottom level L_{h-1} holding slightly more data than what L_{h-2} can accommodate. We then issue a series of deletions that would cancel out all but a few insert entries in L_{h-1}. All corresponding delete entries are eventually pushed down to L_{h-2}, right above L_{h-1}. Without underflow-triggered merges, however, there is no merging with L_{h-1}. At this point, the true number of elements is close to zero, but the tree can be arbitrarily tall.

For both examples above, the lookup cost is arbitrarily high compared with the true number of elements indexed. Furthermore, this situation can last for as long as needed, over any number of operations (future modifications can simply populate and empty the top levels indefinitely without affecting the bottom levels).

Remark A.5 (When to reclaim a block; Section 4.4.2) Consider a block b_1 and its following block b_2 on the same old upper level. Let k_1 denote the key of the last entry in b_1 and let k_2 denote the key of the first entry in b_2. Entries in (k_1, k_2) in the next level reside in some block c that precedes the block pointed to by the first fence of b_2. Suppose m-insert has just processed entries with key k where $k_1 \le k < k_2$, so all entries in b_1 have been processed. However, if we reclaim b_1 at this point, a lookup in the key range (k, k_2), which still should be directed to the old upper levels, would not be able to reach c.

On the other hand, if m-insert has just finished processing k_2, we can safely reclaim b_1 because the wavefront key has moved to k_2, and lookups for keys no greater than k_2 will be directed to the new upper levels instead.

Remark A.6 (Addressing recovery for FD+FC)

We describe at a high level how recovery for FD+FC can be addressed. Note that we did not include recovery in our experimental analysis. Recovery schemes proposed for LSM-tree and LHAM-tree, found in [19] and [16], are in many ways similar to what we describe below.

We need to address recovery for the following parts of the index: (P1) L_0^{new}; (P2) L_0^{old}; (P3) levels $L_1^{\mathsf{old}}, \ldots, L_m^{\mathsf{old}}$; and (P4) levels $L_0^{\mathsf{new}}, \ldots, L_m^{\mathsf{new}}$. Levels with level identifiers larger than m are not touched by the merge.

To facilitate recovery, at the beginning and end of every merge, start-merge and end-merge log records are appended to the log. The merge algorithm also writes a log record for every fence it inserts to L_0^{new}.

Addressing (P3) and (P4) can be achieved by a simple change in FD+FC's node deletion scheme. Remember that *M-Delete* waits until the first key in the following node is processed before deleting a node. This wait ensures lookups don't encounter deleted nodes when they search the old levels. To address recovery, we let *M-Delete* wait a little longer: until the first key in the following node is processed *and* all entries in the node are written to stable storage as

part of the new levels. FD+FC's earlier deletion scheme had a space cost of an extra m blocks; the cost in the worst-case scenario when each level's first node was processed but each of its next sibling was not started processing yet. With recovery included, the space cost increases utmost by only 1 more block, if merge implements a block-at-a-time flush to L_m^{new}.

Recovery for (P1) and (P2) can be achieved in the following way. The recovery manager checks for the oldest start-merge record in the log. If there was no end-merge after it, updates to the index recorded in the log before the start-merge but after the previous start-merge are used to construct L_0^{old}. Otherwise, L_0^{old} is initialized with null set. The recovery manager constructs L_0^{new} from log records that appear after the oldest start-merge in the log.

Based on the last node of L_m^{new} and the old levels as recovered, the recovery manager calculates how far the merge has moved, and initializes the wavefront fence in L_0^{old} accordingly. Entries smaller than or equal to the key of the wavefront are deleted from L_0^{old}. The merge is restarted.

Remark A.7 (Workload generator $\mathcal{G}_R$; Section 5) Given the total number N of requests and the proportion $W_l : W_i : W_d$ of lookup, insertion, and deletion requests for a workload, $\mathcal{G}_R$ first determines the number of insertion requests N_i to be generated. It initializes two variables k_l and k_u with the smallest and largest keys that will be inserted as part of this workload: $k_l = s + 1$ and $k_u = s + N_i$, where s is the largest key (inserted previously) in the index before the workload starts (or zero if the index is empty).

$\mathcal{G}_R$ follows a two-step procedure to generate each request. It first determines which type of request to generate by choosing randomly with the ratio $W_l : W_i : W_d$. To generate a lookup request, $\mathcal{G}_R$ picks a key from a list $\mathcal{K}$ that maintains all keys currently in the index. To generate an insertion request, $\mathcal{G}_R$ picks the smallest key in $[k_l, k_u]\backslash\mathcal{K}$ and a randomly chosen value; the chosen key is added to $\mathcal{K}$. To generate a deletion request, $\mathcal{G}_R$ randomly picks from $\mathcal{K}$ with the constraint that it was inserted at least 100 requests earlier.

Remark A.8 (Workload generator $\mathcal{G}_T$; Section 5) TPC-C benchmark simulates a complete order-entry environment. The following five types of transactions execute against the database: order entry, order delivery, payment, order status check, and inventory check. While each order is entered as a single transaction, 10 orders are delivered together according to the TPC-C specification. Hence, order delivery transactions are roughly 10 times fewer than order entry transactions. The frequencies of the five transactions according to the TPC-C specification are 45%, 43%, 4%, 4% and 4%.

The *NewOrder* table in TPC-C has attributes `NO_W_ID` (warehouse id), `NO_D_ID` (district id) and `NO_O_ID` (order id). The three attributes together form the primary key. Orders that have not been processed yet are stored in this table. Our workload generator $\mathcal{G}_T$ generates index requests for the primary index built on the New Order table. The key is a 4-byte unsigned integer and is constructed by placing the warehouse id in the most significant 8 bits, the district id in the next 8 bits, and using the rest of the 16 bits to store the order number. The corresponding value for the key is generated as a random number.

Remark A.9 (Obtaining R_o; Section 5) In a realistic setting, R_o would be the end-to-end response time observed by the user or application generating the workload. In our evaluation, we calculate R_o in a post-processing step after workload completion. The workload generator generates requests with timestamps according to a pattern where requests arrive at a known uniform rate of v requests per second (starting from time 0). These requests are written to a workload file that is read by the workload injection thread during workload execution.

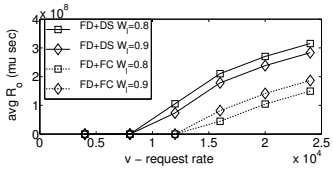

Figure 11: FD+FC vs. FD+DS on R_o (end-to-end response time). X-axis shows the number of request arrivals per second.

The workload injection thread issues requests to the Q_r and Q_w queues at the maximum speed possible, ignoring the arrival timestamps. A trace of the workload execution is captured. Next, wait time w is calculated as the time difference between the supposed arrival time of each request (i.e., the 0-based arrival timestamp recorded in the file plus the actual start time of the workload) and its actual arrival time (time of entry into the queue). If w is positive (which happens when the system processes requests at a rate slower than v), then R_o is calculated as $R_q + w$; otherwise R_o is set to R_q. Note that the R_p and R_q times do not depend on w.

Essentially, such an accounting of R_o means that the underlying execution model can access requests ahead of their supposed arrival time if space is available in the queues. An alternate model is to actually wait until each request's supposed arrival time to issue it, but this approach rules out the possibility of simulating a workload faster than real time, and more importantly, imposes considerable CPU overhead in setting and waiting for timers. We decided against such an alternative in order to minimize the system's extra resource utilization and its influence on measurement results.

Remark A.10 (Additional details on FD+FC vs. FD+DS; Section 5.1) Figure 11 shows how, for a range of request arrival rates (shown on the X-axis), FD+FC outperforms FD+DS on the R_o metric. For $W_l = 0.8$, performance difference between FD+FC and FD+DS is larger. For a lookup-only workload (ie., $W_l = 1.0$), performance of both schemes converge. This is expected because lookups are processed similarly in both schemes. Recall that Figure 6 shows FD+FC and FD+DS performing equally for a lookup only workload.

Remark A.11 (Results for Intel 320S SSD; Section 5)

Figure 13 shows results for $\mathcal{G}_R$ workload on the Intel 320S SSD (referred as *320S*). The trends observed are generally similar to the ones that we saw for the X25-E SSD (see Figure 6). Figure 12(a) shows throughput comparision between FD+XM, FD+DS, and FD+FC. FD+FC's throughput is atleast as high as FD+DS for all workloads, and when $0.2 \leq W_l \leq 0.7$ it is atleast 33% higher than FD+XM. For update intensive workloads, throughput of all schemes are higher relative to their observed throughputs on X25-E because 320S is a newer SSD, and has better write performance. As lookup ratio in the workload increases, performance impact because of writes decreases.

Figure 13(b) shows that the average R_p for update requests exhibits similar trends as in X25-E. But all concurrency schemes run faster than on X25-E because 320S processes writes faster; there is a speed up of around 60% when $W_l = 0$. When $W_l = 0.2$, the average lookup R_p of FD+FC experiences a jump. We believe this is because of intense cache pollution caused by fast running merges. Rest of the Figures 13(d), 13(e) and 13(f) show similar trends as observed in X25-E experiment.

Figure 14 compares the three schemes for $\mathcal{G}_T$ workloads. We observe same trends as seen in Figure 8 for the X25-E experiment. However, absolute numbers for all the schemes have improved.

Remark A.12 (Results with write-back disk caching enabled; Section 5)

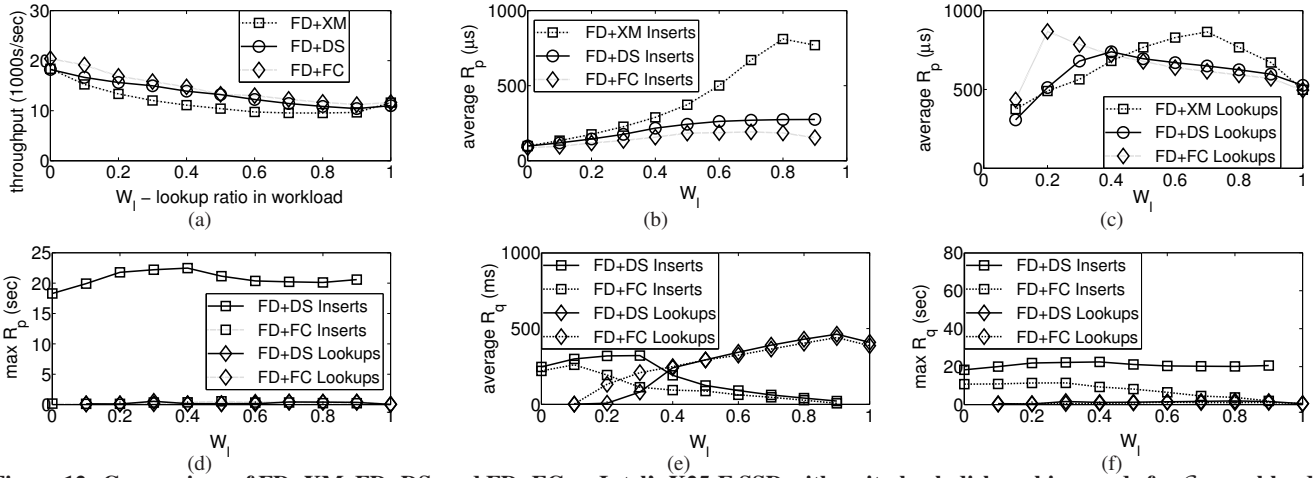

Figure 12: Comparison of FD+XM, FD+DS, and FD+FC on Intel's X25-E SSD with write-back disk caching mode for $\mathcal{G}_R$ workloads. Performance metrics are: total completion time (a), average insertion R_p (b), average lookup R_p (c), worst-case R_p (d), average R_q (e), and worst-case R_q's (f).

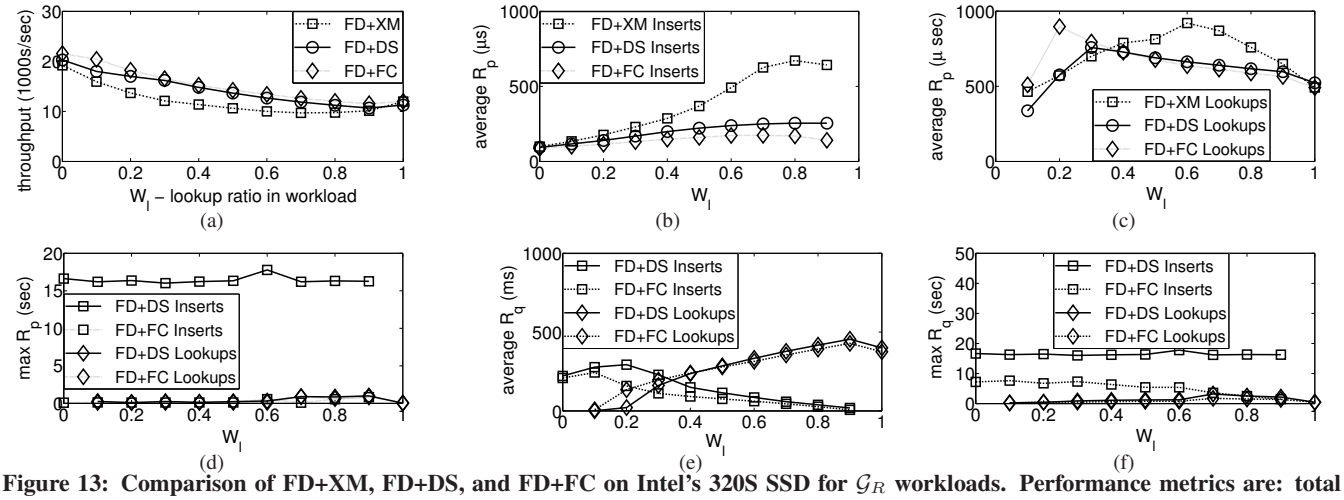

Figure 13: Comparison of FD+XM, FD+DS, and FD+FC on Intel's 320S SSD for $\mathcal{G}_R$ workloads. Performance metrics are: total completion time (a), average insertion R_p (b), average lookup R_p (c), worst-case R_p (d), average R_q (e), and worst-case R_q's (f).

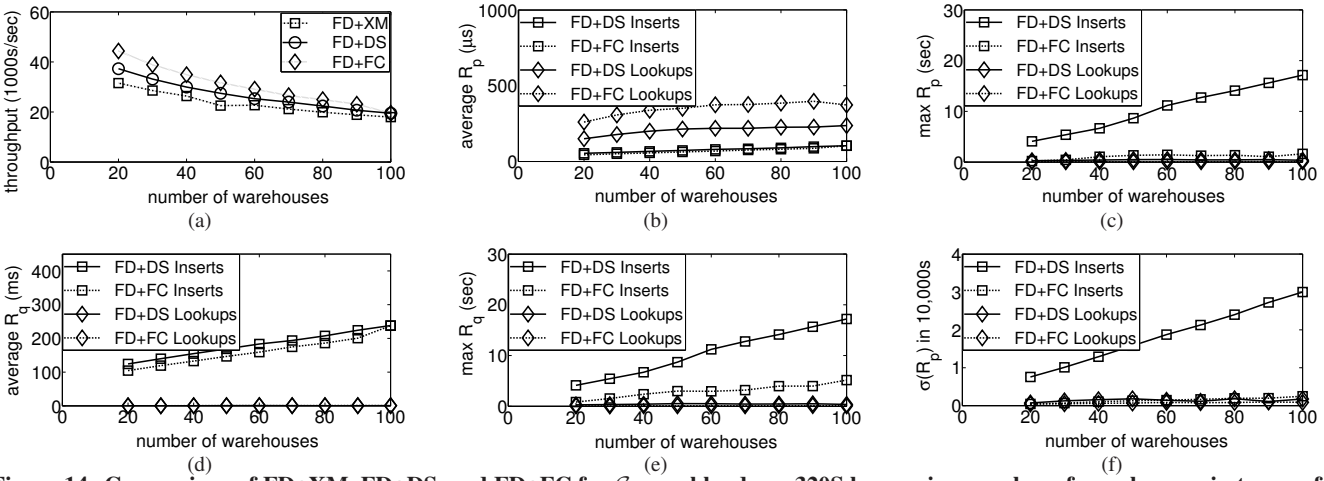

Figure 14: Comparison of FD+XM, FD+DS, and FD+FC for $\mathcal{G}_T$ workloads on 320S by varying number of warehouses, in terms of: total completion time (a), average insertion and lookup R_p (b), worst-case insertion and lookup R_p (c), average insertion and lookup R_q (d), worst-case insertion and lookup R_q (e), and standard deviation in R_p (f).

Figure 12 shows results for $\mathcal{G}_R$ workload on X25-E with write-back disk caching enabled. Disk caching allows effective usage of the SSD's on disk cache, however, data loss could occur when it experiences an unscheduled power off. Most trends observed for

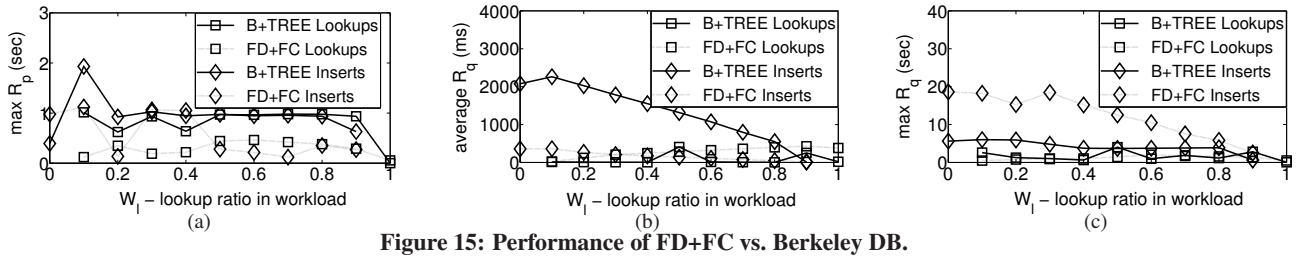

Figure 15: Performance of FD+FC vs. Berkeley DB.

the write-through experiment (see Figure 6) show in these results too. However, because writes are faster as they are not actually stored to the hardware in the SSD, we see more similarities with the 320S experiment (see Figure 13).

Remark A.13 (Extra results for FD+FC vs. B+tree; Section 5.3)

Figure 15 completes the results for the FD+FC vs. B+tree comparision shown in Figure 10. Worst-case R_p of FD+FC is similar to, or in most cases lesser than, B+tree's worst-case R_p. B+tree requires modifying atleast one block for every update to the index. Such frequent random page updates trigger many block erases inside the SSD ultimately affecting the worst-case response time. Figures 15(b) and 15(c) show average and worst-case R_q comparision for the two indexes. Note that for update intensive workloads, FD+FC is by far better, while B+tree outperforms FD+FC for lookup intensive workloads, which is expected. As mentioned in Section 5.3, Berkeley DB's B+tree implementation is very sophisticated while our FD+FC is a prototype we developed to capture a fair comparision with FD+DS and FD+XM. We believe FD+FC code can benefit a lot from further code optimizations.

Remark A.14 (Pseudocode for FD+Tree Operations; Section 2) See Algorithms 1 and 2.

Remark A.15 (Proof of Theorem 1; Section 2.4) We begin by introducing several lemmas.

Lemma 1. *An FD+-tree with n elements in its lowest level has $O(\log_\gamma \frac{n}{\kappa_0\beta})$ levels.*

Proof. Without loss of generality, suppose $h > 2$. By (I3), $B(L_{h-1}) > \kappa_{h-2} = \kappa_0\gamma^{h-2}$, so

$$\begin{aligned} h &< \log_\gamma(B(L_{h-1})/\kappa_0) + 2 \le \log_\gamma(\lceil n/\beta\rceil/\kappa_0) + 2 \\ &= O(\log_\gamma \frac{n}{\kappa_0\beta}). \end{aligned}$$

□

Lemma 2. *The total number of blocks in $L_0, \ldots, L_m$ is $O(\kappa_0\gamma^m)$.*

Proof. By (I3), L_i has at most κ_i blocks, so the total is at most $\sum_{i=0}^{m} \kappa_i = \kappa_0 \frac{\gamma^{m+1}-1}{\gamma-1} = O(\kappa_0\gamma^m)$. □

In the following, an *L_m-merge* refers to a merge that replaces levels $L_0, \ldots, L_m$, where m is calculated by merge-prepare.

Lemma 3. *The number of blocks written by an L_m-merge that produces b blocks in L_m is less than $m + \frac{\beta}{\beta+1}b = O(\kappa_0\gamma^m)$.*

Proof. The levels above L_m contain densely packed fences, so the total number of blocks written is at most

$$\begin{aligned} &b + \lceil b/\beta\rceil + \lceil\lceil b/\beta\rceil/\beta\rceil + \cdots \\ &< b + m + b/\beta + b/\beta^2 + \cdots = m + \frac{\beta}{\beta-1}b = O(\kappa_0\gamma^m). \end{aligned}$$

The last step follows from (I3): L_m has at most κ_m blocks. □

Lemma 4. *Suppose an overflow-triggered L_m-merge has just completed. There must have been $\Omega((\beta-\gamma)\kappa_0\gamma^{m-1})$ insertion/deletion requests since the most recent $L_{m'}$-merge with $m' \ge m$.*

Proof. Let $\mathfrak{M}$ denote the L_m-merge and $\mathfrak{M}'$ denote the $L_{m'}$-merge.

Right before $\mathfrak{M}$, we must have $\widehat{U}(m-1) > \kappa_{m-1}$ (otherwise merge-prepare would not have called for an L_m-merge). Therefore,

$$\left\lceil \sum_{j=0}^{m-1} B(L_j) - \frac{1}{\beta}\sum_{j=1}^{m-1} B(L_j) \right\rceil > \kappa_{m-1},$$

by Eq. (1) (it is easy to see that the above still holds in the case of $m = h$ where $\widehat{U}(h-1)$ has a precise formulation based on N_Δ and N_∇). Since $B(L_0) = \kappa_0$ is necessary to trigger $\mathfrak{M}$, we have:

$$\begin{aligned} \kappa_0 + \frac{\beta-1}{\beta}\sum_{j=1}^{m-1} B(L_j) &> \kappa_0\gamma^{m-1}, \\ \sum_{j=1}^{m-1} B(L_j) &> \frac{\beta}{\beta-1}\kappa_0(\gamma^{m-1}-1). \end{aligned} \tag{5}$$

The total number of data entries in $L_0, \ldots, L_{m-1}$ right before $\mathfrak{M}$ is given by $E - F$, where E is the total number of entries and F is the total number of fences among them. We have:

$$\begin{aligned} E &\ge \beta\kappa_0 + \sum_{i=1}^{m-1}(\beta(B(L_i)-1)+1) \\ &= \beta\kappa_0 - (\beta-1)(m-1) + \beta\sum_{i=1}^{m-1} B(L_i); \\ F &\le \left(B(L_{\mathsf{succ}(m-1)}) + \sum_{i=1}^{m-1} B(L_i)\right) + \sum_{i=0}^{m-1} B(L_i) \\ &= B(L_{\mathsf{succ}(m-1)}) + \kappa_0 + 2\sum_{i=1}^{m-1} B(L_i). \end{aligned} \tag{6}$$

The first term on the right of Ineq. (6) is the number of blocks that need to be pointed to by fences, and the last term of Ineq. (6) upper-bounds the number of fences inserted for (I2) at the beginning of each block. Combining the two quantities above, we get:

$$\begin{aligned} E - F &\ge (\beta-1)(\kappa_0 - m + 1) - B(L_{\mathsf{succ}(m-1)}) \\ &\quad + (\beta-2)\sum_{i=1}^{m-1} B(L_i) \\ &> (\beta-1)(\kappa_0 - m + 1) - \kappa_0\gamma^m && \text{by (I5)} \\ &\quad + \frac{(\beta-2)\beta}{\beta-1}\kappa_0(\gamma^{m-1}-1) && \text{by Ineq. (5)} \\ &= \Omega(\beta\kappa_0\gamma^{m-1} - \kappa_0\gamma^m) \\ &= \Omega((\beta-\gamma)\kappa_0\gamma^{m-1}). \end{aligned}$$

Algorithm 1: Insert and delete algorithms for FD+tree.

```
1  Insert(I, k, v) begin
     // I: index; k: key to insert; v: payload for k;
2    L_0.addInsertEntry(k, v); N_Δ ← N_Δ + 1;   // add (k, v) pair to the top-level of index
3    if L_0.size() > βκ_0 then
4      if Û(h − 1) > κ_{h−1} then
5        m ← h;
6      else
7        m ← arg min_i(Û(i) ≤ κ_i);
8      merge(I, m);                              // overflow-triggered merge
9  end
10 Delete(I, k, v) begin
     // I: index; k: key to delete; v: payload for k;
11   if L_0.getInsertEntry(k, v) ≠ φ then
12     L_0.deleteInsertEntry(k, v);
13     N_Δ ← N_Δ − 1 ;
14   else
15     L_0.addDeleteEntry(k, v);
16     N_∇ ← N_∇ + 1 ;
17   if L_0.size() > βκ_0 then
18     if Û(h − 1) > κ_{h−1} then
19       m ← h;
20     else
21       m ← arg min_i(Û(i) ≤ κ_i);
22     merge(I, m);                              // overflow-triggered merge
23   if N_∇/N_Δ > 1/3 then
24     merge(I, h − 1);                          // underflow-triggered merge
25 end
26 Merge(I, m) begin
     // I: index to reorganize; m: no. of participating levels
27   foreach k ∈ (I.min, I.max) do
28     ∀i ∈ [0, m], S_i ← L_i.getMatchingDataEntries(k);   // pull data entries with key k from all levels of index
29     S ← coalesce(∪_{i∈[0,m]} S_i);            // remove canceling insertions and deletions
30     C = (Σ_{i∈[0,m]} |S_i| − |S|)/2;
31     N_Δ ← N_Δ − C; N_∇ ← N_∇ − C;             // update N_Δ and N_∇
32     R ← S ;                                   // store in R, entries to append to L_m^new
33     if L_m.getMatchingFence(k) ≠ φ then       // if a fence with key k exists
34       p_last ← L_m.getMatchingFence(k); R ← R ∪ (k, p_last) ;   // update p_last and R
35     foreach i ∈ [m, 0] do
36       if L_i^new.canAccommodate(R) then
37         L_i^new.append(R); break;             // append R to L_i and stop
38       else
39         F = L_i^new.newNode(); L_i^new.append(k, p_last);   // attach a new node to L_i^new; set its first entry to the fence (k, p_last)
40         L_i^new.append(R);R ← (k, F);         // append R; assign F to R
41     L_0 ← L_0 − S_0;                          // delete retrieved record entries from L_0
42     foreach i from 1, . . . , m do
43       if S_i ≠ φ ∧ keys in L_i.headNode are smaller than k then
44         H ← L_i.headNode; L_i.headNode ← H.next; delete H;
           // move to next node; and delete current headNode
45     foreach i ∈ [0, m] do
46       delete L_i.headNode;                    // delete last remaining node in L_i
47     foreach i ∈ [0, m] do
48       o_i = arg min_j ( B(L_i^new) ≤ κ_j ∧ B(L_{succ(i)}^new) ≤ κ_{j+1} );
49     z ← arg max_i(o_i = 0);
50     if z > 0 then
51       L_0.deleteAllFenceEntries(); F ← L_z.getAllFenceEntries();
         L_0.addFenceEntries(F) ;                // replace fences in L_0 with those in L_z
52       ∀i ∈ [1, z], L_i^new.dropLevel() ;      // delete levels L_1, . . . , L_z
53     L_0^new.assignLabel(L_0);
54     foreach i ∈ (z, m) do
55       L_i^new.assignLabel(L_{o_i});           // relabel level L_i^new as L_{o_i}
56 end
```

Algorithm 2: Lookup algorithm for FD+tree

```
1  Lookup(I, k) begin
2    f ← φ;
3    foreach i = 0, . . . , m do
4      R_I ← L_i.getMatchingInsertEntry(f, k);
5      if R_I ≠ φ then
6        return R_I;
7      R_D ← L_i.getMatchingDeleteEntry(f, k);
8      if R_D ≠ φ then
9        break;
10     f ← L_i.getNextFenceEntry(f, k); // from L_i's node pointed by f, get fence f̄ s.t. min_{f̄.key}{f̄.key ≤ k}
11   return φ;
12 end
```

Right after $\mathfrak{M}'$, none of $L_0, \ldots, L_{m-1}$ contains any data entries. Therefore, all of the $E - F$ data entries above must have been the results of insertion/deletion requests since $\mathfrak{M}'$. □

Lemma 5. *Consider a phase that begins right after a full merge and ends right after the next full merge. Let N denote the true number of elements indexed at the beginning of the phase. During this phase, the worst-case I/O cost of a lookup is $O(\log_\gamma \frac{N}{\kappa_0\beta})$, and the amortized I/O cost of an insertion or deletion is $O(\frac{\gamma}{\beta-\gamma}\log_\gamma \frac{N}{\kappa_0\beta})$.*

Proof. Let h denote the height of the tree at the beginning of the phase. At this point (right after a full merge), all N elements are in the lowest level, so $h = \log_\gamma \frac{N}{\kappa_0\beta}$ by Lemma 1. Note that the tree remains at this height until the full merge that ends the phase.

A lookup visits one block per materialized level, so the bound on its cost immediately follows. We now turn to insertions and deletions. Let R denote the total number of insertion/deletion requests in the phase. All I/O's of insertions and deletions are incurred by merges. These merges include L_i-merges where $1 \le i \le h-2$ (all of which are overflow-triggered), and one L_{h-1}- or L_h-merge, which ends the phase.

For each $i \in [1, h-2]$, consider the sequence $\mathcal{M}_i$ of all L_i-merges during the phase. Let C_i denote the total I/O cost incurred by merges in $\mathcal{M}_i$. For each merge $\mathfrak{m} \in \mathcal{S}_m$, let $C(\mathfrak{m})$ denote its I/O cost and $R_i(\mathfrak{m})$ denote the number of insertion/deletion requests since the last $L_{i'}$-merge since $\mathfrak{m}$ with $i' \ge i$. We have:

$$\frac{C_i}{R} = \frac{1}{R}\sum_{\mathfrak{m}\in\mathcal{M}_i} C(\mathfrak{m}) < \frac{\sum_{\mathfrak{m}\in\mathcal{M}_i} C(\mathfrak{m})}{\sum_{\mathfrak{m}\in\mathcal{M}_i} R_i(\mathfrak{m})}.$$

By Lemmas 2, 3, and 4, for all $\mathfrak{m} \in \mathcal{M}_i$,

$$\frac{C(\mathfrak{m})}{R_i(\mathfrak{m})} = O\left(\frac{\kappa_0\gamma^i}{(\beta-\gamma)\kappa_0\gamma^{i-1}}\right) = O\left(\frac{\gamma}{\beta-\gamma}\right).$$

Therefore, $C_i/R = O(\frac{\gamma}{\beta-\gamma})$.

Let $\mathfrak{M}$ denote the full merge that ends the phase, and let C_{h-1} denote its cost. There are several cases and we show that for each case, $C_{h-1}/R = O(\frac{\gamma}{\beta-\gamma})$:

- $\mathfrak{M}$ is an overflow-triggered L_{h-1}-merge. In this case, we have $C_{h-1} = O(\kappa_0\gamma^{h-1})$ by Lemmas 2 and 3 and $R = \Omega((\beta-\gamma)\kappa_0\gamma^{h-2})$ by Lemma 4, so $C_{h-1}/R = O(\frac{\gamma}{\beta-\gamma})$.
- $\mathfrak{M}$ is an overflow-triggered L_h-merge, which grows the tree by one level. In this case, $\mathfrak{M}$ reads the existing $L_0, \ldots, L_{h-1}$ and adds a new L_h. By Lemma 2, $\mathfrak{M}$ reads $O(\kappa_0\gamma^{h-1})$ blocks. Furthermore, $\mathfrak{M}$ writes no more blocks in L_h than it reads from $L_0, \ldots, L_{h-1}$, so the total number of blocks written by $\mathfrak{M}$, by Lemma 3, is $O(h + \frac{\beta}{\beta+1}\kappa_0\gamma^{h-1}) = O(\kappa_0\gamma^{h-1})$. Therefore, $C_{h-1} = O(\kappa_0\gamma^{h-1})$.
  To arrive at $C_{h-1}/R = O(\frac{\gamma}{\beta-\gamma})$, we can show that $R = \Omega((\beta-\gamma)\kappa_0\gamma^{h-2})$ using the same argument as in the proof of Lemma 4, by noting that $\widehat{U}(h-2) > \kappa_{h-2}$ before $\mathfrak{M}$

and therefore the number of data entries at or above L_{h-1} is $\Omega((\beta-\gamma)\kappa_0\gamma^{h-2})$.

- $\mathfrak{M}$ is underflow-triggered. In this case, we know $N_\nabla/N_\Delta > 1/3$ right before $\mathfrak{M}$. Note that all N elements indexed at the beginning of the phase are insert entries in the lowest level, and have remained undisturbed until $\mathfrak{M}$. Therefore, $R \geq N_\nabla > \frac{1}{3}N_\Delta \geq \frac{1}{3}N$.
  The cost of reading $L_0, \ldots, L_{h-2}$ is $O(\kappa_0\gamma^{h-2})$ by Lemma 2, and the cost of reading L_{h-1} is $\lceil N/\beta \rceil > \kappa_0\gamma^{h-2}$ by (I4). Therefore, $C_{h-1} = O(N/\beta)$. Since we have shown $R \geq \frac{1}{3}N$ earlier, $C_{h-1}/R = O(1/\beta)$, and $1/\beta < \frac{\gamma}{\beta-\gamma}$.

Overall, the amortized I/O cost of an insertion or deletion over the phase is:

$$\frac{1}{R}\sum_{i=1}^{h-1} C_i = O\left(\frac{h\gamma}{\beta-\gamma}\right) = O\left(\frac{\gamma}{\beta-\gamma}\log_\gamma \frac{N}{\kappa_0\beta}\right). \quad \square$$

Proof of Theorem 1. To show the space bound, consider the bottom level L_{h-1}. By (I4), $B(L_{h-1}) > \kappa_0\gamma^{h-2}$; by Lemma 2, the total number of blocks in all levels except L_{h-1} is $O(\kappa_0\gamma^{h-2})$. Therefore, the total number of blocks in the tree is $O(B(L_{h-1}))$. Note that L_{h-1} contains only insert entries; suppose there are n of them. Clearly, $O(B(L_{h-1})) = O(n/\beta)$. By (I6), $N_\nabla \leq \frac{1}{3}N_\Delta$, so $N = N_\Delta - N_\nabla \geq \frac{2}{3}N_\Delta$. Hence, $n \leq N_\Delta \leq \frac{3}{2}N$, and $O(n/\beta) = O(N/\beta)$.

To prove the time bounds, divide the workload into phases separated by full merges, and consider each phase. Let N_0 denote the true number of elements indexed at the beginning of this phase. By Lemma 5, the worst-case I/O cost of a lookup is $O(\log_\gamma \frac{N_0}{\kappa_0\beta})$, and the amortized I/O cost of an insertion or deletion is $O(\frac{\gamma}{\beta-\gamma}\log_\gamma \frac{N_0}{\kappa_0\beta})$ during this phase. To complete the proof, it suffices to show that $N_0 = O(N)$ throughout the phase. By (I6), $N_\nabla/N_\Delta \leq 1/3$, so

$$N = N_\Delta - N_\nabla \geq 2N_\Delta/3 \geq 2N_0/3.$$

The last step above follows from the observation that all N_0 elements are insert entries in the lowest level, and they remain undisturbed until the end of the phase. $\square$

Remark A.16 (Correctness of FD+FC; Section 4.5) A complete and rigorous proof for the correctness of FD+FC requires enumerating many cases and building blocks. Instead of being exhaustive, we present and prove in the following a series of lemmas leading to one of the most important building blocks, to give a flavor for the complete proof.

In the following, let $H(\cdot)$ denote the head block of a level.

Lemma 6. *During a merge, all head blocks in the old upper levels are connected by a path.*

Proof. We prove by contradiction. Suppose $H(L_i^{\mathsf{old}})$ does not have a fence to $H(L_{i+1}^{\mathsf{old}})$. Then there are two cases:

- $H(L_i^{\mathsf{old}})$'s first fence must be pointing to the right sibling of $H(L_{i+1}^{\mathsf{old}})$. Also, $H(L_i^{\mathsf{old}})$'s first fence has already been processed by merge; in fact, that happened when $H(L_i^{\mathsf{old}})$'s left sibling was deleted. But if its first fence has already been processed, so should $H(L_{i+1}^{\mathsf{old}})$'s because they share the same key. Hence, this case cannot happen.
- $H(L_i^{\mathsf{old}})$'s first fence points to an already deleted block of L_{i+1}^{old}. Hence, $H(L_{i+1}^{\mathsf{old}})$'s fence must be in the right sibling of $H(L_i^{\mathsf{old}})$, or further down the list. If that is the case, $H(L_i^{\mathsf{old}})$ should have already been deleted because $H(L_{i+1}^{\mathsf{old}})$'s first fence has already been processed by the merge; in fact, that happened when $H(L_{i+1}^{\mathsf{old}})$'s left sibling was deleted. Hence, this case cannot happen either. $\square$

Lemma 7. *If m-delete is about to be invoked to delete a head block $H(L_i^{\mathsf{old}})$, the head block's content has only one path from L_0^{old}.*

Proof. Let the head block's contents span the range $[a, b)$, where a is $H(L_i^{\mathsf{old}})$'s first fence's key, and b is its right sibling's first fence's key. We prove by contradiction. Suppose there are multiple paths. Multiple paths are possible only if some level L_j^{old} above L_i^{old} has a set of consecutive blocks including its head block, whose ranges are subsets of $[a, b)$. Essentially, lookups aimed at each of these blocks would also have to reach $H(L_i^{\mathsf{old}})$, and the paths they take are the multiple paths we are referring to. But if there is such a case, then $H(L_i^{\mathsf{old}})$'s right sibling's first fence will be larger than every one of these blocks. Therefore, all of these blocks should have been long deleted. This is a contradiction. $\square$

Lemma 8. *If a lookup entered the old tree to search $H(L_i^{\mathsf{old}})$, while immediately afterward an m-delete step entered to delete the same head block, then our locking scheme ensures the head block is not deleted at least until the lookup has completed processing on it.*

Proof. From Lemma 7, the head block has only one path from L_0^{old}. From Lemma 6, this path must be the path connecting all head blocks. Therefore, both lookup and m-delete have to follow this path to reach $H(L_i^{\mathsf{old}})$. Lookup does not unlock a parent block, unless it receives a lock on the child. Thus, m-delete cannot jump through the lookup and reach $H(L_i^{\mathsf{old}})$ earlier. Therefore, our locking scheme ensures that $H(L_i^{\mathsf{old}})$ will not be deleted at least until the lookup completes processing on it. $\square$

Remark A.17 (Addressing FD+DS's worst-case modification response time; Section 3) In Section 3, we have discussed why FD+DS suffers from poor worst-case modification response times, which is confirmed by our experiments in Section 5. Recall that FD+DS does not free the memory occupied by L_0^{old} during a merge; thus, as soon as memory for the top level is full, new modifications will have to wait for the entire merge to complete, resulting in high response times. Naturally, the question arises whether we can patch FD+DS in some way to avoid this issue. Here, we discuss a few possibilities (assume the reader has already read Section 4).

Removing data entries from L_0^{old} We can allow the merge to remove data entries from L_0^{old}, just as in FD+FC. However, the conceptual simplicity of FD+DS would be lost. Lookups accessing keys smaller than the key of the wavefront will have to search the new levels, because a matching entry in L_0^{old} may have been removed and placed in L_m^{new}. Since the new levels are also being modified by the ongoing merge, a concurrency control protocol is needed for coordinating such lookups with the merge. Hence, this approach will be as complex as FD+FC.

More specifically, the above approach and FD+FC use five main components of concurrency control: (P1) for L_0^{new}; (P2) for L_0^{old}; (P3) for the new disk-resident levels; (P4) for the old disk-resident levels; and (P5) to track lookups in old levels (so such lookups can finish before the old levels are reclaimed). Note that the approach we discussed in the previous paragraph needs (P1), (P2), (P3) and (P5), and is significantly more complex than the basic FD+DS, which needs (P1) and (P5). FD+FC needs (P1)–(P4); it does not need (P5) since the reclamation of old levels is already done by the merge. (P4)'s implementation is similar to (P3).

Cost of (P4) as implemented in FD+FC is not significant because the merge accesses the old levels at the time of a node deletion (not for every entry deletion), and only the first node of each level is

accessed. Those nodes lying in the path to the about-to-be-deleted nodes are only locked and unlocked without incurring any I/O or processing cost. An ongoing merge will not interfere with most lookups because a lookup may be at any node of a level, while the merge examines only the first node. In sum, letting FD+DS remove data entries from L_0^{old} will not be significantly different from FD+FC complexity- or performance-wise, but still has the disadvantage of doubling disk space.

Triggering merge proactively We can trigger a merge proactively, once a predetermined fraction of the memory allocated for the top level fills up. While L_0^{old} still cannot be reclaimed until the merge completes, the fraction of memory that remains can accommodate additional incoming modifications before stalling.

There are two main problems with this approach. First, the approach only delays the inevitable—the problem remains that memory associated with L_0^{old} is stuck until the merge completes. In additional experiments we ran on X25E, we triggered merges when there were still 25% memory remaining, and observed the worst-case insertion R_p for this approach to be between 20 to 30 seconds. When we triggered merges when memory was only 50% full, the worst-case insertion R_p was still between 15 to 30 seconds. From Section 5.1, we know that without proactive merging, FD+DS's worst insertion R_p is around 33 to 38 seconds. Thus, proactive merging certainly helps. However, it remains much worse than FD+FC, whose worst-case insertion R_p is under a second.

Second, trigger merges more proactively means more merges overall, and hence more writes to the SSD. For an 80%-update workload, FD+DS with merges triggered when memory was 75% full generated 12% more page writes to the disk (after counting cache effects). Extra writes to the SSD jumped to 33% when merges are triggered when memory was 50% full.

Writing L_0^{old} temporarily to disk Yet another approach is to write L_0^{old} to disk at the beginning of a merge, so the entire memory can be used as L_0^{new} to accommodate incoming modifications. Lookups in the old levels will now need to start with a disk-resident L_0^{old}; otherwise, no other changes to FD+DS are needed.

The disk-resident L_0^{old} would need to be written in a way to allow efficient search. We can use a tightly packed B+tree, where the smallest entry of every leaf node needs to be a fence (otherwise a lookup may need to read multiple B+tree leaves to find a fence pointing to L_1^{old}). This B+tree results in at least one extra I/O per lookup (and more if there are more levels). This overhead is significant considering that FD+trees typically do not have many levels (4 in our experiments with reasonably large datasets). One could ameliorate this problem by caching the B+tree pages of L_0^{old}, but this fix would amount to using more memory than originally allotted, which would equally benefit other approaches such as FD+FC.

This approach also incurs more writes. Writing L_0^{old} to disk at the beginning of every merge means that every data record will be written one extra time (when it exits the top level).

Range-partitioning the index into smaller ones This approach partitions the key domain into continuous ranges, and uses one subindex for each partition. The top levels of these subindexes together share the entire memory allotted. Each subindex carries out its merge exactly as in FD+DS, independently of others. While this approach does not change the fact that each ongoing merge pins down its old top level in memory, the upside is that it only takes the portion of the main memory occupied by the subindex being reorganized. Similarly, this approach in the worst case still doubles the space requirement like FD+DS, but if few subindexes have ongoing merges simultaneously the space overhead will be lower.

For this approach to work effectively, range partitioning and memory allocation across subindexes must be done intelligently and adaptively, which adds considerable complexity. Otherwise, one large subindex can end up containing most records and taking most of the memory, and its merge will have a similar issue as the basic FD+DS. Dynamic adaption is tricky to implement because we need to avoid oscillating back and forth between states and constantly incurring the adaption overhead, and because adaption complicates concurrency control.

Finally, for workloads such as insertions with uniformly random keys, it is possible that merges will be triggered for many subindexes at roughly the same time. In that case, we need to either pin down many top levels simultaneously, which would lead to the same problem as basic FD+DS, or cap the number of concurrent merges, which would halt modifications in subindexes with pending merges.

Discussion In summary, the various approaches discussed above either introduce other performance issues or complicate FD+DS to the point where it becomes no simpler than FD+FC. Some ideas—such as proactive merge, allowing more memory sharing (among top levels and even page cache), and adaptive partitioning—can be applied orthogonally to FD+FC as well, and it would be interesting to investigate their effectiveness further. However, the fundamental difference between FD+DS and FD+FC and its implication on performance still remain.