# 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.

## Categories and Subject Descriptors

H.2.2 [**Physical Design**]: Access Methods; H.2.4 [**Systems**]: Concurrency

## Keywords

Indexing, Concurrency control, Flash-memory SSDs

## 1 Introduction

*Solid-State Drives* (*SSDs*) have become a viable alternative to magnetic disks in database systems [8, 11, 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* [21], *LA-tree* [2], *FD-tree* [12], 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.

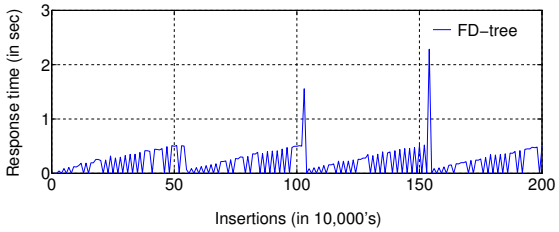


**Figure 1: Completion time of an insertion request—including any index reorganization triggered—for (an improved version of) FD-tree [12], over the course of two million insertions, starting with an empty index. The insertions are grouped into buckets each containing 10,000 requests; we plot the longest 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 research 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 [14]. Engineers at Web-based companies like Facebook care about minimizing variance and ensuring that the "edge cases" are not bad, even at the potential cost of higher average latency [1].

**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


*CIKM'12,* October 29–November 2, 2012, Maui, HI, USA.

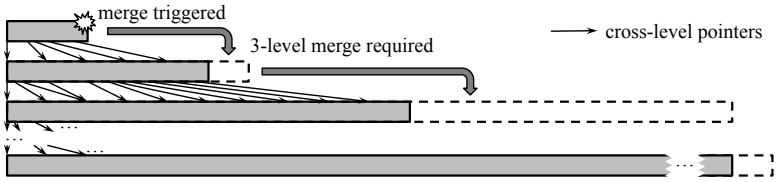


**Figure 2: Illustration of high-level ideas in FD+tree. Maximum capacity of each level is delineated by dashed lines.**

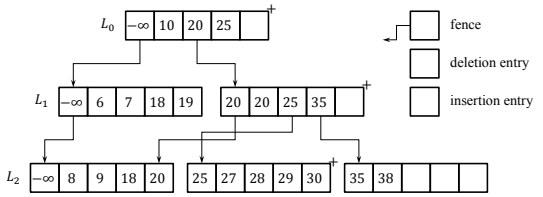


**Figure 3: An example FD+tree. Blocks visited by the lookup for key $25$ are marked with "+".**

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 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 implemented and evaluated FD+FC against alternative schemes.

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. [12]. 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 [12]. 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 invariants, and performs more careful bookkeeping, pre-reorganization planning, and post-reorganization adjustment.

## 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]

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 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, $N_\Delta$ and $N_\nabla$, 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.

Let $\beta$ denote the *block size*, as measured by the number of entries that fit within one block. Let $\gamma$ 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 $\kappa_i$ 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.

[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, 12].
[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.

- **(I5)** *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.

FD+tree supports another optimization, *level skipping*, whose details we omit for simplicity and space limit. The intuition is illustrated in Figure 4. After reorganization, a chain of single-block levels may be generated, which degrades lookup performance. Level skipping allows unnecessary intermediate blocks in this chain to be bypassed, thereby improving performance. For additional details, including its formalization in terms of data structure invariants, see the technical report version of this paper [20].

## 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 $N_\Delta$; otherwise, the entry being deleted is below $L_0$, so we add a delete entry to $L_0$ and increment $N_\nabla$.

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., (I5) 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 $L_{i+1}$ pointed to by $f$.[5] 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

A 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. The last new level written by the merge, which we denote by $L_{\overline{m}}$, consolidates and stores all data entries from the old $m+1$ levels; it also stores fences to $L_{m+1}$ (if any). When the merge ends, new levels above $L_{\overline{m}}$ store only fences. Figure 4 illustrates how merge works. Here, $m = \overline{m} = 3$. In practice, $\overline{m}$ can be less than $m$ when many input entries cancel each other.

Suppose the tree height is $h$ before the merge. A merge that reads all levels is called a *full merge*. A full merge with $\overline{m} = h$ (it can be shown that $m \leq h$) 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 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:

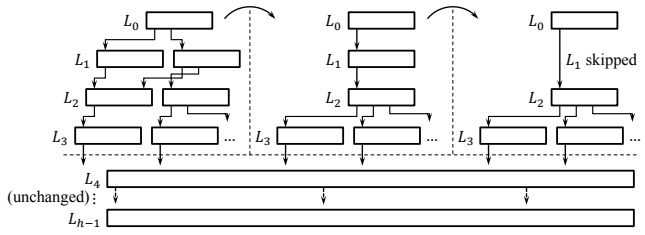


**Figure 4: Illustration of FD+tree merge and motivation for level skipping. From right to left, we show the old levels, the new levels that would have been produced by merge without level skipping, and the actual new levels produced by merge (with level skipping). FD+FC directly produces the new levels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

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

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 $N_\Delta$ and $N_\nabla$. 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.[6]

Note that calculating $\widehat{U}(i)$ and $m$ is computationally trivial, so merge-prepare adds no discernible cost to the overall merge.

**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 $L_0^{\mathsf{new}}, \ldots, L_m^{\mathsf{new}}$.[7]

During the merge, we maintain $p_{\mathsf{last}}$, a pointer to the last block in $L_{m+1}$ for which we have added a fence to $L_m^{\mathsf{new}}$. This information is needed to ensure (I2) and, specifically, that the first entry in every $L_m^{\mathsf{new}}$ 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 $L_{i+1}$, we will add $f$ to $L_m^{\mathsf{new}}$. 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 $L_m^{\mathsf{new}}$. $S$ 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 $N_\Delta$ and $N_\nabla$ to reflect the effect of replacing $S_m, S_{m-1}, \ldots, S_0$ with $S$.

If the last block of $L_m^{\mathsf{new}}$ 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 $L_m^{\mathsf{new}}$, we also update $p_{\mathsf{last}}$.

If $L_m^{\mathsf{new}}$ has insufficient space, we begin a new block of $L_m^{\mathsf{new}}$ 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

[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).

[5] With level skipping, it is possible that $L_{i+1}$ is skipped, so more precisely, the search should continue in the next non-skipped level. For simplicity we ignore this technicality in our discussion; see [20] for a formal description.

[6] 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.

[7] 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.

are fine; otherwise, we add a fence with $key$ and $p_{\mathsf{last}}$ as the first entry of the new block.

When we begin a new block of $L_m^{\mathsf{new}}$, we need to add to $L_{m-1}^{\mathsf{new}}$ a fence to this block. If $L_{m-1}^{\mathsf{new}}$'s last block has no space left, we begin a new block for $L_{m-1}^{\mathsf{new}}$ to put the fence in (which automatically satisfies (I2)). A fence to this new block must then be added to $L_{m-2}^{\mathsf{new}}$. Such fence additions may propagate all the way up to $L_0^{\mathsf{new}}$; merge-prepare ensures that $L_0^{\mathsf{new}}$ 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.

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. Because of space limit, we omit the details, which can be found in [20].

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 [20] 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 [12] in significant ways. Out of FD+tree's 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 [12] assume insertion-only workloads. 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

Before presenting our full solution in Section 4, we present two other approaches 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

[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.

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.

**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.

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* ($L_0^{\mathsf{new}}$). 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, $L_0^{\mathsf{new}}$ becomes the new $L_0$.
- *Old top level* ($L_0^{\mathsf{old}}$). 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, $L_m^{\mathsf{new}}$ 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 $L_{m+1}$ from $L_m^{\mathsf{old}}$ to $L_m^{\mathsf{new}}$.

During $\mathfrak{M}$, the first entry in $L_0^{\mathsf{old}}$ 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 $L_1^{\mathsf{old}}$. Its key indicates how far the merge has progressed, with the following invariant:

- **(I6)** *Wavefront*: Consider all data entries and fences for $L_{m+1}$ 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 $L_0^{\mathsf{new}}$. Those with keys strictly greater than the wavefront key can be found by searching from $L_0^{\mathsf{old}}$.

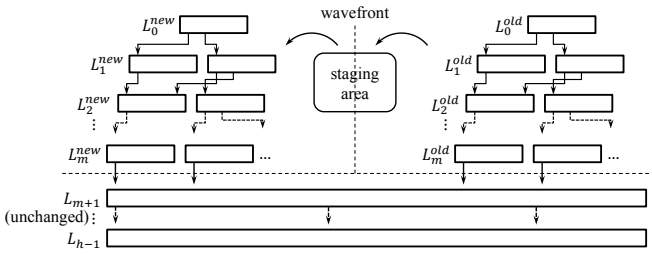


**Figure 5: Components of FD+FC during a merge of $L_0, \ldots, L_m$.**

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 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 $L_0^{\mathsf{old}}, \ldots, L_m^{\mathsf{old}}$ to $L_m^{\mathsf{new}}$. 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.

**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.

**Dynamic Memory Sharing between Top Levels** Both $L_0^{\mathsf{old}}$ and $L_0^{\mathsf{new}}$ 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 $L_0^{\mathsf{old}}$ and $L_0^{\mathsf{new}}$ 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 $L_0^{\mathsf{old}}$; we always reserve $\kappa_1$ slots for $L_0^{\mathsf{new}}$ to accommodate fences to be produced by $\mathfrak{M}$. As $\mathfrak{M}$ progresses, space taken by entries removed from $L_0^{\mathsf{old}}$ is given to $L_0^{\mathsf{new}}$ for new modifications; a modification may have to wait for $\mathfrak{M}$ to make new space. At the end of $\mathfrak{M}$, $L_0^{\mathsf{old}}$ becomes empty, and $L_0^{\mathsf{new}}$ becomes $L_0$.

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

## 4.3 Modification and Lookup

A modification goes to $L_0^{\mathsf{new}}$ if there is an ongoing merge; otherwise it goes to $L_0$. It waits if no space is available in $L_0^{\mathsf{new}}$. Then, it locks $L_0^{\mathsf{new}}$ 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 $L_0^{\mathsf{new}}$ (but without going below $L_0^{\mathsf{new}}$). We can stop if we find a data entry with $key$ here. If not, we search $L_0^{\mathsf{old}}$, and through it, the old upper levels and below-merge levels.
- Otherwise, we search $L_0^{\mathsf{new}}$, 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 $L_0^{\mathsf{old}}$ and $L_0^{\mathsf{new}}$, 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, *merge-prepare*, *merge-execute*, and *merge-finalize*. As discussed in Section 4.2, merge-prepare additionally estimates which result levels to skip, so merge-execute can avoid materializing unnecessary levels. On the result of merge-execute, merge-finalize further performs level tightening and skipping, in case that merge-prepare's estimation turns out too conservative.

In the following, we focus on the *merge-execute* step, highlighting how it performs concurrency control. Details on the other two steps can be found in [20].

**Starting Merge-Execute** At this point, the wavefront fence (the first fence in $L_0^{\mathsf{old}}$) has key $-\infty$ and points to the first block of $L_1^{\mathsf{old}}$. 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 $L_m^{\mathsf{new}}$ 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 $L_0^{\mathsf{old}}$ 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 $L_0^{\mathsf{old}}$. First, m-delete updates the wavefront fence in $L_0^{\mathsf{old}}$: 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 $L_0^{\mathsf{old}}$.

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.1 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 $L_0^{\mathsf{old}}$ 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 $L_0^{\mathsf{old}}$, we finally delete the wavefront fence and make $L_0^{\mathsf{new}}$ 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 $L_0^{\mathsf{old}}$ 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.

## 4.5 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 [20].

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 $N/\beta$) 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.

# 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 [20]. 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; again, see [20] 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.2 and A.3 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 [20] 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 presented in [20].

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 ($\kappa_0$, but measured in bytes), size ratio ($\gamma$), 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 $W_l$ 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.

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 is around 27% and 20% 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 than 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 as well as FD+DS except for update-heavy workloads; lock/unlock calls of frequently triggered merges slow down the lookups. FD+XM's lookup performance is much worse, because lookups are blocked whenever there is an ongoing merge.

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 the entries only after the merge completes. When the merge involves many levels (for ex., a full merge), this scheme is obviously costly. For lookups, FD+FC and FD+DS show similar worst $R_p$'s (around 500 milliseconds), well below FD+XM.

Figure 6(f) shows equi-width histograms with a bucket length of 2.5 milliseconds for lookup $R_p$'s. Similar trends were observed for insertions. 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 FD+DS.

Figures 6(h)-6(l) compare the schemes on the $R_q$ metric (recall 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

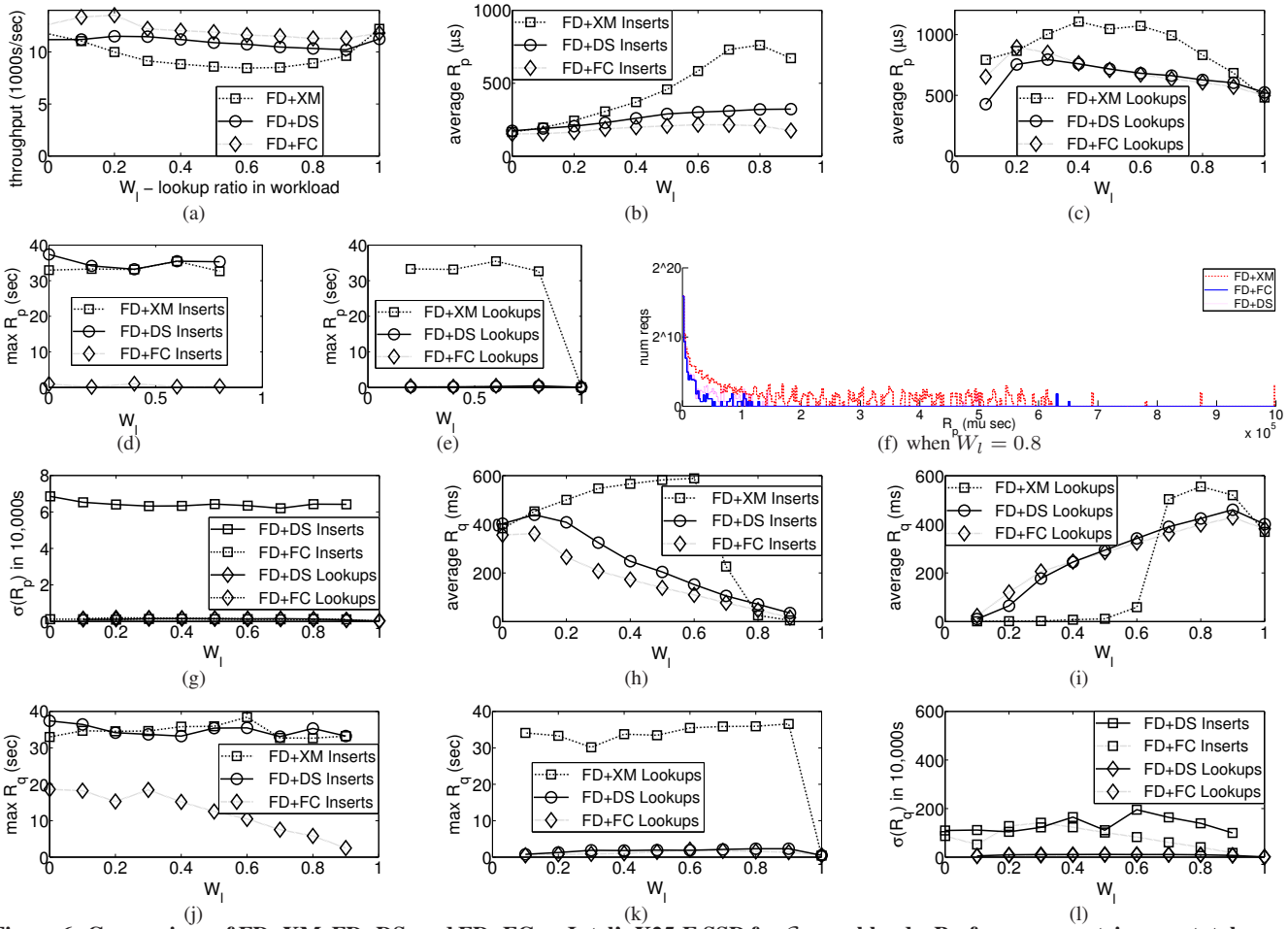


**Figure 6: Comparison of FD+XM, FD+DS, and FD+FC on Intel's X25-E SSD for $\mathcal{G}_R$ workloads. Performance metrics are: total completion time (a), average insertion $R_p$ (b), average lookup $R_p$ (c), worst-case insertion $R_p$ (d), worst-case lookup $R_p$ (e), distribution of lookup $R_p$'s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

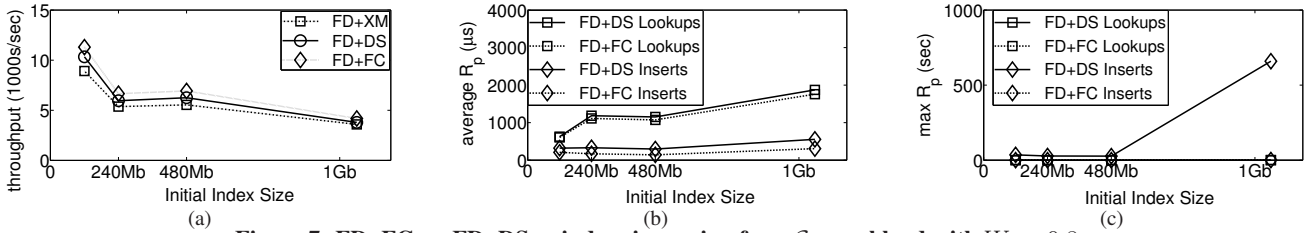


**Figure 7: FD+FC vs. FD+DS as index size varies, for a $\mathcal{G}_R$ workload with $W_l = 0.8$.**

better performance on $R_q$ 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, 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+XM have high worst-case insertion $R_q$. FD+FC's 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. Workload size is set to be equal to the initial index size. $W_l$ is set to 0.8. The worst-case insertion $R_p$ for FD+DS jumps to 659 seconds when the database size is 90M. But, FD+FC's 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 compares the schemes 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.3); 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

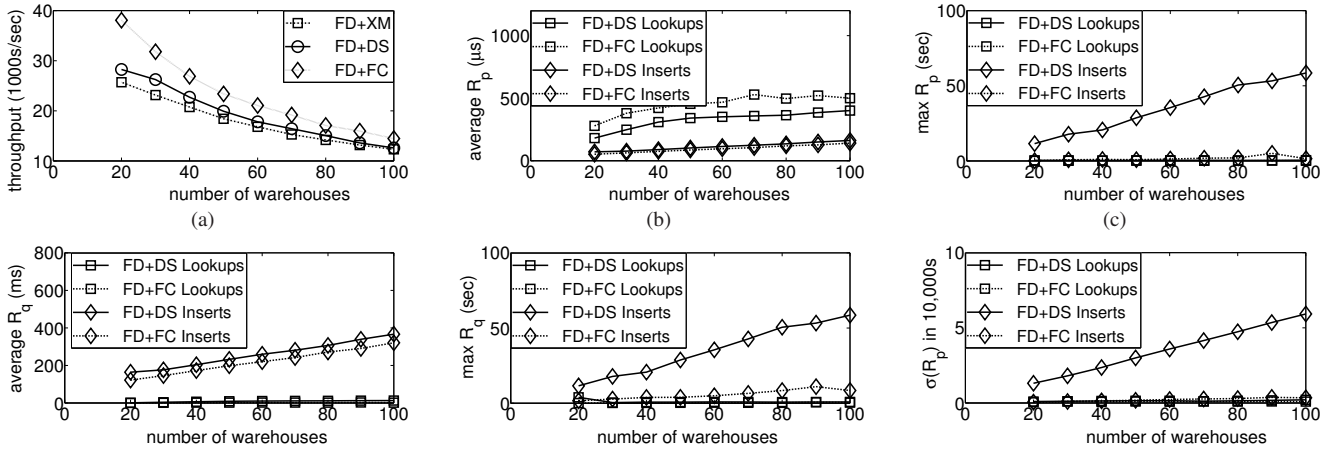


**Figure 8: Comparison of FD+XM, FD+DS, and FD+FC for $\mathcal{G}_T$ workloads on X25-E by varying number of warehouses, in terms of: total completion time (a), average insertion and lookup $R_p$ (b), worst-case insertion and lookup $R_p$ (c), average insertion and lookup $R_q$ (d), worst-case insertion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

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. Still, FD+FC has higher throughput, and as Figure 8(c) shows, has lower worst-case insertion $R_p$. FD+DS insertion $R_p$ values also exhibit high variance (see Figure 8(f)). FD+FC's average $R_q$ for update requests is as bad as FD+DS (Figure 8(d)), this is because the workload is very update intensive. But, FD+DS's worst case $R_q$ rises linearly (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 $L_0^{\text{new}}$ and $L_0^{\text{old}}$** FD+FC's memory sharing feature allows space freed from $L_0^{\text{old}}$ by a merge to be added to $L_0^{\text{new}}$ 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. FD+DS's worst insertion $R_p$ is close to that of 'FD+FC w/o MS,' 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.

# 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* [21], *FlashDB* [16], 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* [13] are exact-match key-value stores based on hashing. FD-tree [12] 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* [18] 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* [17] 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* [15] is conceptually similar to LSM-tree, but targets temporal databases. The *bLSM-tree* [19] 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 [9] is similar to LSM-tree, but maintains multiple B-trees at each level. The *T1SM* [10] 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.

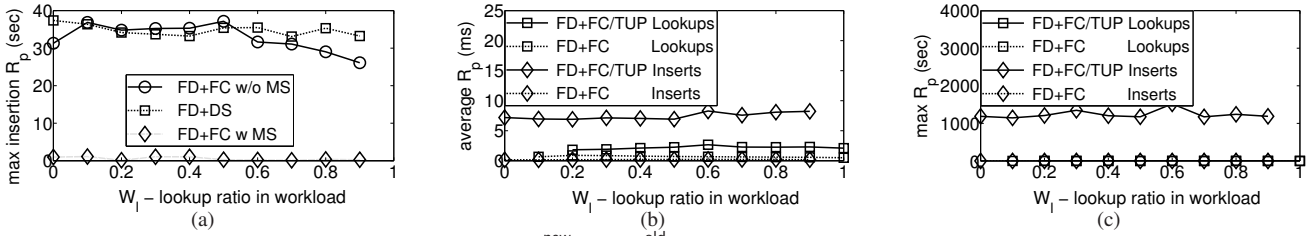


**Figure 9: (a) Benefits of memory sharing between $L_0^{\text{new}}$ and $L_0^{\text{old}}$. (b, c) Performance of FD+FC vs. FD+FC/TUP that shows the importance of FD+FC's block-wise data movement.**

# 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. *Proc. 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 *Proc. 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 *Proc. 2011 ACM SIGMOD*, pages 25–36, Athens, Greece, June 2011.

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

[9] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti. Incremental organization for data recording and warehousing. In *Proc. 1997 VLDB*, pages 16–25, Athens, Greece, 1997.

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

[11] 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 *Proc. 2008 ACM SIGMOD*, pages 1075–1086, Vancouver, Canada, June 2008.

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

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

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

[15] 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.

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

[17] 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.

[18] 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. *Proc. of the VLDB Endowment*, 5(4):286–297, 2011.

[19] R. Sears and R. Ramakrishnan. blsm: A general purpose log-structured merge tree. In *Proc. 2012 ACM SIGMOD*, pages 217–228, Scottsdale, Arizona, USA, June 2012.

[20] R. Thonangi, S. Babu, and J. Yang. A practical concurrent index for solid-state drives. Technical report, Duke University, 2012. `http://www.cs.duke.edu/dbgroup/papers/2012-ThonangiBabuYang-ssd_tree_cc.pdf`.

[21] 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 (When to reclaim a block; Section 4.4)** 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 $k_1 \leq k < k_2$, 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.2 (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.3 (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). They 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.