Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices

Devesh Agrawal, Deepak Ganesan, Ramesh Sitaraman, Yanlei Diao, and Shashi Singh
{*dagrawal,dganesan,ramesh,yanlei,shashi*}*@cs.umass.edu*,
Department of Computer Science
University of Massachusetts Amherst

Flash memories are in ubiquitous use for storage on sensor nodes, mobile devices, and enterprise servers. However, they present significant challenges in designing tree indexes due to their fundamentally different read and write characteristics in comparison to magnetic disks.

In this paper, we present the Lazy-Adaptive Tree (LA-Tree), a novel index structure that is designed to improve performance by minimizing accesses to flash. The LA-tree has three key features : 1) it amortizes the cost of node reads and writes by performing update operations in a lazy manner using cascaded buffers, 2) it dynamically adapts buffer sizes to workload using an online algorithm, which we prove to be optimal under the cost model for raw NAND flashes, and 3) it optimizes index parameters, memory management, and storage reclamation to address flash constraints. Our performance results on raw NAND flashes show that the LA-Tree achieves 2× to 12× gains over the *best* of alternate schemes across a range of workloads and memory constraints. Initial results on SSDs are also promising, with 3× to 6× gains in most cases.

1. INTRODUCTION

Flash memories are finding widespread use for storage on embedded sensors, mobile platforms, and enterprise servers. Their ubiquitous use is due to a myriad of benefits: small size, low cost, low power consumption, and high random read performance. These advantages have enabled flash to be employed in a variety of roles that require local storage and indexing. Examples include in-network sensor data storage and querying [5], embedded databases on mobile devices [8], and enterprise databases using SSDs [18, 19]. Thus, the design of flash-optimized index structures is an important problem in flash-based database systems.

A key challenge in designing flash-optimized index structures is that, as a storage medium, flash has fundamentally different read/write characteristics from other non-volatile media such as magnetic disks. In particular, flash updates need to be preceded by an erase operation, which is particularly cumbersome since the unit of erase spans multiple pages. Modifying a node in its original location is prohibitively expensive: it requires copying all valid pages from the erase block, then erasing the block, and finally copying the valid pages and updated page back to the block. A trivial solution to this problem is provided by Flash Translation Layers (FTLs [14]) which write the modified node to a new unwritten flash page, and hide this low-level page movement by exposing logical page numbers to the index. This is inefficient as every update results in node-rewrites and it increases the cost of reclaiming invalidated pages.

Many recent studies have addressed the problem of designing flash-optimized index structures (e.g. [21, 23]). At their core, these approaches employ *page-level write optimizations*. Instead of performing an expensive re-write of the original flash page, deltas to the page are stored in a separate location on flash. When a page needs to be read back to memory, both the base page, as well as delta pages are retrieved, and the page is re-constructed.

We argue that page-level optimizations are fundamentally ill-suited to the design of tree indexes on flash. Page-level optimizations reduce the cost for node writes, but end up greatly increasing the cost of node reads due to the overhead of reading delta pages. Despite reads often being cheaper than writes on flash, the fact that node reads are performed at every level of the tree for both update and lookup operations dramatically increases the overhead of page-level optimizations. In the case of a tree, the overhead of reading delta pages at each level can make a delta-based scheme more expensive even than an FTL-based approach.

In this paper, we present the LA-Tree, a fundamentally different tree index for flash that is designed around lazy updation techniques rather than delta-based techniques. Our design of the LA-Tree has the following key features:

Cascaded Buffers: Unlike a traditional tree index, update operations on an LA-Tree are not immediately propagated down the tree. Instead, flash-resident buffers are attached to various levels of the tree, and updates drip down from one buffer to another in a cascading manner, eventually propagating to the leaf nodes. The key benefit of such lazy updates is that it optimizes *both node reads and node writes* during update operations, thereby overcoming a central drawback of a delta-based approach.

Adaptive Buffering: While a lazy approach is efficient for update operations, it is inefficient for lookup operations where an *immediate* response needs to be provided to the user. Performing such immediate lookups on a lazy tree involves scanning large flash-resident buffers, which is prohibitively expensive. The LA-Tree addresses this drawback by using a novel online algorithm that intelligently adapts the buffer size to the observed workload. We prove that this algorithm is optimal under the cost model for raw NAND flashes. It enables the LA-Tree to transition seamlessly from being a fully lazy data structure with large buffers for an update-intensive workload, to being a fully eager data structure with no buffers for a lookup-intensive workload, while choosing an ideal buffer size for any intermediate workload. The LA-Tree's adaptive capability is a key distinction from prior work on lazy trees for bulk loading [1, 2].

Memory and Reclamation optimizations: A holistic implementation of the LA-Tree requires addressing two important practical considerations. First, available memory resources need to be carefully used to optimize performance. Besides the traditional use of memory for caching

to reduce page reads and writes, memory is also needed for write-coalescing to reduce fragmentation of buffers on flash. Reducing fragmentation makes buffer scans and reclamation cheaper. Thus, the LA-Tree appropriately partitions memory for both caching and write-coalescing to improve performance. Second, tree nodes and buffers need to be carefully laid out on flash to optimize reclamation when flash space runs low. The large size of a flash erase block makes reclamation expensive since valid pages need to be copied out to a different location before reclaiming a block. The LA-Tree facilitates reclamation since it uses *flash-friendly buffers* to reduce node updates. Buffers are sequentially written and can be emptied instead of being copied, which make them cheap to maintain and reclaim.

Generality to flash devices: While our work primarily focuses on raw NAND flashes, we empirically evaluate the LA-Tree's performance over "packaged" flash devices such as Solid State Drives (SSDs) . These devices are designed using raw NAND flashes, but have an on-board hardware controller that hides flash complexity and provides a disk-like abstraction to the operating system. Our evaluation shows that the LA-Tree can provide impressive benefits for these packaged flash devices as well.

In summary, we present a lazy, adaptive, flash-optimized tree index structure that provides a fundamentally new approach to optimizing tree indexes over flash devices. Our technical contributions include:

- We describe the LA-Tree, a novel index structure for flash memories that combines the benefits of lazy updates with immediate querying. It dynamically adapts to the workload using an optimal online algorithm.
- We provide a holistic system implementation that optimizes 1) LA-Tree parameters such as node size and buffer placement based on flash characteristics, 2) memory management across nodes and buffers to maximize benefits of write-coalescing and caching, and 3) storage reclamation by exploiting flash-friendly buffers that are easy to maintain and reclaim.
- We compare the LA-Tree against a spectrum of state-of-the-art approaches for flash-optimized indexes [18, 21, 23]. Our performance results on raw NAND flashes show that the LA-Tree achieves 2× to 12× gains over the *best* of alternate schemes across a range of workloads and memory constraints. Initial results with SSDs are also promising, with 3× to 6× gains in most cases.

2. A LAZY ADAPTIVE TREE INDEX

In this section, we propose a new index structure called Lazy Adaptive (LA) Tree. The main idea underlying an LA-Tree is to avoid the high cost of updating a flash-based tree index using *lazy updates*, and to do so in an *adaptive* manner to achieve good performance for both updates and lookups. To this end, an LA-Tree employes two main techniques:

Cascaded buffers. An LA-Tree attaches buffers to nodes at multiple levels of the tree. Each buffer contains update operations to be performed on the node and its descendants. At an appropriate time, all elements in the buffer are pushed down in a batch to the buffers at the next level. Thus, the LA-Tree performs multiple updates all at once, sharing node accesses from the root to the leaves among the updates, thereby yielding a better amortized cost for each update.

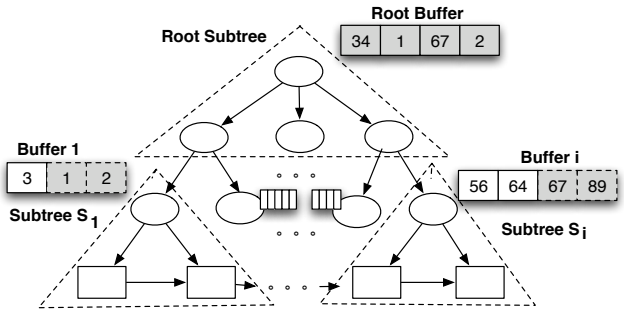


Figure 1: An LA-Tree with non-leaf nodes (ovals), leaf nodes (rectangles), and cascaded buffers at every alternate level. The shaded boxes show the result of emptying the root buffer into next level buffers.

Parameter	Symbol	Value
Memory size	M	given by application
Node size (in num. of elements)	**F**	variable
Num. of elements in the tree	N	variable
Height of the tree	H	$\approx \lceil \log_F \frac{N}{F} \rceil + 1$
Subtree height for buffering	**K**	variable
Effective buffer size (in num. of elements)	B_i	$\leq U < M$, variable, subtree-specific

Table 1: Notation used in the paper. Bold letters denote the parameters of an LA-Tree that are tunable.

An online adaptive algorithm. The idea of lazy batched updates is reminiscent of earlier work on buffering for bulk updating tree indexes [1, 2]. However, buffering introduces an inherent tension between update performance and lookup performance. Hence, those approaches achieve a low update cost at the expense of a high lookup cost. In contrast, the LA-Tree adapts dynamically to arbitrary workloads using an optimal online algorithm, thereby offering efficient support for both updates and lookups.

2.1 Overview of LA-Trees

Like a B+ tree, an LA-Tree is a balanced tree with fanout F. A key distinction from a B+ tree is that the LA-Tree attaches a **flash-resident buffer** to the nodes at every K^{th} level of the search directory, starting at the root. The buffer is used to batch the update (insert and delete) operations to be performed on this node and its descendants. Figure 1 illustrates the buffers placed at every alternate level from the root, i.e., K=2. In an LA-Tree, the term "subtree" is used to refer to a fragment of the tree that starts with a buffer-attached node and ends above the next level of buffer-attached nodes, so a subtree has exactly K levels. Finally, we use B_i to denote the effective buffer size (in number of entries) of the i^{th} subtree. B_i cannot exceed the predefined limit U (chosen to be a fraction of the memory size M), but often has a smaller value because it is dynamically determined based on the current workload seen by the subtree. Table 1 summarizes the notation used in this paper.

We next describe the basic operations on an LA-Tree. Figure 2 and Figure 3 show the sketches of these operations.

Lazy insertions and deletions (Figures 2(a) and (b)). An update operation in an LA-Tree is lazy: the new entry for insert or delete is simply appended to the buffer of the root node. Then the *AppendToBuffer* routine checks if the buffer needs to be emptied by calling the ADAPT algorithm. If the

(a) Routine Insert(R, E)
Input: Root of the tree R, Entry E = (k: key value, *rid*: record id)

1. Let BUF = buffer of the root node, invoke AppendToBuffer(BUF, E, i).

(b) Routine Delete(R, E)
Input: Root of the tree R, Entry E = (k: key value, *rid*: record id)

1. Let BUF = buffer of the root node, invoke AppendToBuffer(BUF, E, d).

(c) Routine Search(R, q)
Input : Root of the tree R, Predicate q
Output: Set of entries A that satisfy q

1. Top down search in the tree. For each non-leaf node on the path, if the node has a buffer BUF, invoke SearchBuffer(BUF, q) $\rightarrow A$.
2. At the leaf node N, invoke SearchNode(N) $\rightarrow A$.
3. If q is a range predicate, for each next leaf node N, SearchNode(N) $\rightarrow A$, for each buffer BUF at an ancestor node of N, if BUF has not been scanned, SearchBuffer(BUF, q) $\rightarrow A$.
4. If A contains deletion entries, apply deletions to collapse content of A.

Figure 2: Top level routines of an LA-tree (sketches).

decision is yes (which occurs infrequently), buffer emptying is invoked to push all its entries in a batch to the buffers at lower levels of the tree, where buffer emptying can occur recursively until the entries finally reach the leaf nodes.

Immediate lookups (Figure 2(c)). When an LA-Tree receives a lookup request, it searches the tree top-down. A main distinction from a B+ tree is that if a node on the root-to-leaf path has an attached buffer, the LA-Tree also scans the buffer for updates that have not reached the leaf nodes. A unique feature of our work is that while performing a buffer scan (Figure 3(b)), the ADAPT algorithm checks whether emptying the buffer will improve performance based on the workload seen so far. If buffer emptying is deemed beneficial, all entries in the buffer are pushed down to the next level; otherwise, the buffer is scanned to find matches of the search predicate. This procedure continues until reaching the leaf node where more matches may be collected.

For a range predicate, the LA-Tree may sequentially access other leaf nodes. While scanning each leaf node, the LA-Tree also searches the buffers attached to the ancestors of the leaf node if they have not been visited before (This is another distinction between the LA-Tree and a B+ Tree). This additional buffer search cost however is limited: repeated scans of those buffers will soon make ADAPT decide to empty the buffers. The collected matches are finally condensed by sorting them based on the timestamp and key value and applying deletions, if present.

Dynamic decisions about buffer emptying. ADAPT is an online algorithm that is the central intelligence module of the LA-Tree. It sees a sequence of update and lookup requests for each buffer and controls the effective buffer size by deciding when to empty it. Update requests do not trigger a buffer empty unless the buffer overflows its predefined limit U. For lookups, ADAPT records both the cost of scanning the buffer and the estimated cost of emptying it. It then weighs the one-time cost of emptying the buffer at this point versus the savings that subsequent lookups can receive due to this empty. When the savings outweigh the emptying cost, ADAPT decides to empty the buffer. ADAPT is explained in detail in §2.2.

(a) Routine AppendToBuffer(BUF, E, *mode*)
Input: Buffer BUF, Entry E, Mode *mode* = i (insertion) or d (deletion)

1. Add an entry E' = (E, *mode*, *timestamp*) to BUF.
2. If ADAPT decides to empty the buffer, invoke EmptyBuffer(BUF).

(b) Routine SearchBuffer(BUF, q)
Input: Buffer BUF, Predicate q
Output: Set of entries A that satisfy q

1. If ADAPT decides to empty the buffer, invoke EmptyBuffer(BUF) and piggyback the search for q. Else, directly search BUF for q. Return result.

(c) Routine EmptyBuffer(BUF)
Input: Buffer BUF be emptied

If buffer BUF is attached to an intermediate subtree
1. Sort the buffer.
2. For buffer entries in sorted order, search through the subtree, insert a set of entries $\{E\}$ to each next-level buffer BUF' by invoking AppendToBuffer(BUF', E).

If buffer BUF is attached to a bottom subtree
3. Sort the buffer.
4. For buffer entries in sorted order, search through the subtree, find a set of buffer entries $\{E\}$ for each leaf node N.
5. Merge $\{E\}$ with N, apply deletions if present.
6. If N overflows, split N, add index entries $\{Ei\}$ to the parent node. If needed, recursively split ancestor nodes and their attached buffers.
7. If N underflows, borrow entries from neighbors. If still underflow, merge leaf nodes, remove index entries $\{Ed\}$ from the parent node. If needed, recursively apply redistribution/merging to ancestor nodes and possibly merge their attached buffers.

Figure 3: Buffer routines of an LA-tree (sketches).

Buffer emptying operations (Figure 3(c)). Upon deciding to empty the buffer, ADAPT invokes the buffer emptying routine. Buffer emptying at an intermediate subtree consists of steps 1-2: sort the buffer, and distribute all its entries in sorted order to the next level buffers after searching the subtree. For example, Figure 1 shows the root buffer being emptied. The buffer is first read into memory, sorted, and its keys distributed to the next level buffers.

Buffer emptying at a leaf subtree (steps 3-7) also begins with sorting the buffer. The buffer entries are then distributed in sorted order to the leaf nodes. From left to right, the leaf nodes are processed one at a time: for each leaf N, the received buffer entries are merged into N by performing insertions and deletions as specified in those entries. If N overflows, it is split into two or more nodes. Such splits can propagate to the ancestor nodes; non-empty buffers attached to these nodes are also split accordingly. Due to the mix of insertions and deletions, N seldom underflows. But when it happens, N is first adjusted by borrowing entries from a neighboring node. Then, if needed it is merged with a neighboring node just like in a B+ tree. In the rare event of a merge, the adjustment of the parent and ancestor nodes is the same as the B+ tree, except that the buffers of two merged nodes are also merged.

Buffer emptying can be implemented so that the sorting step (steps 1, 3) can be performed by scanning the buffer stored on flash in a *single* pass, and the distribution steps (steps 2, 4) performed by reading the subtree from flash in a *single* pass. We discuss this optimization in §2.3.

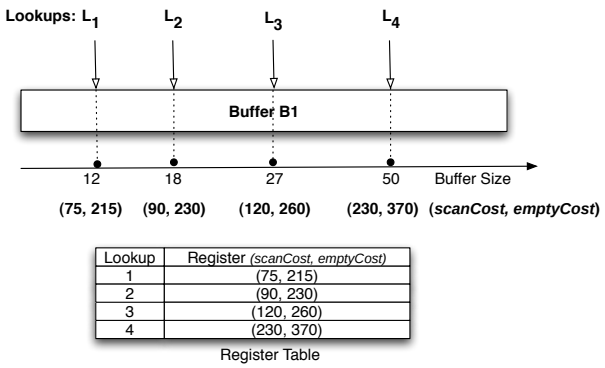


Figure 4: For the example sequence of four lookups, ADAPT empties the buffer at the fourth lookup L_4.

Finally, it is important to note that the root to leaf path taken by each update operation in the LA-Tree is identical to a B+ Tree. The only change is the increased "travel time" due to buffering at intermediate nodes along the path.

2.2 Adaptive Buffering

Our motivation for adaptive buffering is based on the observation that a fixed buffer size does not work well for arbitrary workloads. To illustrate this, consider a root buffer having a fixed size B. When the buffer has accumulated B entries, it is emptied by distributing the entries through the top subtree. As described above, the emptying cost involves one scan of the buffer and subtree, followed by writing the elements to lower-level buffers. Thus, the amortized emptying cost (particularly, the amortized subtree read cost) over all update operations is inversely proportional to B. In contrast, during a lookup all entries in the buffer must be scanned, incurring a cost proportional to B. Hence, buffering introduces an inherent tension between update performance and lookup performance.

In this section, we present the ADAPT algorithm that dynamically determines buffer sizes based on the workload. Such dynamic decisions are made for each buffer individually because different subtrees can see different workloads, depending on the key distribution. As a sequence of update and lookup requests arrive at the buffer, ADAPT decides in an online manner whether to empty the buffer so that the cost of the entire sequence of operations is minimized. In the following, we first describe the key idea of ADAPT, then present the algorithm, and finally prove that ADAPT is an optimal online algorithm.

Key Insight. We illustrate the key insight behind ADAPT using Figure 4, which shows a single buffer that processes a sequence of lookup operations. Each lookup, denoted by L_i, sees a certain buffer size b_i, and hence pays a certain buffer scan cost s_i. For example, the scan cost of lookup L_1 is 75, while that of lookup L_2 is 90. For each L_i, ADAPT decides whether to empty the buffer at that point. It bases this decision on the estimated cost of emptying the buffer at each lookup. We use e_i to denote the estimated cost of emptying the buffer at lookup L_i, which increases with the buffer size b_i. For example, the estimated emptying cost at lookup L_1 is 215, and at lookup L_2 is 230.

Consider the benefit of emptying at lookup L_1 in Figure 4. Each lookup L_j $(j > 1)$ after L_1 would avoid scanning the b_1 portion of the buffer, thereby accruing savings s_1. Each of the three lookups after L_1 saves s_1. Hence the benefit of emptying at lookup L_1, denoted by payoff p_1, is given by $p_1 = 3 \cdot s_1 = 225$. This benefit outweighs the estimated cost of emptying e_1 and hence it is beneficial to empty at lookup L_1. We can generalize from this that it is always beneficial to empty at lookup L_i, if the future savings p_i outweigh the estimated cost of emptying e_i. However, since ADAPT is an online algorithm, it does not know the future workload and hence cannot predict the future payoff p_i. ADAPT avoids this handicap by reasoning in hindsight, as explained below.

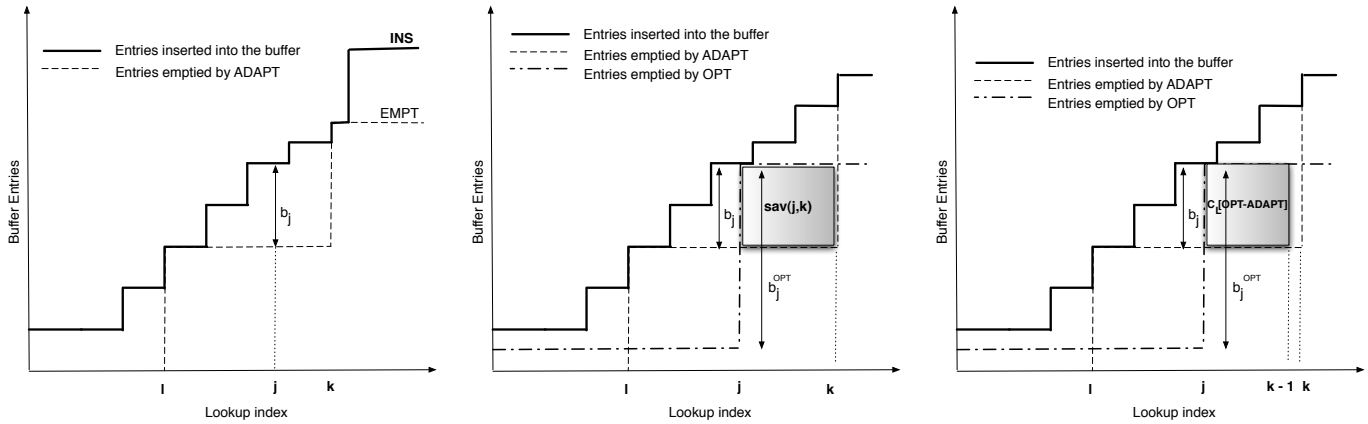
The most important concept of ADAPT is ***savings in hindsight***. Consider a lookup L_j. The savings in hindsight of a prior lookup L_i $(i < j)$ is the potential savings obtained from emptying at lookup L_i. There are $(j - i)$ lookups between L_i and L_j, and each saves the buffer scan cost s_i. Hence, the *savings in hindsight* of L_i, denoted by $sav(i, j)$, is given as $sav(i, j) = (j - i) \cdot s_i$. ADAPT empties the buffer at lookup L_j, if the savings in hindsight of *any* prior lookup L_i outweigh the corresponding estimated emptying cost e_i. More formally, ADAPT empties the buffer at lookup L_j if $\exists_{i<j} sav(i, j) > e_i$. In order to compute this, ADAPT maintains a book-keeping structure called *Register* R_i for each old lookup L_i. As shown in the figure, R_i has two fields: *scanCost* stores the scan cost s_i, and *emptyCost* stores the estimated emptying cost e_i. In this example, ADAPT first empties at lookup L_4 as the savings in hindsight of lookup L_1, given by $sav(1, 4) = (4-1) \cdot R_1.scanCost = 225$, exceed the corresponding emptying cost $R_1.emptyCost = 215$.

Algorithm 1 ADAPT(operationType)

```
1: if (operationType = lookup) then
2:    j ← j + 1 {j is the lookup index}
3:    {Check if any old lookup triggers emptying}
4:    for all lookups L_i such that (i < j) do
5:       savings ← (j − i) · R_i.scanCost {sav(i, j)}
6:       if (savings ≥ R_i.emptyCost) then
7:          Return EMPTY
8:       end if
9:    end for
10:   Create new R_j = (emptyCost, scanCost)
11: else if (bufferSize > U) then
12:    Return EMPTY
13: end if
14: Return NOT_EMPTY
```

The Algorithm. Having explained the key insight, we now present the complete algorithm as shown in Algorithm 1. To handle *lookup* operations, ADAPT performs two steps (lines 2 to 10): First, it checks if the *savings in hindsight* of *any* old lookup L_i exceed its estimated emptying cost. If so, it decides to empty the buffer. Then, ADAPT creates a new register to save the estimated emptying and scan costs for the current lookup. Handling *update* operations is trivial: ADAPT simply empties the buffer when it exceeds the limit U (line 11).

We now discuss several practical considerations about the algorithm. One is to optimize the number of registers stored. For most real workloads, ADAPT requires only a few (e.g., 3) registers for each buffer due to three reasons. First, ADAPT is only concerned with the lookups since the last buffer empty. Hence, we erase a buffer's registers each time it is emptied. Second, we only keep one register for each group of consecutive lookups. This is because the first lookup in the group will always be the first to trigger an empty, hence obviating the need to keep state about the subsequent lookups in the group. Finally, the number of registers required is low both under update-heavy workloads, where there are few lookups,



(a) Buffer entries inserted and emptied (b) Proof of Lemma 1 (c) Proof of Lemma 4

Figure 5: LA-Tree proof of optimality. The solid staircase line plots the number of buffer entries inserted into the buffer versus the lookup id. The dashed lines plot the number of buffer entries emptied from the buffer.

and under lookup-heavy workloads, for which the buffer size is already reduced to a small value.

Another important issue is online estimation of buffer scan and empty costs. We discuss this aspect in §3 where a thorough cost analysis is presented.

2.2.1 Proof of Optimality

We now prove that ADAPT is an optimal online algorithm under the cost model for raw NAND flashes. We compare ADAPT's total cost with the cost of an omniscient offline optimal algorithm OPT that knows the entire sequence of operations in advance. Hence, OPT represents a lower bound on ADAPT's cost. We first show that ADAPT is 2-competitive, i.e. its cost is no more than twice that of OPT for any input sequence. We then show that there is no deterministic online algorithm having a competitive ratio smaller than 2, thereby proving that ADAPT is an optimal online algorithm.

Our cost model is abstracted from the detailed model presented in §3. Lookups and buffer empty costs are linear in terms of the number of buffer entries. Each lookup operation costs $\rho \cdot b_i$ and each buffer empty costs $\delta + \gamma \cdot b_i$, where ρ, δ and γ are constants and b_i is the current buffer size (in the number of entries). We ignore the cost of update operations as they are already included in the buffer emptying cost of the parent subtree. We note that this model only holds for byte addressable flashes like raw NAND, but §6.4 shows how it may serve as an initial approximation for block based devices like SSDs.

Now we characterize the cost of an entire sequence of operations on a buffer using this simplified cost model. Let L be the entire sequence of lookups seen by a buffer. For any given algorithm ALG, let $L_E[\text{ALG}]$ denote the subset of lookups at which a buffer empty is triggered. Therefore the buffer emptying cost incurred by ALG can be written as:

$$C^E[\text{ALG}] = \sum_{i \in L_E[\text{ALG}]} (\delta + \gamma \cdot b_i) = \delta \cdot |L_E[\text{ALG}]| + \gamma \cdot \sum_{i \in L_E} b_i$$

where we use $C_f^E[\text{ALG}]$ and $C_v^E[\text{ALG}]$ to denote the first and second terms respectively. They represent the *fixed* and *variable* components of the total emptying cost $C^E[\text{ALG}]$ respectively. The cost of the remaining lookups $L - L_E[\text{ALG}]$ is:

$$C^L[\text{ALG}] = \rho \cdot \sum_{i \in L - L_E[\text{ALG}]} b_i$$

. Therefore the total cost incurred by algorithm ALG for the entire lookup sequence L is simply $C[\text{ALG}] = C^E[\text{ALG}] + C^L[\text{ALG}] = C_f^E[\text{ALG}] + C_v^E[\text{ALG}] + C^L[\text{ALG}]$.

Figure 5(a) illustrates an arbitrary mix of updates and lookups on a buffer. The solid staircase line INS plots the total number of update entries inserted into the buffer versus each lookup id. The vertical segments of the staircase represent consecutive updates, while horizontal segments denote consecutive lookups. The dashed line $EMPT$ shows the total number of entries emptied from the buffer versus each lookup id. A vertical jump in this line represents a buffer empty operation. For example, the figure shows two buffer empties at lookups L_i and L_k respectively. The instantaneous buffer size is given by the difference between the INS and $EMPT$ curves. For instance, the buffer size b_j at lookup L_j in the figure is equal to $INS(j) - EMPT(j)$.

In the following, Lemma 1 proves an important property about ADAPT and Lemmas 2 to 4 individually bound each of the above three cost components. Theorem 1 then uses these bounds to derive the competitive ratio for ADAPT. Finally, Theorem 2 proves that ADAPT achieves the optimal competitive ratio.

Lemma 1: OPT has to empty at least once between any two consecutive empties of ADAPT.

Proof by contradiction: Let the two consecutive empties of ADAPT be at lookups L_i and L_k. Assume that there is no empty of OPT in between. Figure 5(b) illustrates such a scenario. The two dashed lines in the figure depict the emptying operations of ADAPT and OPT. The buffer emptying at L_k can be triggered due to two cases, which we consider in turn:

Case 1: Empty at L_k was caused by the buffer getting full. Thus, there should be at least U update operations between L_i and L_k to fill up the buffer. Let OPT contain b_i^{OPT} entries at lookup L_i, therefore it should contain at least $b_k^{\text{OPT}} = b_k^{\text{OPT}} + U$ entries at lookup L_k. Since this exceeds the maximum buffer size U, OPT should have emptied at least once before L_k. This contradicts our original assumption

and thereby proves the lemma.

Case 2: Empty at L_k was not caused by the buffer getting full. Therefore, there must be some prior lookup L_j whose *savings in hindsight* $sav(j,k)$ outweighs the cost of emptying the buffer e_j. By our cost model, we get $sav(j,k) = q \cdot \rho \cdot b_j$, where there are $q = (k-j+1)$ lookups between L_j and L_k. $sav(j,k)$ is thus equal to ρ times the area of the shaded rectangle in Figure 5(b). Since the buffer was emptied at L_k, $sav(j,k) >= e_j$ holds. Where e_j is the cost of emptying the buffer at L_j, given by $e_j = \delta + \gamma \cdot b_j$. The buffer size of OPT b_j^{OPT} at lookup L_j is necessarily greater than the buffer size of ADAPT b_j, as ADAPT emptied after OPT. Therefore we can extend the above inequality to conclude that $q \cdot \rho \geq \frac{\delta}{b_j} + \gamma > \frac{\delta}{b_j^{\text{OPT}}} + \gamma$. Hence, OPT can $q \cdot \rho \cdot b_j^{\text{OPT}} - (\delta + \gamma \cdot b_j^{\text{OPT}})$ units of cost (a positive amount) by choosing to empty at L_j. This contradicts our assumption that OPT is optimal. Therefore, OPT has to empty at least once before L_k, thereby proving the lemma.

Lemma 2: $C_f^E[\text{ADAPT}] \leq C_f^E[\text{OPT}]$.

Proof: By Lemma 1, we can show that there should be at least one empty of OPT between any two consecutive empties of ADAPT. Thus, OPT empties at least as many times as ADAPT and hence $|L_E[\text{ADAPT}]| \leq |L_E[\text{OPT}]|$. Since $C_f^E[alg] = \delta \cdot |L_E[alg]|$, the lemma immediately follows.

Lemma 3 : $C_v^E[\text{ADAPT}] \leq C_v^E[\text{OPT}] + \gamma \cdot U$.

Proof: Consider the total number of entries emptied by both OPT and ADAPT. The difference between the number of entries emptied by OPT and ADAPT can be at most U, because the buffer size of both algorithms is upper bounded by U. Each entry emptied contributes a constant cost γ to the variable cost component C_v^E. The lemma immediately follows.

Lemma 4 : $C^L[\text{ADAPT}] \leq C^L[\text{OPT}] + C_f^E[\text{OPT}] + C_v^E[\text{OPT}]$.

Proof: We can write $C^L[\text{ADAPT}]$ as $C^L[\text{ADAPT}] = C^L[\text{OPT}] + C^L[\text{ADAPT}-\text{OPT}] - C^L[\text{OPT}-\text{ADAPT}]$. Here $C^L[\text{ADAPT}-\text{OPT}]$ denotes the additional lookup cost incurred by ADAPT that is not incurred by OPT. Similarly $C^L[\text{OPT} - \text{ADAPT}]$ denotes the additional lookup cost incurred by OPT that is not incurred by ADAPT. Thus we can see that $C^L[\text{ADAPT}] \leq C^L[\text{OPT}] + C^L[\text{ADAPT} - \text{OPT}]$. In the following we prove that $C^L[\text{ADAPT} - \text{OPT}] \leq C_f^E[\text{OPT}] + C_v^E[\text{OPT}]$, which proves the lemma.

Consider Figure 5(c). Similar to Figure 5(b), ADAPT empties consecutively at lookups L_i and L_k. From Lemma 1, we know that OPT should empty at least once in this interval. The figure shows one such empty at lookup L_j. As shown in the figure, the instantaneous buffer size of ADAPT is greater than that of OPT during the interval $[L_j, L_{k-1}]$. Lookup L_k is not included in this interval as at that point ADAPT's buffer size is also reduced to zero. Thus, after lookup L_k the buffer size of ADAPT is b_j greater than that of OPT. Therefore, the additional lookup cost incurred by ADAPT $C^L[\text{ADAPT} - \text{OPT}]$ is given by $\rho \cdot (k-j) \cdot b_j$. It is equal to ρ times the area of the shaded rectangle in Figure 5(c) and is hence identical to $sav(j, k-1)$. The shaded area in Figure 5(c) highlights this region. Since ADAPT did not empty at L_{k-1}, $sav(j,k-1)$ has to be strictly less than the corresponding emptying cost of ADAPT e_j. Further, e_j is less than the emptying cost suffered by OPT e_j^{OPT} as ADAPT has a smaller buffer size at L_j. This gives us the inequality $C^L[\text{ADAPT}-\text{OPT}] = sav(j,k-1) < e_j \leq e_j^{\text{OPT}}$. Therefore, the $C^L[\text{ADAPT}-\text{OPT}]$ cost in the interval (L_i, L_k) is bounded by the emptying cost suffered by OPT in that interval. We can show this holds for every interval, thereby proving that the total $C^L[\text{ADAPT} - \text{OPT}]$ cost is upper bounded by the total emptying cost of OPT given by $C_v^E[\text{OPT}] + C_f^E[\text{OPT}]$.

Theorem 1: ADAPT is 2-competitive.

Proof: Using the three lemmas and the corollary, we have: $C_f^E[\text{ADAPT}] + C_v^E[\text{ADAPT}] + C^L[\text{ADAPT}] \leq C_f^E[\text{OPT}] + C_v^E[\text{OPT}] + C^L[\text{OPT}] + C_f^E[\text{OPT}] + C_v^E[\text{OPT}] + \gamma \cdot U$. So, $cost[\text{ADAPT}] \leq 2 \cdot cost[\text{OPT}] - C^L[\text{OPT}] + \gamma \cdot U \leq 2 \cdot cost[\text{OPT}] + \gamma \cdot U$. The last term $\gamma \cdot U$ is a constant, hence it follows that ADAPT is 2-competitive [16] □

Theorem 2 : ADAPT is an optimal online algorithm.

Proof: *Ski-rental* [16] is a special case of our problem, where we only get *one* update and a sequence of lookups afterwards. Since there is no deterministic online algorithm for ski-rental having a competitive ratio lower than 2, the same holds for our more general problem. Hence ADAPT is an optimal online algorithm □

2.3 Buffer Emptying in a Single Scan

Having discussed the key insights behind adaptive buffering, we now discuss how buffer emptying can be performed in a *single* scan of the flash-resident buffers and subtree. Buffer emptying involves two main steps: (1) sorting the buffer, and (2) distributing the sorted buffer entries to the lower level buffers.

The buffer can be easily sorted in a single scan if it fits in memory. This is usually the case given our adaptive buffer size control. In rare cases, the buffer size can exceed available memory when multiple large buffers cascade into a single lower level buffer. To handle this, we adopt the buffer emptying method in [1]. Here, a buffer is broken into two parts: the batch of entries Y distributed in the last empty from its parent subtree, and the entries X in the buffer before Y. Since Y was distributed from above, it is already sorted. X was in the buffer within the maximal capacity, thus $X < M$. So, we can read X, sort it in memory, and merge it with Y. As we merge, we can distribute them into the buffers at the next level. This way, the entire buffer $X + Y$ can be sorted in a single pass.

The sorted buffer entries are then distributed to the lower level buffers by reading the subtree once, one path at a time. Thus, a single subtree scan suffices to empty the buffer.

3. LA-TREE COST ANALYSIS

The ADAPT algorithm relies on accurate estimates of buffer scan and empty costs to make informed decisions. In this section, we present a cost analysis of buffer operations. The result of the analysis supports the linear cost model of buffer emptying used in our optimality proof (§2.2). It also allows us to perform online cost estimation for use by ADAPT.

We exclusively focus on raw NAND flashes in our analysis. In contrast to SSDs, these flashes are byte addressable and their IO cost function has two main components: there is a fixed cost of accessing the page and a per-byte cost of reading or writing each byte [9]. The fixed access cost is much smaller than the disk counterpart due to the absence of any rotational or seek delay. (Table 2 in §6 shows the cost function for one such raw NAND flash.)

In the following discussion, we use a_r and a_w to denote the costs of reading and writing a tree node, and b_r and b_w

for the costs of reading and writing a buffer entry. These costs depend on how the nodes and buffers are laid out on flash, which is described in detail in §4. The following two implementation aspects are relevant to our cost analysis.

First, nodes have a fixed size and do not straddle page boundaries. Therefore, the node costs a_r and a_w are constants, which can be calculated by plugging the node size in the flash cost function. Second, buffers are of variable size and can span multiple pages. We also pack entries belonging to different buffers into a single flash page to minimize page writes. Thus, a buffer can be fragmented across many pages. This impacts the cost of reading the buffer as it incurs multiple fixed access costs. We use the term *fragmentation-overhead*, f, to denote the overhead of accessing multiple buffer pages for each buffer read operation, normalized across all entries in the buffer. Intuitively, f increases as a buffer is spread over more pages. We assume for simplicity that f remains relatively stable and approximate it to be a constant. (In practice, f can be estimated online.) Thus, we can model the total cost of reading each buffer entry as $b_r = R_b + f$, where R_b denotes the constant per-byte cost of reading a buffer entry. In contrast, fragmentation of buffers does not affect the cost of writing a buffer. This is due to each buffer page being completely packed. Hence each written buffer entry incurs a fixed amortized overhead of accessing its page. Therefore, b_w remains a constant that can be readily calculated from the (fixed) number of buffer entries contained in a page.

Non-leaf subtree: The cost of emptying a non-leaf subtree, having S nodes and B buffer entries, includes:

- *buffer read*: cost of reading the buffer once, $b_r \cdot B$,
- *subtree read*: cost of reading the subtree once, $a_r \cdot S$,
- *buffer writes*: cost of adding B entries to the buffers of the child subtrees, $b_w \cdot B$,

So, the total cost is $C_m^E = a_r \cdot S + (b_r + b_w) \cdot B$. As it can be seen, this cost is linear in the buffer size B.

Leaf subtree: Buffer emptying at a leaf subtree is similar to above except that the third step is replaced by writing buffer entries to the leaf nodes and adjusting the tree bottom-up, if needed. For tree adjustment, we apply the prior result on node rebalancing operations for the general family of *a-b* trees [12] which includes the LA-Tree. In our case, the total number of node rebalancing operations, including splits, merges, and redistributions, for a buffer of size B_n is bounded by $H + \frac{8 \cdot B_n}{F}$, where H is the height, F is the fanout of the tree and each node is required to be at least a quarter full.

Node rebalancing during the bottom up tree adjustment can also trigger buffer rebalancing operations as discussed in §2.1. However, we note that buffer rebalancing rarely happens due to two reasons. First, when we are emptying the buffer of a leaf subtree, the first non-empty buffer that we can encounter bottom-up is at the level $H - 2K$ (K is the subtree height) or higher. If K=2, this means levels $H - 4$ or higher. Since all node splits and merges originate at the bottom of the tree, they rarely propagate that high. Second, higher-levels buffers see more lookups than lower-level buffers so they are emptied more frequently. Therefore, the cost of buffer rebalancing is much smaller than that of node rebalancing (e.g., less than 1% in practice) and hence is omitted in the analysis below. To summarize, the buffer empty cost for a leaf subtree, having S nodes and B buffer entries, includes:

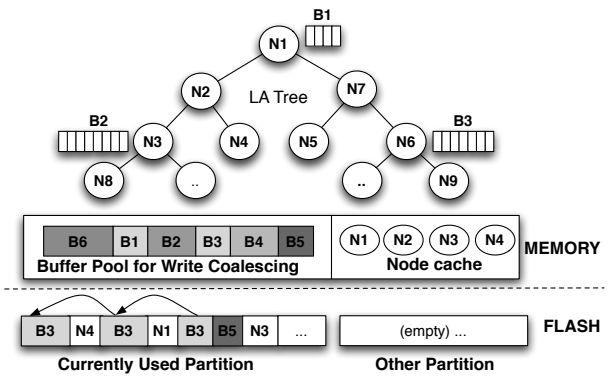


Figure 6: Implementation of the LA-Tree.

- *buffer read*: cost of reading the buffer once, $b_r \cdot B$,
- *subtree read*: cost of reading the subtree once, $a_r \cdot S$,
- *node writes with tree adjustment*: $a_w \cdot (L + H + \frac{8 \cdot B}{F})$,

where L is the number of leaf nodes in the subtree. So, the buffer empty cost at a leaf subtree is $C_n^E = (a_r \cdot S + a_w \cdot L + a_w \cdot H) + (b_r + 8\frac{a_w}{F}) \cdot B$. Again, this cost is linear in the buffer size B.

Online statistics: In-order to estimate the above costs at runtime, our implementation collects the following statistics for each subtree: (1) the number of nodes S, (2) the number of buffer entries B, (3) the number of leaf nodes L for each leaf subtree, and (4) the *fragmentation-overhead* term f, which allows us to approximate the cost of reading a buffer entry b_r as discussed above.

Hence, this cost analysis enables ADAPT to accurately estimate the buffer costs to make intelligent emptying decisions.

4. OPTIMIZING LA-TREE FOR FLASH

Having discussed the the key ideas behind LA-Tree, we now describe its efficient implementation on flash. In particular, we develop system-level optimizations including tuning index parameters, optimizing memory usage for write-coalescing, and optimizing storage layout for reclamation. Our implementation, shown in Figure 6, is general and works on both raw NAND and SSDs flash devices.

4.1 Tuning LA-Tree Parameters

We now briefly describe how to tune LA-Tree parameters, namely, the fanout and the subtree height to further improve the overall performance. We verify these intuitions empirically in §6.1.

Fanout: The fanout determines the node size and hence the cost of reading and writing a node. Being byte addressable, raw NAND flashes favor small nodes as this reduces the cost of reading and writing the index nodes. However, an extremely small node size is detrimental since it increases the tree height and more buffers need to be accessed on each root-to-leaf path. In contrast to raw NAND flashes, SSDs are block devices and hence favor page sized nodes. Thus, the LA-Tree chooses a relatively small node size for raw NAND flashes while using page sized nodes for SSDs.

Subtree height: The subtree height K also affects the overall cost. For a very small sub-tree height (e.g. K=1), the LA-Tree attaches a buffer to each node. This increases the total number of buffers as well as the number of buffers

on a particular root-to-leaf path, thereby increasing buffer access costs. When K takes a large value, e.g., 4, the subtree size is relatively large, causing a high buffer empty cost. As a result, the ADAPT algorithm is less likely to empty the buffer, leading to increased lookup cost. Therefore, we set K to a relatively small value between 2 and 3.

4.2 Optimizing Memory Usage

Memory is needed for optimizing both write-coalescing of buffers and caching of LA-Tree nodes, as shown in Figure 6. Write-coalescing of buffers is particularly important since adaptive buffering can result in small buffers being emptied, triggering writes in units that are considerably smaller than a flash page. This makes the buffer spread over more pages, which increases its *fragmentation-overhead* f, thereby increasing its read cost. Since buffers are scanned for both lookups and buffer empties, packing buffers is important.

Therefore, we allocate most of the memory in the system for write-coalescing of buffers, and only use a small amount of memory for the nodes (25% of total memory in our implementation). This is because 1) the cost of scanning buffers during lookups and empties is typically the dominant factor in overall cost, hence optimizing packing is critical to performance of the LA-Tree, and 2) lazy updates already optimizes node reads and writes, hence the additional improvement by node caching is limited.

The memory allocated to the buffers and nodes is managed in different ways. Memory allocated to buffers is managed as a buffer pool. To improve packing, the eviction policy for the buffer pool is "highest-packed buffer fragment" *i.e.* the buffer fragment having the largest number of buffer entries in it is flushed. For example, in Figure 6, buffer $B6$ is the highest packed, hence it is chosen as the next buffer to be evicted. Nodes are managed as a standard node cache with LRU eviction policy to evict old nodes.

4.3 Optimizing Storage and Reclamation

Storage reclamation can be a complex and expensive operation on flash due to large erase units and the high cost of performing an in-place update. Node updates on NAND flashes need to be handled by a translation table that avoids an in-place update by re-writing the page to an unwritten location on flash and hiding this physical movement of the page from the index. Such out-of-place rewrites use flash space quickly, and trigger more frequent reclamation. In turn, reclamation incurs the high cost of copying valid node pages from blocks that need to be erased.

The LA-Tree facilitates efficient reclamation in two ways. First, it triggers fewer node updates due to cascaded buffers, thereby invalidating fewer pages. Second, it reduces node updates by using *flash-friendly buffers* that are easy to store and reclaim. Buffers are sequentially written data structures and trigger no in-place updates. They also trigger no data copying upon reclamation since they can be emptied instead of being copied.

Storage and reclamation of the LA-Tree for raw NAND flashes is done as follows. The LA-Tree is allocated two equal-sized flash partitions, as shown in Figure 6. The two partitions are distinct from the other flash partitions used by the database to store data tables and other metadata. When one of these two partitions fills up, it is reclaimed in following two steps: First, we empty all the buffers and recursively copy the valid nodes to the other partition. Finally, the erase blocks in the first partitions are erased.

Pages are written sequentially within a partition. Each page can either be a buffer page that contains one or more buffer fragments, or a node page that contains one or more nodes. Fragments belonging to the same buffer are reverse linked to enable traversal, whereas pages allocated to nodes are managed using a Node Table (equivalent of an index-specific FTL). Every node update results in an out-of-place re-write to the head of the partition, and an update of the Node Table to reflect the new location[1]. Note that sequential writes on flash has the auxiliary benefit of improving wear-levelling, since writes can be distributed evenly across erase blocks.

In contrast to raw NAND flashes, SSDs expose a disk-like block interface and have an on-board controller which handles reclamation and in-place updates on flash. This obviates the need for writing nodes out-of-place and hence we do not use a separate in-memory Node Table nor consider reclamation for SSDs.

5. IMPLEMENTATION OF OTHER SCHEMES

We compare against three state-of-art flash-optimized indexing schemes — BFTL, FlashDB, and IPL. All techniques are "delta-based" schemes and encode modifications to nodes in the form of "deltas" (e.g. an insert operation or update of child pointer). Each node is stored as a base node and a sequence of deltas on flash. All delta schemes require an in-memory table to keep track of the list of delta pages for each node [21, 23]. Note that all three schemes are just node level optimizations in that they only change the way a node is laid out on flash, while retaining the familiar B+ Tree index structure on top. To ensure a fair comparison, we use the same underlying B+ Tree implementation as used by the LA-Tree. Each scheme was further optimized by using compact semantic-deltas [21] and by tweaking index level parameters like fanout. We briefly discuss the specifics of each scheme below:

BFTL: BFTL [23] is designed to maximize write-coalescing of deltas for a B+ tree node. BFTL buffers the generated deltas into an in-memory delta pool. This delta pool is flushed after every N insertions to the tree. While flushing the delta pool, BFTL packs deltas belonging to the same node into one (or more) consecutive flash pages. Unused memory by the delta pool is employed for caching nodes. We set $N = 60$ based on the value recommended in [23].

FlashDB: FlashDB [21] reduces the overhead of reading a long chain of deltas for each lookup by using an online algorithm, called *Switch-Mode*, to adaptively switch between maintaining a node as a list of deltas (*Log-Mode*), or as a single page on flash (*Disk-Mode*). Intuitively, this algorithm maintains frequently read nodes, like non-leaf nodes in Disk-Mode, whereas frequently updated nodes like leaf nodes are left in the Log-Mode. This adaptive technique allows FlashDB to get the benefits of deltas without suffering the excessively high cost of node reconstruction.

IPL: IPL [18] is designed to minimize scattered writes of deltas across erase blocks, thereby optimizing storage reclamation. While IPL was proposed as a general storage layer for flash-based databases, we consider the case where a tree index uses IPL as the storage layer. IPL partitions pages in

[1] A Node Table can also be maintained efficiently on flash if needed [14], but we do not address this case in this paper.

			Write	Read
Toshiba SLC Small Block 128MB 512B page, 16KB erase block [22]	Energy	Fixed	$24\mu J$	$4\mu J$
		Per-byte	$0.09\mu J$	$0.1\mu J$
	Latency	Fixed	$274\mu s$	$69\mu s$
		Per-byte	$1.5\mu s$	$1.7\mu s$

Table 2: Flash energy/latency numbers for raw NAND flash. (Numbers are measured using a custom-fabricated raw NAND flash board for the Mica2 sensor mote.)

an erase block into an equal number of node and delta pages. Whenever a node page is dirtied and needs to be written to flash, IPL writes the deltas to one of the delta pages within the same erase block. When all the delta pages in an erase block fills up, the node and delta pages are merged, and re-written to a new erase block. The original erase block can now be erased and reused.

6. PERFORMANCE EVALUATION

In this section we compare the LA-Tree against the standard B+ tree, and flash optimized indexes like BFTL, FlashDB and IPL. We first evaluate the LA-Tree over raw NAND flash, while presenting our initial results with enterprise grade SSDs at the end. We start by detailing the flash and traces used in our experiments.

We built a raw NAND flash emulator for use in Linux, since we were unable to find a suitable device driver that would allow us to directly interface the underlying flash chip. The emulator was populated with exhaustive measurements from a custom-designed Mica2 sensor board with a 128MB Toshiba raw NAND flash chip (TC58DVG02A1FT00 [22]).

This Toshiba flash also obeys a linear cost function as described in §3. Table 2 summarizes the energy and latency cost functions for this flash. We use W_{fixed} and W_{byte} to denote the fixed and per-byte cost of writing to flash, and correspondingly R_{fixed} and R_{byte} for the cost of reading from flash. We can now write the parameters R_b, b_w, a_r, and a_w introduced in §3 in terms of the flash cost function as follows: R_b is the per-byte cost component of reading an 8 byte buffer entry given by $R_b = 8 \cdot R_{byte}$. b_w is the cost of writing a buffer entry and can be written as $b_w = \frac{8}{512} \cdot W_{fixed} + 8 \cdot W_{byte}$, where the first term represents the amortized fixed cost of accessing a 512 byte flash pages. The cost of reading and writing a node, a_r and a_w, depend on the node size. For an X byte node, a_r is readily obtained from the flash cost function as $R_{fixed} + X \cdot R_{byte}$, while a_w is calculated similar to b_w as $a_w = \frac{X}{512} \cdot W_{fixed} + X \cdot W_{byte}$.

Traces: Our traces consist of a sequence of updates and lookups, with each operation specifying a 4-byte integer key and an optional 4-byte record-id. For most experiments, the workload consists of a random mix of updates and lookups with a given lookup-to-update-ratio (LTU), which captures the likelihood of getting a lookup over an update. To avoid initial tree construction from influencing the results, we apply the workload to a pre-constructed tree, containing 200K keys. The costs obtained were normalized by the workload size to report the cost per operation. We use the following workloads in our evaluation that offer a range of LTU ratios, and different correlations in the key distribution:

(1) Uniform: The keys are drawn uniformly and the number of updates is fixed to one million. We consider a range of LTUs from 10% to 1000%.

(2) Temperature: The keys are temperature readings taken at a local weather station. Similar to the Uniform trace, we fix the number of inserts to 800K while considering 10% and 200% LTU.

(3) Radar Velocity: This trace is obtained from a meteorological radar that scans the atmosphere every minute. Each radar scan consists of about 300K air-velocity readings, one for each atmospheric cell. Our discussion with meteorologists reveal that 1) they are interested in detecting high velocity air pockets within the area, and 2) queries are relatively infrequent. Hence, we use range queries of the form $[X, \infty)$, and a Poisson query arrival pattern with a mean rate of one query every two seconds. X is chosen by a power law (Pareto) distribution, such that it is within the top 20% of the velocity range with a 95% likelihood. This yields high selectivity, i.e. each query selects approximately 30K keys (1% of the keys inserted).

(4) TPC-C: The TPC-C trace is representative of an online transaction processing system (OLTP). We used a TPC-C implementation [13] to obtain the sequence of updates and lookups for each of the 10 indexes. We used four main traces – Customer, Order, Item and Neworder. The sizes of these workloads ranged from about 1 million to 4 million keys. The other six consisted of only lookups or only updates and are hence omitted in our study.

6.1 LA-Tree Benchmarks

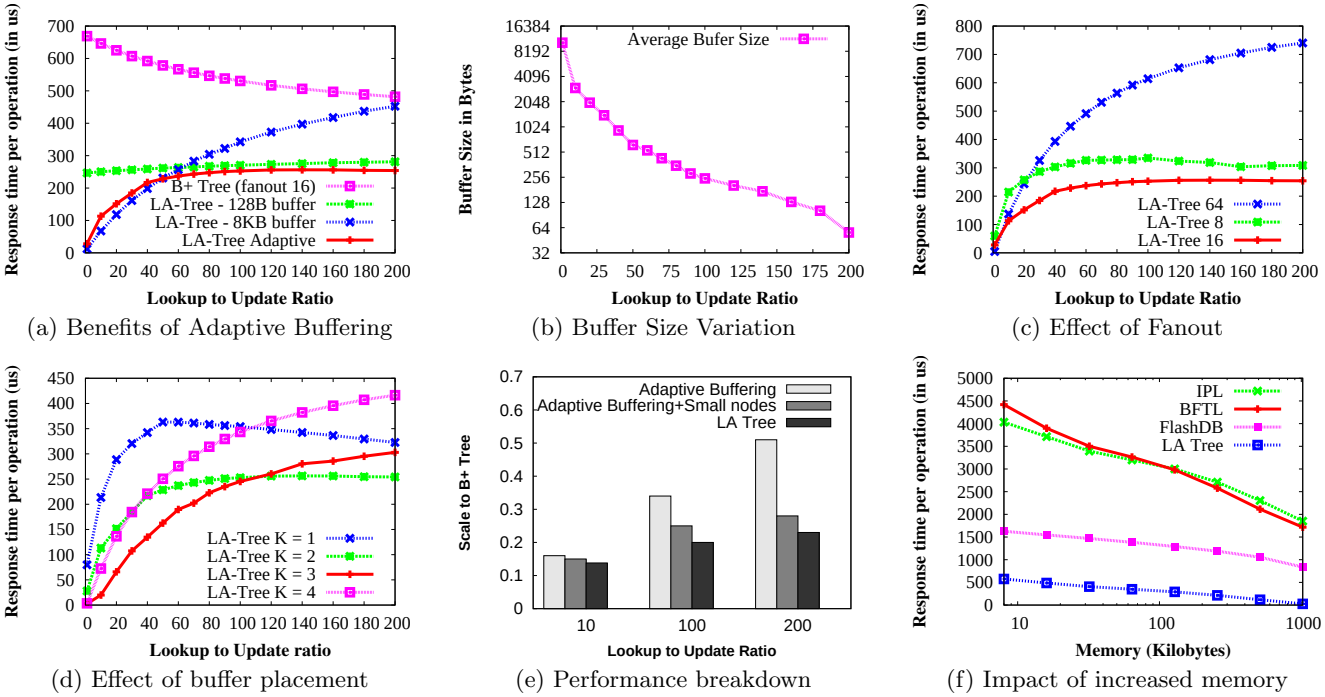
In this section we individually evaluate the core components of LA-Tree over a raw NAND flash. We first quantify the role of adaptive-buffering and then benchmark key optimizations like small-fanout and buffer-placement. All experiments were done with the Uniform workload consisting of a million updates. A majority of the updates (66% percent) were insertions, while the remaining updates were deletions. We assume 128 KB of RAM and do not consider storage reclamation costs to focus on index layer optimizations.

Evaluating Adaptive Buffer Size Control: A key benefit of the LA-Tree is its ability to adapt to varying workloads by dynamically adapting buffer sizes at different subtrees. We evaluate this adaptive capability by comparing it against three alternate buffer sizes: a) B+ Tree with the same fanout as LA-Tree (no-buffering), b) a large fixed size buffer (8KB), c) a small buffer (128 bytes).

Our response time results shown in Figure 7(a) reveal two key observations: First, the graph shows the benefits of adaptive buffering over fixed-size buffers. We see that large buffers are well suited for update-heavy workloads, whereas small buffers are well suited for lookup-heavy ones. On the other hand, the LA-Tree adjusts its buffer size to match any workload. This behavior is exemplified in Figure 7(b), which shows that the LA-Tree gradually reduces its buffer size with increasing LTU. Second, it shows that the B+ Tree is worse than all other schemes. For example, at 200% LTU the B+ Tree is 1.9× worse than the LA-Tree. This shows that lazy updates are useful even for lookup-heavy workloads.

The results of this experiment also indicate the benefits of adapting each subtree's buffer size independently to respond to workload variations within the tree. For example, the root buffer observes more frequent lookups compared to the leaf buffer. Therefore, ADAPT chooses an 8KB buffer for the leaf subtrees, while restricting the root buffer up to 1KB.

We also use the profiling data from this experiment to validate the cost estimation procedure outlined in §3. For



(a) Benefits of Adaptive Buffering (b) Buffer Size Variation (c) Effect of Fanout

(d) Effect of buffer placement (e) Performance breakdown (f) Impact of increased memory

Figure 7: Evaluation of LA-Tree over raw NAND flash.

instance, the estimated buffer costs in this experiment were found to be within 8% of the actual buffer costs.

Choosing Fanout and Sub-tree Height: Raw NAND flash enables fine-grained optimization of node fanout using sub-page reads and writes. Figure 7(c) shows that the LA-Tree favors a node size that is much smaller than a page. While small fanouts are generally superior, we find that decreasing the fanout below 16 results in somewhat worse performance since it increases the number of buffers. This hurts performance as it increases the fragmentation of buffers.

What levels should the LA-Tree place its buffers? Figure 7(d) shows the effect of placing buffers at every K^{th} level, where K is varied from 1 to 4. As can be seen, extreme choices of k ($K = 1$ and $K = 4$) perform poorly because of high node and buffer costs respectively. The choice between $K = 2$ and $K = 3$ depends on the workload. In an update-heavy workload, the adaptive algorithm chooses large buffers; hence, the dominant cost is due to buffer reads and writes. In this case, $K = 3$ is better since it results in fewer buffers and thus has lower buffer access costs. For a lookup-heavy workload, the LA-Tree operates mostly with small buffers, hence node costs dominate, which favors $K = 2$. We choose $K = 2$ since it works better across a wider range of LTU settings.

LA-Tree Breakdown: We now consider how adaptive buffering, fanout, and buffer placement contribute to the overall performance of the LA-Tree. We compare three versions of LA-Tree (a) LA-Tree with *adaptive buffering*, page-sized nodes and buffers at each level, (b) LA-Tree with adaptive buffering, *optimized fanout*, and buffers at every level, and (c) LA-Tree with adaptive buffering, small node size, and *buffers at alternate levels*. Since the impact of these mechanisms depend on the workload, we consider 10%, 100% and 200% LTU regimes.

Figure 7(e) shows the response time performance of the three schemes normalized to that of a regular B+ Tree with page-sized nodes. The maximum gains stem from our core mechanism of Adaptive Buffering, which boosts performance by more than 6×, 3×, and 2× for the three cases. The reduction in benefits with increasing lookups is expected since the LA-Tree operates more eagerly as the workload becomes more lookup-dominated. Optimizing the fanout improves performance by 6%, 26%, and 45% for the three workload settings. The gains are higher for lookup-heavy settings since node size has greater impact on overall performance when the dominant cost is node access for lookups. Finally, attaching buffers at alternate levels improves performance by an additional 8% to 20% since it reduces the number of buffer pages that need to be maintained in memory.

Summary: In conjunction, (a) Adaptive buffering (b) Small node size and (c) Buffer placement achieve 4.3× to 7.2× improvement over a traditional B+ Tree over a raw NAND flash. *Hence, we do not consider the B+ Tree in our subsequent raw NAND flash based experiments.*

6.2 Workload-driven Performance Study

In this section, we demonstrate that the LA-Tree offers significant performance gains across a range of workloads over a raw NAND flash. We use four workloads in our study — Uniform, Temperature, Radar-Velocity and TPC-C — and evaluate our performance against three flash-optimized tree indexes — FlashDB [21], BFTL [23], and IPL [18]. In all cases, we chose the best node size for each scheme to ensure fair comparison.

Table 3 shows the performance of difference schemes. As with the benchmarking experiments, each scheme was given 128KB of RAM. We consider two cases: (a) all schemes are given sufficiently large flash so that reclamation is not triggered, and (b) each scheme is given a fixed amount of flash such that storage reclamation is triggered.

	Uniform			Temperature		Radar	TPC-C			
	10% LTU	200% LTU	1000% LTU	10% LTU	200% LTU	Scan	Customer	Order	Item	New Order
FlashDB	1270	844	581	345	254	1385	550	111	66	22
BFTL	1322	2073	2386	899	1794	1478	862	69	78	19
IPL	2489	1702	1342	802	1418	2205	999	115	80	19
LA-Tree	113	254	253	71	21	163	119	31	46	14
Gain (resp. time)	**11.2×**	**3.3×**	**2.3×**	**4.8×**	**12×**	**8.5×**	**4.6×**	**2.2×**	**43%**	**36%**
Gain (Energy)	10.7×	2.9×	2.3×	4.6×	9×	8.2×	4.3×	2.1×	25%	28%
Gain (+reclamation)	12.5×	3.5×	2.5×	5×	12.5×	9.5×	5.8×	70%	23%	24%

Table 3: Workload driven evaluation over raw NAND flash. First four rows show *mean response time per operation* in μs for different schemes when reclamation is not considered. Last three rows show gains by LA-Tree over the *best* of alternate schemes: 1) in terms of response time, 2) in terms of energy, and 3) when storage reclamation is also taken into account.

	Uniform			Temperature		Radar	TPC-C			
	10% LTU	200% LTU	1000% LTU	10% LTU	200% LTU	Scan	Customer	Order	Item	New Order
B+ Tree	2190	1230	427	136	494	2926	1196	73	56	1.5
FlashDB	1487	1083	485	330	879	642	1877	64	55	1.4
BFTL	1463	1192	571	452	1146	1175	2661	108	61	1.4
IPL	2791	2289	908	2083	3198	6705	5312	544	245	135
LA-Tree	315	351	378	34	133	12	216	9	53	1.4
Gain (over best)	**4.6×**	**3×**	**1.1×**	**3.9×**	**3.7×**	**52×**	**5.5×**	**6.6×**	**3%**	**1×**
Gain (over worst)	**8.8×**	**6.5×**	**2.4×**	**60×**	**24×**	**540×**	**24×**	**56×**	**4.5**	**91×**

Table 4: Workload driven evaluation over SSD. First five rows show *mean response time per operation* in μs. Last two rows show gains by LA-Tree over the best and worst of alternate schemes, respectively.

Results without reclamation: We discuss LA-Tree's gains by considering each dataset in turn. As our energy gains are almost identical to our response time gains, we focus on response time gains in the discussion below.

Uniform: Table 3 shows that the LA-Tree outperforms other schemes across both lookup and update dominated workloads. We outperform our closest competitor FlashDB by 2.3× to 11.2×, with larger gains for update-heavy workloads when the LA-Tree can be lazier.

Temperature: This workload is highly correlated as temperature follows a periodic daily pattern. Therefore, node accesses have high locality and this improves cache locality of all schemes. This helps the LA-Tree as it reduces the working set of its node cache, and therefore frees up most of the memory for its buffer pool. In turn this significantly lowers the cost of batching. Table 3 shows that this boosts LA-Tree's gains over FlashDB to more than 4.8×.

Radar Air Velocity Scan: This trace is extremely update-heavy since each radar scan has a huge number of keys. The LA-Tree operates in a highly lazy manner given the highly update-dominated workload (0.01% LTU), thereby performing 8.5× - 13.5× better than other schemes. This experiment demonstrates that LA-Tree is efficient at answering range queries in addition to point queries.

TPC-C: The LA-Tree out-performs other approaches across all four traces. The maximum improvement is observed for the *Customer* trace, which is update-heavy (8% LTU) and has highly uncorrelated keys. This enables the LA-Tree to operate in a lazy manner, thereby achieving at least 4.6× improvement over alternate schemes. The other three traces are lookup-heavy and have highly regular access patterns. As a result all schemes perform well on them and the LA-Tree only achieves modest gains of upto 2.2×.

Results with reclamation: How do reclamation costs impact the above results? The last row in Table 3 shows the gains when reclamation is considered. Since traces were of different lengths, we used a flash size that was roughly three times the size of the tree constructed by that trace. The flash sizes given to each trace were: 15MB to the Uniform and Temperature traces, 3.5MB each to the TPC-C *Customer*, *Order*, and *NewOrder* traces, 35MB to the TPC-C *Item* trace and 100MB to the Radar trace. These settings trigger up to seven storage reclamation attempts for different cases. The reclamation overhead includes cost of movement of valid pages and cost of block erase operations.

The results show that the gains when including reclamation overhead are similar to gains when not considering this overhead. In fact, the LA-Tree has significantly lower reclamation overhead in most cases. Only IPL incurs marginally lower reclamation cost than the LA-Tree for a subset of the TPC-C traces, however its aggregate performance is considerably worse than all other schemes. Schemes other than IPL have a high reclamation overhead since they have long chains of deltas that need to be reclaimed. Thus, the LA-Tree can be efficiently reclaimed under limited flash space.

Summary: The LA-Tree and three other schemes were evaluated over diverse workloads on raw NAND flashes. The LA-Tree improves response time and energy consumption by up to 12× over the *best* of other schemes. Similar gains are obtained when storage reclamation overhead is considered.

6.3 Impact of Increasing Memory

We next explore the impact of memory on the performance benefits of the LA tree over raw NAND flash. We evaluate various schemes using the 100% LTU Uniform workload and and vary the memory between 8 KB to 1 MB.

Figure 7(f) shows how LA-Tree's performance scales with increasing memory. At very low memory, LA-Tree buffers are highly fragmented on flash due to limited memory for write-coalescing. This increases the cost of scanning buffers, which causes the ADAPT algorithm to empty more frequently. This reduces the benefits of buffering. As available memory increases, both write-coalescing and node caching hit rates improve. The ADAPT algorithm therefore empties less fre-

quently and allows buffers to grow larger, thereby improving performance. This trend is seen in the figure: LA-Tree's gains scale up with memory – from 3× at 8KB to roughly 33× at 1MB. Our results demonstrate that LA-Tree can adapt to both workload variations as well as memory variations using the same adaptive scheme.

Summary: The LA-Tree is 3× to 33× better than the best of other schemes across a range of memory settings.

Results with other NAND flashes: Our results until now are on the small block Toshiba raw NAND flash. We have also evaluated our performance over several large block raw NAND flashes. Our gains are even better in these cases. For instance, in the case of a Samsung X9F1208X0C flash, the LA-Tree's gains jump to 5× for the 1000% LTU Uniform workload and 23× for the 10% LTU Uniform workload.

6.4 LA-Tree over SSDs

	Random Writes	Random Reads	Seq. Writes	Seq. Reads
MSP-SATA7525 SSD [6]	8.3 ms	0.11 ms	0.07 ms	0.28 ms

Table 5: Response time benchmarks for SSD.

Until now we have evaluated LA-Tree over a raw NAND flash. We now provide initial results for LA-Tree's performance over enterprise grade SSDs. Unlike raw NAND flashes, these devices are not byte addressable and only allow a block I/O interface. As pointed out in §2.2.1, this impacts the optimality of the ADAPT algorithm as the buffer cost is no longer a linear function of the buffer size. Despite this key difference, we empirically demonstrate that LA-Tree continues to greatly outperform other schemes.

We use the 16GB MSP-SATA7525 [6] SSD attached to a Dual-Core Intel Pentium Linux server for all our SSD experiments. This SSD does not have enough on-disk caching and as a result it exhibits very poor random write performance. For example, as can be seen from benchmarks in Table 5, the random write cost is up to 30× the cost of other I/Os. Recent work suggests that this is due to the coarse granularity of FTLs employed in such SSDs, which exacerbate the number of read-modify-write operations [3, 4].

Unlike raw NAND flashes, SSDs are block devices and do not have a notion of fixed or per-byte costs. We use the same cost framework as described in §3, but change the parameters a_r, a_w, b_w, and b_r as follows: a_r is the cost of reading a page sized node and thus it is equal to the cost of one random read operation. Similarly, a_w is the cost of writing a page sized node and hence it is equal to the cost of one random write operation. The cost of writing an (8 byte) buffer entry b_w is the amortized cost of writing 8 bytes sequentially, since buffer entries are packed and written out sequentially. Therefore, we set b_w as $\frac{8}{512}^{th}$ of the sequential write cost. Reading a buffer can incur multiple random page reads. To model this, we estimate b_r in a manner similar to how the fragmentation-overhead f is estimated for the case of raw NAND flashes. That is, b_r is now the estimated cost of reading the buffer normalized by the number of entries.

We first consider the workloads used in §6.2 with 128KB of RAM. (We disabled the kernel's buffer cache to avoid impacting our results.) Table 4 shows that the LA-Tree significantly improves performance with more than an order of improvement over the *best* alternate scheme for some update-heavy workloads. LA-Tree continues to outperform other schemes due to two main reasons: First, adaptive buffering amortizes the buffer emptying cost over multiple insertions. In turn, this reduces expensive random write operations, which is a major source of inefficiency on SSDs (as shown in Table 5). Thus, the LA-Tree is able to leverage sequentially written buffers to reduce the number of node re-writes. In contrast, other schemes trigger many more node updates and therefore random write operations, which greatly increases their response time. Second, the ADAPT algorithm continues to keep the buffer scan cost in check by adapting the buffer size to the workload.

Scheme	10% LTU	200% LTU	1000% LTU
B+ Tree	2117 μs	1184 μs	395 μs
LA-Tree	308 μs	329 μs	361 μs
Gain	**6.8×**	**3.5×**	**10%**

Table 6: *Mean response time per operation* for a 2GB workload over SSD.

We now evaluate the LA-Tree against the B+ Tree over a large uniform workload consisting of 256 million insertions yielding an index of size more than 2GB. We gave 32MB of RAM to both schemes to ensure the same memory-to-workload ratio as used in §6.2. Table 6 demonstrates the benefits of LA-Tree for such enterprise grade settings. We see that the results are quite similar to the ones obtained with the smaller workload size, as shown in Table 4.

Summary: The adaptive buffering technique of LA-Tree provides significant benefits over SSDs too, with 3× to 6× performance gains in most cases.

7. EXTENSION TO GiST INDEXES

Finally, we briefly discuss how the main techniques of the LA-Tree can be extended to the GiST indexes [11]. The key insight in the extension is that cascaded buffers in the LA-Tree do not change the path that each update operation takes to traverse the tree. They only increases the "travel time" through buffering at intermediate levels of the tree. When a batch of updates are merged into a node, node rebalancing operations propagate bottom-up as before, possibly triggering similar operations on the buffers attached to those nodes.

As a proof of concept, we sketch the search and insert algorithms for GiST indexes that do not have key values from an ordered domain, e.g., the R-tree. A complete implementation is a focus of our future work.

Search in GiST is controlled by the *Consistent* method. In an unordered domain, a search may take multiple paths. The LA-Tree will strictly follow those paths in a search and scan buffers attached along the paths. However, once ADAPT observes repeated scans of the buffers, it will decide to empty those buffers sooner to adapt to the workload.

Insertion in GiST takes a single path in the tree, guided by the *Penalty* method. Once this path is determined, the LA-Tree can buffer updates at intermediate levels and decide when to empty using ADAPT as before. One distinction from insertion in the LA-Tree is that sorted distribution of buffer entries is not possible in the unordered domain. To still perform buffer emptying using a single scan of the buffer and the subtree, the LA-Tree needs to choose a small subtree size that can fit in memory. With the subtree loaded in memory, it can distribute the buffer entries through it one at a time.

This way, it still amortizes the subtree read cost over all the buffer entries. Since buffer entries can now go to lower-level buffers in an arbitrary order, write-coalescing of buffers will play a crucial role in avoiding excessive buffer fragmentation.

Finally, the only GiST method that needs to be extended is *PickSplit*, for which the LA-Tree will change from a two-way split to an n-way split. This is because emptying a leaf subtree could entail multiple index entries being merged into a leaf node which could split it into multiple nodes.

8. RELATED WORK

In this section, we survey related work that we have not discussed previously in this paper. (BFTL [23], FlashDB [21], and IPL [18] are discussed in detail in §5.)

Buffered Tree Structures: The idea of attaching buffers to nodes or subtrees has been explored in the context of efficient bulk index operations and bulk-loading of indexes [1, 2, 24]. However, these are specialized indexes and can only be used for specific workloads — [24] for lookup-dominated workloads and [1] for update dominated ones. The LA-Tree is fundamentally different in that we can dynamically adapt to any workload by means of our online algorithm. Furthermore, we optimize tree parameters, memory management, and reclamation to suit flash constraints and characteristics.

SSD in DBMS: SSDs offer fast sequential write and random read performance compared to disks, and consume lower power [7]. These advantages have led to much recent research on using SSDs: (1) as replacements for magnetic disks for enterprise databases [19], (2) as a write cache to improve latency [10], and (3) as extensions of disks along with intelligent mechanisms to split or migrate workloads between the two media [17].

From an indexing perspective, a key limitation of SSDs is their poor random write performance, which makes in-place node updates very expensive [3, 4]. FD-Trees is a recent approach that addresses this problem [20]. Every tree level in FD-Tree is treated as a sequentially written sorted log of keys and pointers, and can grow to a fixed size. When a log fills up, it is merged with the log at the level below and the merged log is rewritten to flash. This process continues down if necessary. Any rewrite of a log also triggers rewrites of *all* logs above it to maintain the tree structure. Thus, the FD-Tree avoids the high cost of random writes on SSDs by using many more sequential writes. Unlike the LA-Tree, the FD-Tree is designed solely for SSDs, and is impractical for raw NAND flashes that do not have a significant difference between sequential and random writes. Further, the LA-Tree can also be adapted to avoid random writes by using it in conjunction with our Node Table (described in §4.3). With this setup, we believe that the LA-Tree will perform better than the FD-Tree since it fundamentally reduces the total number of I/Os without incurring the overhead of many more sequential I/Os.

Raw NAND flash based Indexes: The μ-tree tries to minimize the impact of node rewrites on raw NAND flash in the absence of an FTL [15]. It does so by packing the entire path from the root to the leaf node into a single page on flash. This is orthogonal to the problem of minimizing the accesses to flash, which we address.

9. CONCLUSION

The use of flash devices has grown dramatically in the last five years, thereby increasing the importance of designing flash-optimized indexes. In this paper, we presented the design, analysis, and implementation of the LA-Tree, an optimized index structure for flash devices. We show that the core techniques underlying the design of the LA-Tree— lazy updates, adaptive buffering, and memory/reclamation optimizations — yield significant performance benefits over diverse flash devices ranging from raw NAND flashes to SSDs. The LA-Tree achieves 2× to 12× gains over the *best* of alternate schemes on NAND flashes. Initial results on SSDs are also promising, with 3× to 6× gains in most cases.

LA-Tree enables new research directions in designing flash-optimized database systems. One direction is designing a broader spectrum of SSD-optimized indexes in the GiST [11] family using similar approaches, and integrating them with a full-function database system. A second research direction is to explore techniques that maximize the end-to-end benefits of using flash storage in enterprise database servers.

10. REFERENCES

[1] L. Arge. The buffer tree: A Technique for Designing Batched External Data Structures. In *Algorithmica* 37(1):1-24, 2003.

[2] L. Arge, K. Hinrichs et al. Efficient bulk operations on dynamic R-trees. In *Algorithmica* 33(1):104-128, 2002.

[3] A. Birrell, M. Isard et al. A Design for High Performance Flash Disks. In *SIGOPS Operating system review*, 41(2), 2007.

[4] N. Agrawal, V. Prabhakaran et al. Design Tradeoffs for SSD Performance. In *USENIX 2008*.

[5] Y. Diao, D. Ganesan et al. Rethinking data management for storage-centric sensor networks. In *CIDR 2007*.

[6] MSP-SATA7525. `http://mtron.net/Upload_Data/Spec/ASIC/PRO/SATA/MSP-SATA7525_rev0.4.pdf`

[7] J. Gray, and B. Fitzgerald. Flash Disk Opportunity for Server Applications. In *ACM Queue 2008* 4(6).

[8] Enea polyhedra flashlite. `http://www.enea.com`.

[9] G. Mathur, P. Desnoyers et al. Ultra-Low Power Data Storage for Sensor Networks. In *IPSN-SPOTS*, 2006.

[10] G. Graefe. The Five-minute Rule Twenty Years Later, and How Flash Memory Changes the Rules. In *DAMON 2007*.

[11] J.M. Hellerstein, J.F. Naughton et al. Generalized Search Trees for Database Systems. In *VLDB*, 1995.

[12] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17, 1982.

[13] H-Store: A Next Generation OLTP DBMS. `http://db.cs.yale.edu/hstore/`.

[14] J.-U. Kang, H. Jo et al. A superblock-based flash translation layer for nand flash memory. In *EMSOFT 2006*.

[15] D. Kang, D. Jung et al. μ-tree: an ordered index structure for NAND flash memory. In *EMSOFT 2007*.

[16] A. Borodin and E. Y. Ran. Online computation and competitive analysis. Published by *Cambridge University Press*, 1998

[17] I. Koltsidas, S. D. Viglas. Flashing Up the Storage Layer. In *VLDB 2008*.

[18] S. Lee and B. Moon. Design of flash-based DBMS: an in-page logging approach. In *SIGMOD 2007*.

[19] S. Lee, B. Moon et al. A Case for Flash Memory SSD in Enterprise Database Applications. In *SIGMOD 2008*.

[20] Y. Li, B. He et al. Tree Indexing on Flash Disks. In *ICDE 2009*.

[21] S. Nath and A. Kansal. FlashDB: dynamic self-tuning database for NAND flash. In *IPSN 2007*.

[22] Toshiba TC58DVG02A1FT00 NAND flash chip datasheet. `http://toshiba.com/taec`, 2003.

[23] C. Wu, T. Wei, and L. P. Chang. An efficient B-tree layer implementation for flash-memory storage systems. *Trans. on Embedded Computing Systems*, 6(3), 2007.

[24] J. Zhou and K. A. Ross. Buffering accesses to memory-resident index structures. In *VLDB 2003*.