FAST: A Generic Framework for Flash-Aware Spatial Trees

Mohamed Sarwat[1,⋆], Mohamed F. Mokbel[1,⋆], Xun Zhou[1], and Suman Nath[2]

[1] Department of Computer Science and Engineering, University of Minnesota
[2] Microsoft Research
{sarwat,mokbel,xun}@cs.umn.edu, sumann@microsoft.com

Abstract. Spatial tree index structures are crucial components in spatial data management systems, designed with the implicit assumption that the underlying external memory storage is the conventional magnetic hard disk drives. This assumption is going to be invalid soon, as flash memory storage is increasingly adopted as the main storage media in mobile devices, digital cameras, embedded sensors, and notebooks. Though it is direct and simple to port existing spatial tree index structures on the flash memory storage, that direct approach does not consider the unique characteristics of flash memory, i.e., slow write operations, and erase-before-update property, which would result in a sub optimal performance. In this paper, we introduce FAST (i.e., Flash-Aware Spatial Trees) as a generic framework for flash-aware spatial tree index structures. FAST distinguishes itself from all previous attempts of flash memory indexing in two aspects: (1) FAST is a generic framework that can be applied to a wide class of data partitioning spatial tree structures including R-tree and its variants, and (2) FAST achieves both *efficiency* and *durability* of read and write flash operations through smart memory flushing and crash recovery techniques. Extensive experimental results, based on an actual implementation of FAST inside the GiST index structure in PostgreSQL, show that FAST achieves better performance than its competitors.

1 Introduction

Data partitioning spatial tree index structures are crucial components in spatial data management systems, as they are mainly used for efficient spatial data retrieval, hence boosting up query performance. The most common examples of such index structures include R-tree [7], with its variants [4,9,22,24]. Data partitioning spatial tree index structures are designed with the implicit assumption that the underlying external memory storage is the conventional magnetic hard disk drives, and thus has to account for the mechanical disk movement and its seek and rotational delay costs. This assumption is going to be invalid soon, as flash memory storage is expected to soon prevail in the storage market replacing the magnetic hard disks for many applications [6,21]. Flash memory storage is increasingly adopted as the main storage media in mobile devices and as a storage alternative in laptops, desktops, and enterprise class servers (e.g.,

⋆ The research of these authors is supported in part by the National Science Foundation under Grants IIS-0811998, IIS-0811935, CNS-0708604, IIS-0952977, by a Microsoft Research Gift, and by a seed grant from UMN DTC.

D. Pfoser et al. (Eds.): SSTD 2011, LNCS 6849, pp. 149–167, 2011.

in forms of SSDs) [3,13,15,18,23]. Recently, several data-intensive applications have started using custom flash cards (e.g., ReMix [11]) with large capacity and access to underlying raw flash chips. Such a popularity of flash is mainly due to its superior characteristics that include smaller size, lighter weight, lower power consumption, shock resistance, lower noise, and faster read performance [10,12,14,19].

Flash memory is block-oriented, i.e., pages are clustered into a set of blocks. Thus, it has fundamentally different characteristics, compared to the conventional page-oriented magnetic disks, especially for the write operations. First, write operations in flash are slower than read operations. Second, random writes are substantially slower than sequential writes. In devices that allow direct access to flash chips (e.g., ReMix [11]), a random write operation updates the contents of an already written part of the block, which requires an expensive block erase operation[1], followed by a sequential write operation on the erased block; an operation termed as *erase-before-update* [5,12]. SSDs, which emulate a disk-like interface with a *Flash Translation Layer* (FTL), also need to internally address flash's erase-before-update property with logging and garbage collection, and hence random writes, especially *small* random writes, are significantly slower than sequential writes in almost all SSDs [5].

Though it is direct and simple to port existing tree index structures (e.g., R-tree and B-tree) on FTL-equipped flash devices (e.g., SSDs), that direct approach does not consider the unique characteristics of flash memory and therefore would result in a suboptimal performance due to the random writes encountered by these index structures. To remedy this situation, several approaches have been proposed for flash-aware index structures that either focus on a specific index structure, and make it a flash-aware, e.g., flash-aware B-tree [20,26] and R-tree [25], or design brand new index structures specific to the flash storage [2,16,17].

Unfortunately, previous works on flash-aware search trees suffer from two major limitations. First, these trees are specialized—they are not flexible enough to support new data types or new ways of partitioning and searching data. For example, FlashDB [20], which is designed to be a B-Tree, does not support R-Tree functionalities. RFTL [25] is designed to work with R-tree, and does not support B-tree functionalities. Thus, if a system needs to support many applications with diverse data partitioning and searching requirements, it needs to have multiple tree data structures. The effort required to implement and maintain multiple such data structures is high.

Second, existing flash-aware designs often show trade-offs between efficiency and durability. Many designs sacrifice strict durability guarantee to achieve efficiency [16,17,20,25,26]. They buffer updates in memory and flush them in batches to amortize the cost of random writes. Such buffering poses the risk that in-memory updates may be lost if the system crashes. On the other hand, several designs achieve strict durability by writing (in a sequential log) all updates to flash [2]. However, this increases the cost of search for many log entries that need to be read from flash in order to access each tree node [20]. In summary, no existing flash-aware tree structure achieves both strict durability and efficiency.

[1] In a typical flash memory, the cost of *read*, *write*, and *erase* operations are 25, 200, and 1500 μs, respectively [3].

In this paper, we address the above two limitations by introducing FAST; a framework for Flash-Aware Spatial Tree index structures. FAST distinguishes itself from all previous flash-aware approaches in two main aspects: (1) Rather than focusing on a specific index structure or building a new index structure, FAST is a generic framework that can be applied to a wide variety of tree index structures, including B-tree, R-tree along with their variants. (2) FAST achieves both efficiency *and* durability in the same design. For efficiency, FAST buffers all the incoming updates in memory while employing an intelligent *flushing policy* that smartly evicts selected updates from memory to minimize the cost of writing to the flash storage. In the mean time, FAST guarantees durability by sequentially logging each in-memory update and by employing an efficient *crash recovery* technique.

FAST mainly has four modules, *update*, *search*, *flushing*, and *recovery*. The *update* module is responsible on buffering incoming tree updates in an in-memory data structure, while writing small entries sequentially in a designated flash-resident log file. The *search* module retrieves requested data from the flash storage and updates it with recent updates stored in memory, if any. The *flushing* module is triggered once the memory is full and is responsible on evicting flash blocks from memory to the flash storage to give space for incoming updates. Finally, the *recovery* module ensures the durability of in-memory updates in case of a system crash.

FAST is a generic system approach that neither changes the structure of spatial tree indexes it is applied to, nor changes the search, insert, delete, or update algorithms of these indexes. FAST only changes the way these algorithms reads, or updates the tree nodes in order to make the index structure flash-aware. We have implemented FAST within the GiST framework [8] inside PostgrSQL. As GiST is a generalized index structure, FAST can support any spatial tree index structure that GiST is supporting, including but not restricted to R-tree [7], R*-tree [4], SS-tree [24], and SR-tree [9], as well as B-tree and its variants. In summary, the contributions of this paper can be summarized as follows:

- We introduce FAST; a general framework that adapts existing spatial tree index structures to consider and exploit the unique properties of the flash memory storage.
- We show how to achieve efficiency and durability in the same design. For efficiency, we introduce a *flushing policy* that smartly selects parts of the main memory buffer to be flushed into the flash storage in a way that amortizes expensive random write operations. We also introduce a *crash recovery* technique that ensures the durability of update transactions in case of system crash.
- We give experimental evidence for generality, efficiency, and durability of FAST framework when applied to different data partitioning tree index structures.

The rest of the paper is organized as follows: Section 2 gives an overview of FAST along with its data structure. The four modules of FAST, namely, *update*, *search*, *flushing*, and *recovery* are discussed in Sections 3 to 6, respectively. Section 7 gives experimental results. Finally, Section 8 concludes the paper.

2 Fast System Overview

Figure 1 gives an overview of FAST. The original tree is stored on persistent flash memory storage while recent updates are stored in an in-memory buffer. Both parts need to be combined together to get the most recent version of the tree structure. FAST has four main modules, depicted in bold rectangles, namely, *update*, *search*, *flushing*, and *crash recovery*. FAST is optimized for both SSDs and raw flash devices. SSDs are the dominant flash device for large database applications. On the other hand, raw flash chips are dominant in embedded systems and custom flash cards (e.g., ReMix [11]), which are getting popular for data-intensive applications.

2.1 FAST Modules

In this section, we explain FAST system architecture, along with its four main modules; (1) Update, (2) Search, (3) Flushing, and (4) Crash recovery. The actions of these four modules are triggered through three main events, namely, *search queries*, *data updates*, and *system restart*.

Update Module. Similar to some of the previous research for indexing in flash memory, FAST buffers its recent updates in memory, and flushes them later, in bulk, to the persistent flash storage. However, FAST update module distinguishes itself from previous research in two main aspects: (1) FAST does not store the update operations in memory, instead, it stores the *results* of the update operations in memory, and (2) FAST ensures the *durability* of update operations by writing small log entries to the persistent storage. These log entries are written sequentially to the flash storage, i.e., very small overhead. Details of the update module will be discussed in Section 3.

Search Module. The search module in FAST answers point and range queries that can be imposed to the underlying tree structure. The main challenge in the search module is that the actual tree structure is split between the flash storage and the memory. Thus, the main responsibility of the search module is to construct the recent image of the tree by integrating the stored tree in flash with the tree updates in memory that did not make it to the flash storage yet. Details of the search module will be discussed in Section 4.

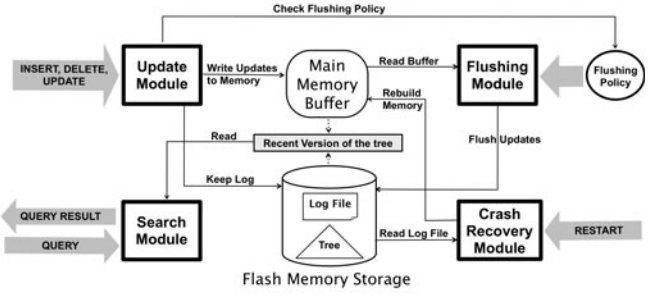


Fig. 1. Tree Modifications Table

Flushing Module. As the memory resource is limited, it will be filled up with the recent tree updates. In this case, FAST triggers its flushing module that employs a *flushing*

policy to smartly select some of the in-memory updates and write them, in bulk, into the flash storage. Previous research in flash indexing flush their in-memory updates or log file entries by writing *all* the memory or log updates once to the flash storage. In contrast, the flushing module in FAST distinguishes itself from previous techniques in two main aspects: (1) FAST uses a *flushing policy* that smartly selects *some* of the updates from memory to be flushed to the flash storage in a way that amortizes the expensive cost of the block erase operation over a large set of random write operations, and (2) FAST logs the flushing process using a single log entry written sequentially on the flash storage. Details of the flushing module will be discussed in Section 5.

Crash Recovery Module. FAST employs a crash recovery module to ensure the durability of update operations. This is a crucial module in FAST, as only because of this module, we are able to have our updates in memory, and not to worry about any data losses. This is in contrast to previous research in flash indexing that may encounter data losses in case of system crash, e.g., [25,26,16,17]. The crash recovery module is mainly responsible on two operations: (1) Once the system restarts after crash, the crash recovery module utilizes the log file entries, written by both the update and flushing modules, to reconstruct the state of the flash storage and in-memory updates just before the crash took place, and (2) maintaining the size of the log file within the allowed limit. As the log space is limited, FAST needs to periodically compact the log entries. Details of this module will be discussed in Section 6.

2.2 FAST Design Goals

FAST avoids the tradeoff of durability and efficiency by using a combination of buffering and logging. Unlike existing efficient-but-not-durable designs [16,17,20,25,26], FAST uses write-ahead-logging and crash recovery to ensure strict system durability. FAST makes tree updates efficient by buffering write operations in main memory and by employing an intelligent flushing policy that optimizes I/O costs for both SSDs and raw flash devices. Unlike existing durable-but-inefficient solutions [2], FAST does not require reading in-flash log entries for each search/update operation, which makes reading FAST trees efficient.

2.3 FAST Data Structure

Other than the underlying index tree structure stored in the flash memory storage, FAST maintains two main data structures, namely, the *Tree Modifications Table*, and *Log File*, described below.

Tree Modifications Table. This is an in-memory hash table (depicted in Figure 2) that keeps track of recent tree updates that did not make it to the flash storage yet. Assuming no hashing collisions, each entry in the hash table represents the modification applied to a unique node identifier, and has the form (*status*, *list*)

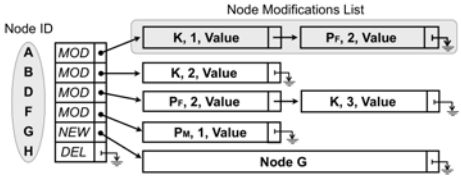


Fig. 2. Tree Modifications Table

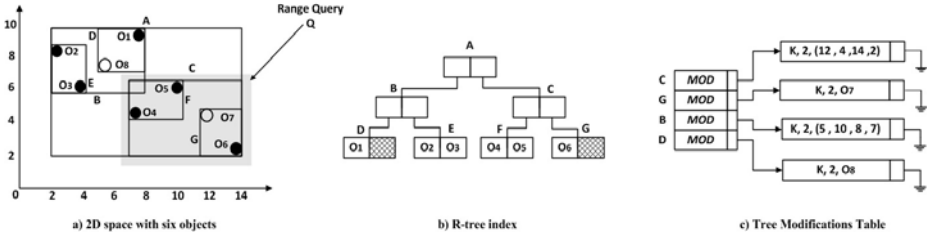


Fig. 3. An illustrating example for all FAST operations

where *status* is either *NEW*, *DEL*, or *MOD* to indicate if this node is newly created, deleted, or just modified, respectively, while *list* is a pointer to a new node, null, or a list of node modifications based on whether the *status* is *NEW*, *DEL*, or *MOD*, respectively. For *MOD* case, each modification in the list is presented by the triple (*type*, *index*, *value*) where *type* is either K, P_F, or P_M, to indicate if the modified entry is the key, a pointer to a flash node, or a pointer to an in-memory node, respectively, while *index* and *value* determines the index and the new value for the modified node entry, respectively. In Figure 2, there are two modifications in nodes A and D, one modification in nodes B and F, while node G is newly created and node H is deleted.

Log File. This is a set of flash memory blocks, reserved for recovery purposes. A log file includes short logs, written *sequentially*, about insert, delete, update, and flushing operations. Each log entry includes the triple (*operation*, *node_list*, *modification*) where *operation* indicates the type of this log entry as either insert, delete, update, or flush, *node_list* includes the list of affected nodes by this operation in case of a flush operation, or the only affected node, otherwise, *modification* is similar to the triple (*type*, *index*, *value*), used in the *tree modifications table*. All log entries are written *sequentially* to the flash storage, which has a much lower cost than *random* writes that call for the erase operation.

2.4 Running Example

Throughout the rest of this paper, we will use Figure 3 as a running example where six objects O_1 to O_6, depicted by small black circles, are indexed by an R-tree. Then, two objects O_7 and O_8, depicted by small white circles, are to be inserted in the same R-tree. Figure 3a depicts the eight objects in the two-dimensional space domain, while Figure 3b gives the flash-resident R-tree with only the six objects that made it to the flash memory. Finally, Figure 3c gives the in-memory buffer (*tree modifications table*) upon the insertion of O_7 and O_8 in the tree.

3 Tree Updates in FAST

This section discusses the update operations in FAST, which include inserting a new entry and deleting/updating an existing entry. An update operation to any tree in FAST

may result in creating new tree nodes as in the case of splitting operations (i.e., when inserting an element in the tree leads to node overflow), deleting existing tree nodes as in the case of merging operations (i.e., when deleting an element from the tree leads to node underflow), or just modifying existing node keys and/or pointers.

Main idea. For any update operation (i.e., insert, delete, update) that needs to be applied to the index tree, FAST does not change the underlying insert, delete, or update algorithm for the tree structure it represents. Instead, FAST runs the underlying update algorithm for the tree it represents, with the only exception of writing any changes caused by the update operation in memory instead of the external storage, to be flushed later to the flash storage, and logging the result of the update operation. A main distinguishing characteristic of FAST is that what is buffered in memory, and also written in the log file, is the *result* of the update operation, not a log of this operation.

Algorithm. Algorithm 1 gives the pseudo code of inserting an object *Obj* in FAST. The algorithms for deleting and updating objects are similar in spirit to the insertion algorithm, and thus are omitted from the paper. The algorithm mainly has two steps: (1) *Executing the insertion in memory* (Line 2 in Algorithm 1). This is basically done by calling the insertion procedure of the underlying tree, e.g., R-tree insertion, with two main differences. First, The insertion operation calls the search operation, discussed later in section 4, to find where we need to insert our data based on the most recent version of the tree, constructed from main memory recent updates and the in-flash tree index structure. Second, the modified or newly created nodes that result back from the insertion operation are *not* written back to the flash storage, instead, they will be returned to the algorithm in a list $\mathcal{L}$. Notice that the insertion procedure may result in creating new nodes if it encounters a split operation. (2) *Buffering and logging the tree updates* (Lines 3 to 22 in Algorithm 1). For each modified node $\mathcal{N}$ in the list $\mathcal{L}$, we check if there is an entry for $\mathcal{N}$ in our in-memory buffer, *tree modifications table*. If this is the case, we first add a corresponding log entry that records the changes that took place in $\mathcal{N}$. Then, we either add the changes in $\mathcal{N}$ to the list of changes in its entry in the *tree modifications table* if this entry status is *MOD*, or update $\mathcal{N}$ entry in the *tree modifications table*, if the entry status is *NEW*. On the other hand, if there is no entry for $\mathcal{N}$ in the *tree modifications table*, we create such entry, add it to the log file, and fill it according to whether $\mathcal{N}$ is a newly created node or a modified one.

Example. In our running example of Figure 3, inserting O_7 results in modifying two nodes, G and C. Node G needs to have an extra key to hold O_7 while node C needs to modify its minimum bounding rectangle that points to G to accommodate its size change. The changes in both nodes are stored in the *tree modifications table* depicted in Figure 3c. The log entries for this operation are depicted in the first two entries of the log file of Figure 4a. Similarly, inserting O_8 results in modifying nodes, D and B.

4 Searching in FAST

Given a query Q, the *search* operation returns those objects indexed by FAST and satisfy Q. The search query Q could be a point query that searches for objects with a specific (point) value, or a range query that searches for objects within a specific range.

Algorithm 1. Insert an Object in the Tree

```
1: Function INSERT(Obj)
   /* STEP 1: Executing the Insertion in Memory only */
2: L ← List of modified nodes from the in-memory execution of inserting Obj in the underlying tree
   /* STEP 2: Buffering and Logging the Updates */
3: for each Node N in L do
4:     HashEntry ← N entry in the Tree Modifications Table
5:     if HashEntry is not NULL then
6:         Add the triple (MOD, N, updates in N) to the log file
7:         if the status of HashEntry is MOD then
8:             Add the changes in N to the list of changes of HashEntry
9:         else
10:            Apply the changes in N to the new node of HashEntry
11:        end if
12:    else
13:        HashEntry ← Create a new entry for N in the Tree Modifications Table
14:        if N is a newly created node then
15:            Add the triple (NEW, N, updates in N) to the log file
16:            Set HashEntry status to NEW, and its pointer to N
17:        else
18:            Add the triple (MOD, N, updates in N) to the log file
19:            Set HashEntry status to MOD, and its pointer to the list of changes that took place in N
20:        end if
21:    end if
22: end for
```

Algorithm 2. Searching for an Object indexed by the Tree

```
1: Function SEARCH(Query Q, Tree Node R)
   /* STEP 1: Constructing the most recent version of R */
2: N ← RetrieveNode(R)
   /* STEP 2: Recursive search calls */
3: if N is non-leaf node then
4:     Check each entry E in N. If E satisfies the query Q, invoke Search(Q, E.NodePointer) for the subtree below E
5: else
6:     Check each entry E in N. If E satisfies the search query Q, return the object to which E is pointing
7: end if
```

An important promise of FAST is that it does not change the main search algorithm for any tree it represents. Instead, FAST complements the underlying searching algorithm to consider the latest tree updates stored in memory.

Main idea. As it is the case for any index tree, the *search* algorithm starts by fetching the root node from the secondary storage, unless it is already buffered in memory. Then, based on the entries in the root, we find out which tree pointer to follow to fetch another node from the next level. The algorithm goes on recursively by fetching nodes from the secondary storage and traversing the tree structure till we either find a node that includes the objects we are searching for or conclude that there are no objects that satisfy the search query. The challenging part here is that the retrieved nodes from the flash storage do not include the recent in-memory stored updates. FAST complements this *search* algorithm to apply the recent tree updates to each retrieved node from the flash storage. In particular, for each visited node, FAST constructs the latest version of the node by merging the retrieved version from the flash storage with the recent in-memory updates for that node.

Algorithm. Algorithm 2 gives the pseudo code of the search operation in FAST. The algorithm takes two input parameters, the query Q, which might be a point or range

Algorithm 3. Retrieving a tree node

```
1: Function RETRIEVENODE(Tree Node R)
2: FlashNode ← Retrieve node R from the flash-resident index tree
3: HashEntry ← R's entry in the Tree Modifications Table
4: if HashEntry is NULL then
5:     return FlashNode
6: end if
7: if the status of HashEntry is MOD then
8:     FlashNode ← FlashNode ∪ All the updates in HashEntry list
9:     return FlashNode
10: end if
    /* We are trying to retrieve either a new or a deleted node */
11: return the node that HashEntry is pointing to
```

query, and a pointer to the root node *R* of the tree we want to search in. The output of the algorithm is the list of objects that satisfy the input query Q. Starting from the root node and for each visited node R in the tree, the algorithm mainly goes through two main steps: (1) *Constructing the most recent version of* R (Line 2 in Algorithm 2). This is mainly to integrate the latest flash-residant version of R with its in-memory stored updates. Algorithm 3 gives the detailed pseudo code for this step, where initially, we read R from the flash storage. Then, we check if there is an entry for R in the *tree modifications table*. If this is not the case, then we know that the version we have read from the flash storage is up-to-date, and we just return it back as the most recent version. On the other hand, if R has an entry in the *tree modifications table*, we either apply the changes stored in this entry to R in case the entry status is *MOD*, or just return the node that this entry is pointing to instead of R. This return value could be null in case the entry status is *DEL*. (2) *Recursive search calls* (Lines 3 to 7 in Algorithm 2). This step is typical in any tree search algorithm, and it is basically inherited from the underlying tree that FAST is representing. The idea is to check if R is a leaf node or not. If *R* is a non-leaf node, we will check each entry *E* in the node. If *E* satisfies the search query *Q*, we recursively search in the subtree below *E*. On the other hand, if *R* is a leaf node, we will also check each entry *E* in the node, yet if *E* satisfies the search query *Q*, we will return the object to which *E* is pointing to as an answer to the query.

Example. Given the range query Q in Figure 3a, FAST search algorithm will first fetch the root node A stored in flash memory. As there is no entry for A in the *tree modifications table* (Figure 3c), then the version of A stored in flash memory is the most recent one. Then, node C is the next node to be fetched from flash memory by the searching algorithm. As the *tree modifications table* has an entry for C with status *MOD*, the modifications listed in the *tree modifications table* for C will be applied to the version of C read from the flash storage. Similarly, the search algorithm will construct the leaf nodes F and G Finally, the result of this query is $\{O_4, O_5, O_6, O_7\}$.

5 Memory Flushing in FAST

As memory is a scarce resource, it will eventually be filled up with incoming updates. In that case, FAST triggers its flushing module to free some memory space by evicting a selected part of the memory, termed a *flushing unit*, to the flash storage. Such flushing is

done in a way that amortizes the cost of expensive random write operations over a high number of update operations. In this section, we first define the flushing unit. Then, we discuss the flushing policy used in FAST. Finally, we explain the FAST flushing algorithm.

5.1 Flushing Unit

An important design parameter, in FAST, is the size of a *flushing unit*, the granularity of consecutive memory space written in the flash storage during each flush operation. Our goal is to find a suitable *flushing unit* size that minimizes the average cost of flushing an update operation to the flash storage, denoted as C. The value of C depends on two factors: $C_1 = \frac{average\ writing\ cost}{number\ of\ written\ bytes}$; the average cost per bytes written, and $C_2 = \frac{number\ of\ written\ bytes}{number\ of\ updates}$; the number of bytes written per update. This gives $C = C_1 \times C_2$.

Interestingly, the values of C_1 and C_2 show opposite behaviors with the increase of the *flushing unit* size. First consider C_1. On raw flash devices (e.g., ReMix [11]), for a *flushing unit* smaller than a flash block, C_1 decreases with the increase of the flushing unit size (see [19] for more detail experiments). This is intuitive, since with a larger *flushing unit*, the cost of erasing a block is amortized over more bytes in the flushing unit. The same is also true for SSDs since small random writes introduce large garbage collection overheads, while large random writes approach the performance of sequential writes. Previous work has shown that, on several SSDs including the ones from Samsung, MTron, and Transcend, random write latency per byte increases by $\approx 32\times$ when the write size is reduced from 16KB to 0.5KB [5]. Even on newer generation SSDs from Intel, we observed an increase of $\approx 4\times$ in a similar experimental setup. This suggests that a flushing unit should *not* be very small, as that would result in a large value of C_1. On the other hand, the value of C_2 increases with increasing the size of the *flushing unit*. Due to non-uniform updates of tree nodes, a large flushing unit is unlikely to have as dense updates as a small flushing unit. Thus, the larger a *flushing unit* is, the less the number of updates per byte is (i.e., the higher the value of C_2 is). Another disadvantage of large *flushing unit* is that it may cause a significant pause to the system. All these suggest that the *flushing unit* should *not* be very large.

Deciding the optimal size of a *flushing unit* requires finding a sweet spot between the competing costs of C_1 and C_2. Our experiments show that for raw flash devices, a *flushing unit* of one flash block minimizes the overall cost. For SSDs, a *flushing unit* of size 16KB is a good choice, as it gives a good balance between the values of C_1 and C_2.

5.2 Flushing Policy

The main idea of FAST *flushing policy* is to minimize the average cost of writing each update to the underlying flash storage. To that end, FAST flushing policy aims to flush the in-memory tree updates that belong to the *flushing unit* that has the highest number of in-memory updates. In that case, the cost of writing the *flushing unit* will be amortized among the highest possible number of updates. Moreover, since the maximum number of updates are being flushed out, this frees up the maximum amount of

Algorithm 4. Flushing Tree Updates

```
1: Function FLUSHTREEUPDATES()
   /* STEP 1: Finding out the list of flushed tree nodes */
2: FlushList ← {φ}
3: MaxUnit ← Extract the Maximum from FlushHeap
4: for each Node 𝒩 in tree modifications table do
5:     if 𝒩 ∈ MaxUnit then
6:         ℱ ← RetrieveNode(𝒩)
7:         FlushList ← FlushList ∪ ℱ
8:     end if
9: end for
   /* STEP 2: Flushing, logging, and cleaning selected nodes */
10: Flush all tree updates ∈ FlushList to flash memory
11: Add (Flush, All Nodes in FlushList) to the log file
12: for each Node ℱ in FlushList do
13:     Delete ℱ from the Tree Modifications Table
14: end for
```

memory used by buffered updates. Finally, as done in the update operations, the flushing operation is logged in the log file to ensure the *durability* of system transactions.

Data structure. The flushing policy maintains an in-memory max heap structure, termed *FlushHeap*, of all *flushing units* that have at least one in-memory tree update. The max heap is ordered on the number of in-memory updates for each *flushing unit*, and is updated with each incoming tree update.

5.3 Flushing Algorithm

Algorithm 4 gives the pseudo code for flushing tree updates. The algorithm has two main steps: (1) *Finding out the list of flushed tree nodes* (Lines 2 to 9 in Algorithm 4). This step starts by finding out the victim *flushing unit*, *MaxUnit*, with the highest number of in-memory updates. This is done as an O(1) heap extraction operation. Then, we scan the *tree modifications table* to find all updated tree nodes that belong to *MaxUnit*. For each such node, we construct the most recent version of the node by retrieving the tree node from the flash storage, and updating it with the in-memory updates. This is done by calling the *RetrieveNode*($\mathcal{N}$) function, given in Algorithm 3. The list of these updated nodes constitute the list of to be flushed nodes, *FlushList*. (2) *Flushing, logging, and cleaning selected tree nodes* (Lines 10 to 14 in Algorithm 4). In this step, all nodes in the *FlushList* are written once to the flash storage. As all these nodes reside in one *flushing unit*, this operation would have a minimal cost due to our careful selection of the *flushing unit* size. Then, similar to update operations, we log the flushing operation to ensure *durability*. Finally, all flushed nodes are removed from the *tree modifications table* to free memory space for new updates.

Example. In our running example given in Figure 3, assume that the memory is full, hence FAST triggers its flushing module. Assume also that nodes B, C, and D reside in the same *flushing unit* B_1, while nodes E, F, and G reside in another *flushing unit* B_2. The number of updates in B_1 is three as each of nodes B, C and D has been updated once. On the other hand, the number of updates in B_2 is one because nodes E and F has no updates at all, and node G has only a single update. Hence, *MaxUnit* is set to B_1, and

we will invoke *RetrieveNode* algorithm for all nodes belonging to B_1 (i.e., nodes B, C, and D) to get the most recent version of these nodes and flush them to flash memory. Then, the log entry ($Flush$; Nodes B, C, D) is added to the log file (depicted as the last log entry in Figure 4a). Finally, the entries for nodes B, C, and D are removed from the *tree modifications table*.

6 Crash Recovery and Log Compaction in FAST

As discussed before, FAST heavily relies on storing recent updates in memory, to be flushed later to the flash storage. Although such design efficiently amortizes the expensive random write operations over a large number of updates, it poses another challenge where memory contents may be lost in case of system crash. To avoid such loss of data, FAST employs a crash recovery module that ensures the *durability* of in-memory updates even if the system crashed. The crash recovery module in FAST mainly relies on the log file entries, written sequentially upon the update and flush operations.

6.1 Recovery

The recovery module in FAST is triggered when the system restarts from a crash, with the goal of restoring the state of the system just before the crash took place. The state of the system includes the contents of the in-memory data structure, *tree modifications table*, and the flash-resident tree index structure. By doing so, FAST ensures the *durability* of all non-flushed updates that were stored in memory before crash.

Main Idea. The main idea of the *recovery* operation is to scan the log file bottom-up to be aware of the flushed nodes, i.e., nodes that made their way to the flash storage. During this bottom-up scanning, we also find out the set of operations that need to be replayed to restore the *tree modifications table*. Then, the *recovery* module cleans all the flash blocks, and starts to replay the non-flushed operations in the order of their insertion, i.e., top-down. The replay process includes insertion in the *tree modifications table* as well as a new log entry. It is important here to reiterate our assumption that there will be no crash during the recovery process, so, it is safe to keep the list of operations to be replayed in memory. If we will consider a system crash during the recovery process, we might just leave the operations to be replayed in the log, and scan the whole log file again in a top-down manner. In this top-down scan, we will only replay the operations for non-flushed nodes, while writing the new log entries into a clean flash block. The result of the crash recovery module is that the state of the memory will be stored as it was before the system crashes, and the log file will be an exact image of the *tree modifications table*.

Log#	Operation	Node	Modification
1	MOD	C	K, 2, (12,4,14,2)
2	MOD	G	K, 2, O_7
3	MOD	B	K, 2, (5,10, 8, 7)
4	MOD	D	K, 2, O_8
5	FLUSH	B, C, D	*

(a) FAST Log File

Log#	Operation	Node	Modification
2	MOD	G	K, 2, O_7

(b) FAST Log File after Crash Recovery

Fig. 4. FAST Logging and Recovery

Algorithm. Algorithm 5 gives the pseudo code for crash recovery in FAST, which has two main steps: (1) *Bottom-Up scan* (Lines 2 to 12 in Algorithm 5). In this step, FAST

Algorithm 5. Crash Recovery

```
1: Function RecoverFromCrash()
   /* STEP 1: Bottom-Up Cleaning */
2: FlushedNodes ← φ
3: for each Log Entry L in the log file in a reverse order do
4:     if the operation of L is Flush then
5:         FlushedNodes ← FlushedNodes ∪ the list of nodes in L
6:     else
7:         if the node in entry L ∉ FlushedNodes then
8:             Push L into the stack of updates RedoStack
9:         end if
10:    end if
11: end for
12: Clean all the log entries by erasing log flash blocks
   /* Phase 2: Top-Down Processing */
13: while RedoStack is not Empty do
14:    Op ← Pop an update operation from the top of RedoStack
15:    Insert the operation Op into the tree modifications table
16:    Add a log entry for Op in the log file
17: end while
```

scans the log file bottom-up, i.e., in the reverse order of the insertion of log entries. For each log entry $\mathcal{L}$ in the log file, if the operation of $\mathcal{L}$ is *Flush*, then we know that all the nodes listed in this entry have already made their way to the flash storage. Thus, we keep track of these nodes in a list, termed *FlushedNodes*, so that we avoid redoing any updates over any of these nodes later. On the other side, if the operation of $\mathcal{L}$ is not *Flush*, we check if the node in $\mathcal{L}$ entry is in the list *FlushedNodes*. If this is the case, we just ignore this entry as we know that it has made its way to the flash storage. Otherwise, we push this log entry into a stack of operations, termed *RedoStack*, as it indicates a non-flushed entry at the crash time. At the end of this step, we erase the log flash blocks, and pass the *RedoStack* to the second step. (2) *Top-Down processing* (Lines 13 to 17 in Algorithm 5). This step basically goes through all the entries in the *RedoStack* in a top-down way, i.e., the order of insertion in the log file. As all these operations were not flushed by the crash time, we just add each operation to the *tree modifications table* and add a corresponding log entry. The reason of doing these operations in a top-down way is to ensure that we have the same order of updates, which is essential in case one node has multiple non-flushed updates. At the end of this step, the *tree modifications table* will be exactly the same as it was just before the crash time, while the log file will be exactly an image of the *tree modifications table* stored in the flash storage.

Example. In our running example, the log entries of inserting Objects O_7 and O_8 in Figure 3 are given as the first four log entries in Figure 4a. Then, the last log entry in Figure 4a corresponds to flushing nodes B, C, and D. We assume that the system is crashed just after inserting this flushing operation. Upon restarting the system, the *recovery* module will be invoked. First, the *bottom-up scanning* process will be started with the last entry of the log file, where nodes B, C, and D are added to the list *FlushedNodes*. Then, for the next log entry, i.e., the fourth entry, as the node affected by this entry D is already in the *FlushedNodes* list, we just ignore this entry, since we are sure that it has made its way to disk. Similarly, we ignore the third log entry for node B. For the second log entry, as the affected node G is not in the *FlushedNodes* list, we know that this operation did not make it to the storage yet, and we add it to the

RedoStack to be redone later. The *bottom-up scanning* step is concluded by ignoring the first log entry as its affected node C is already flushed, and by wiping out all log entries. Then, the *top-down processing* step starts with only one entry in the *RedoStack* that corresponds to node G. This entry will be added to the *tree modifications table* and log file. Figure 4b gives the log file after the end of the *recovery* module which also corresponds to the entries of the *tree modifications table* after recovering from failure.

6.2 Log Compaction

As FAST log file is a limited resource, it may eventually become full. In this case, FAST triggers a *log compaction* module that organizes the log file entries for better space utilization. This can be achieved by two space saving techniques: (a) Removing all the log entries of flushed nodes. As these nodes have already made their way to the flash storage, we do not need to keep their log entries anymore, and (b) Packing small log entries in a larger writing unit. Whenever a new log entry is inserted, it mostly has a small size that may occupy a flash page as the smallest writing unit to the flash storage. At the time of compaction, these small entries can be packed together to achieve the maximum possible space utilization.

The main idea and algorithm for the *log compaction* module are almost the same as the ones used for the *recovery* module, with the exception that the entries in the *RedoStack* will not be added to the *tree modifications table*, yet they will just be written back to the log file, in a more compact way. As in the *recovery* module, Figures 4a and 4b give the log file before and after *log compaction*, respectively. The *log compaction* have similar expensive cost as the *recovery* process. Fortunately, with an appropriate size of log file and memory, it will not be common to call the *log compaction* module.

It is unlikely that the *log compaction* module will not really compact the log file much. This may take place only for a very small log size and a very large memory size, as there will be a lot of non-flushed operations in memory with their corresponding log entries. Notice that if the memory size is small, there will be a lot of flushing operations, which means that *log compaction* can always find log entries to be removed. If this unlikely case takes place, we call an *emergency flushing* operation where we force flushing all main memory contents to the flash memory persistent storage, and hence clean all the log file contents leaving space for more log entries to be added.

7 Experimental Evaluation

This section experimentally evaluates the performance of FAST, compared to the state-of-the-art algorithms for one-dimensional and multi-dimensional flash index structures: (1) Lazy Adaptive Tree (LA-tree) [2]: LA-tree is a flash friendly one dimensional index structure that is intended to replace the B-tree. LA-tree stores the updates in cascaded buffers residing on flash memory and, then empties these buffers dynamically based on the operations workload. (2) FD-tree [16,17]: FD-tree is a one-dimensional index structure that allows small random writes to occur only in a small portion of the tree called the head tree which exists at the top level of the tree. When the capacity of

the head tree is exceeded, its entries are merged in batches to subsequent tree levels. (3) RFTL [25]: RFTL is a mutli-dimensional tree index structure that adds a buffering layer on top of the flash translation layer (FTL) in order to make R-trees work efficiently on flash devices.

All experiments are based on an actual implementation of FAST, LA-tree, FD-tree, and RFTL inside PostgreSQL [1]. We instantiate B-tree and R-tree instances of FAST, termed FAST-Btree and FAST-Rtree, respectively, by implementing FAST inside the GiST generalized index structure [8], which is already built inside PostgreSQL. In our experiments, we use two synthetic workloads: *(1) Lookup intensive workload* (W_L): that includes 80% search operations and 20% update operations (i.e., insert, delete, or update). *(2) Update intensive workload*, (W_U): that includes 20% search operations and 80% update operations.

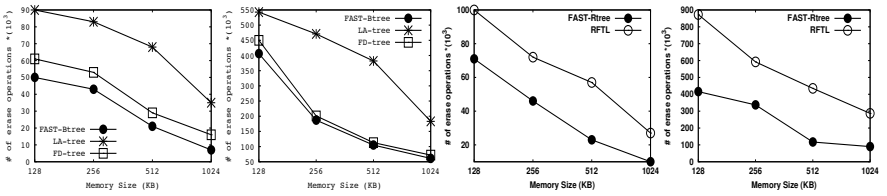
Unless mentioned otherwise, we set the number of workload operations to 10 million operations, main memory size to 256 KB (i.e., the amount of memory dedicated to main memory buffer used by FAST), tree index size to 512 MB, and log file size to 10 MB, which means that the default log size is $\approx$2% of the index size.

The experiments in this section mainly discuss the effect of varying the memory size, log file size, index size, and number of updates on the performance of FAST-Btree, FAST-Rtree, LA-tree, FD-tree, and RFTL. Also, we study the performance of flushing, log compaction, and recovery operations in FAST. In addition, we compare the implementation cost between FAST and its counterparts. Our performance metrics are mainly the number of flash memory erase operations and the average response time. However, in almost all of our experiments, we got a similar trend for both performance measures. Thus, for brevity, we only show the experiments for the number of flash memory erase operations, which is the most expensive operation in flash storage. Although we compare FAST to its counterparts from a performance point of view, however we believe the main contribution of FAST is not in the performance gain. The generic structure and low implementation cost are the main advantages of FAST over specific flash-aware tree index structures.

All experiments were run on both raw flash memory storage, and solid state drives (SSDs). For raw flash, we used the raw NAND flash emulator described in [2]. The emulator was populated with exhaustive measurements from a custom-designed Mica2 sensor board with a Toshiba1Gb NAND TC58DVG02A1FT00 flash chip. For SSDs, we used a 32GB MSP-SATA7525032 SSD device. All the experiments were run on a machine with Intel Core2 8400 at 3Ghz with 4GB of RAM running Ubuntu Linux 8.04.

7.1 Effect of Memory Size

Figures 5(a) and 5(b) give the effect of varying the memory size from 128 KB to 1024 KB (in a log scale) on the number of erase operations, encountered in FAST-Btree, LA-tree, and FD-tree, for workloads W_L and W_U, respectively. For both workloads and for all memory sizes, FAST-Btree consistently has much lower erase operations than that of the LA-tree. More specifically, Fast-Btree results in having only from half to one third of the erase operations encountered by LA-tree. This is mainly due to the smart choice of *flushing unit* and *flushing policy* used in FAST that amortize the block erase operations over a large number of updates.



(a) W_L (b) W_U (c) Spatial-W_L (d) Spatial-W_U

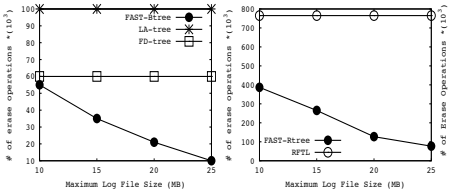
Fig. 5. Effect of Memory Size

The performance of FAST-Btree is slightly better than that of FD-tree, because FD-tree does not employ a crash recovery technique (i.e., no logging overhead). FAST still performs better than FD-tree due to FAST flushing policy that smartly selects the best block to be flushed to flash memory. Although the performance of FD-tree is close to FAST-Btree, however FAST has the edge of being a generic framework which is applied to many tree index structures and needs less work and overhead (in terms of lines of code) to be incorporated in the database engine.

Figures 5(c) and 5(d) give similar experiments to that of Figures 5(a) and 5(b), with the exception that we run the experiments for two-dimensional search and update operations for both the Fast-Rtree and RFTL. To be able to do so, we have adjusted our workload W_L and W_U to *Spatial*-W_L and *Spatial*-W_U, respectively, which have two-dimensional operations instead of the one-dimensional operations used in W_L and W_U. The result of these experiments have the same trend as the ones done for one-dimensional tree structures, where FAST-Rtree has consistently better performance than RFTL in all cases, with around one half to one third of the number of erase operations encountered in RFTL.

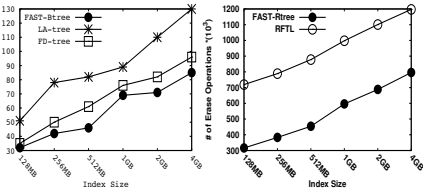
7.2 Effect of Log File Size

Figure 6 gives the effect of varying the log file size from 10 MB (i.e., 2% of the index size) to 25 MB (i.e., 5% of the index size) on the number of erase operations, encountered in FAST-Btree, LA-tree, and FD-tree for workload W_L (Figure 6(a)) and FAST-Rtree and RFTL for workload *Spatial*-W_U (Figure 6(b)). For brevity, we do not show the experiments of FAST-Btree, LA-tree, and FD-tree for workload W_U nor the experiment of FAST-Rtree and RFTL for workload *Spatial*-W_L. As can be seen from the figures, the performance of both LA-tee, FD-tree, and RFTL is not affected by the change of the log file size. This is mainly because these three approaches rely on buffering incoming updates, and hence does not make use of any log file. It is interesting, however, to see that the number of erase operations in FAST-Btree and FAST-Rtree significantly decreases with the increase of the log file size, given that the memory size is set to its default value of 256 KB in all experiments. The justification for this is that with the increase of the log file size, there will be less need for FAST to do log compaction.



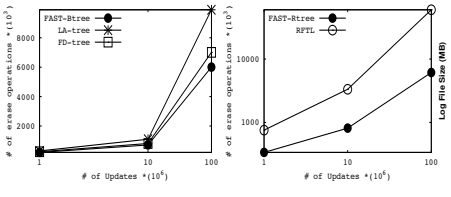
(a) W_L (b) Spatial-W_U

Fig. 6. Effect of FAST log file size



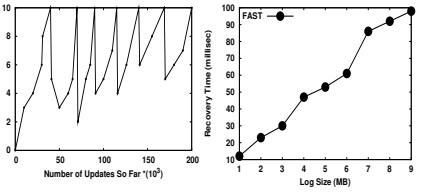
(a) W_L (b) Spatial-W_U

Fig. 7. Effect of Tree index Size



(a) FAST-Btree (b) FAST-Rtree

Fig. 8. Effect of Number of Updates



(a) Log Compaction (b) Recovery

Fig. 9. Log Compaction and recovery

7.3 Effect of Index Size

Figure 7 gives the effect of varying the index size from 128 MB to 4 GB (in a log scale) on the number of erase operations, encountered in FAST-Btree, LA-tree, and FD-tree for workload W_L (Figure 7(a)) and FAST-Rtree and RFTL for workload *Spatial*-W_U (Figure 7(b)). Same as in Section 7.2, we omit other workloads for brevity. In all cases, FAST consistently gives much better performance than its counterparts. Both FAST and other index structures have similar trend of a linear increase of the number of erase operations with the increase of the index size. This is mainly because with a larger index, an update operation may end up modifying more nodes in the index hierarchy, or more overlapped nodes in case of multi-dimensional index structures.

7.4 Effect of Number of Updates

Figure 7 gives the effect of varying the number of update operations from one million to 100 millions (in a log scale) on the number of erase operations for both one-dimensional (i.e., FAST-Btree, LA-tree, and FD-tree in Figure 8(a)) and multi-dimensional index structures (i.e., FAST-Rtree and RFTL in Figure 8(b)). As we are only interested in update operations, the workload for the experiments in this section is just a stream of incoming update operations, up to 100 million operations. As can be seen from the figure, FAST scales well with the number of updates and still maintains its superior performance over its counterparts from both one-dimensional (LA-tree) and multi-dimensional index structures (RFTL). FAST performs slightly better than FD-tree; this is because FD-tree (one dimensional index structure) is buffering some of the tree

updates in memory and flushes them when needed, but FAST applies a smart flushing policy, which flushes only the block with the highest number of updates.

7.5 Log Compaction

Figure 9(a) gives the behavior and frequency of log compaction operations in FAST when running a sequence of 200 thousands update operations for a log file size of 10 MB. The Y axis in this figure gives the size of the filled part of the log file, started as empty. The size is monotonically increasing with having more update operations till it reaches its maximum limit of 10 MB. Then, the log compaction operation is triggered to compact the log file. As can be seen from the figure, the log compaction operation may compact the log file from 20 to 60% of its capacity, which is very efficient compaction. Another take from this experiment is that we have made only seven log compaction operations for 200 thousands update operations, which means that the log compaction process is not very common, making FAST more efficient even with a large amount of update operations.

7.6 Recovery Performance

Figure 9(b) gives the overhead of the recovery process in FAST, which serves also as the overhead of the log compaction process. The overhead of recovery increases linearly with the size increase of the log file contents at the time of crash. This is intuitive as with more log entries in the log file, it will take more time from the FAST recovery module to scan this log file, and replay some of its operations to recover the lost main memory contents. However, what we really want to emphasize on in this experiment is that the overhead of recovery is only about 100 msec for a log file that includes 9 MB of log entries. This shows that the recovery overhead is a low price to pay to ensure transaction *durability*.

8 Conclusion

This paper presented FAST; a generic framework for flash-aware spatial tree index structures. FAST distinguishes itself from all previous attempts of flash memory indexing in two aspects: (1) FAST is a generic framework that can be applied to a wide class of spatial tree structures, and (2) FAST achieves both *efficiency* and *durability* of read and write flash operations. FAST has four main modules, namely, *update*, *search*, *flushing*, and *recovery*. The *update* module is responsible on buffering incoming tree updates in an in-memory data structure, while writing small entries sequentially in a designated flash-resident log file. The *search* module retrieves requested data from the flash storage and updates it with recent updates stored in memory, if any. The *flushing* module is responsible on evicting flash blocks from memory to the flash storage to give space for incoming updates. Finally, the *recovery* module ensures the durability of in-memory updates in case of a system crash.

References

1. PostgreSQL, http://PostgreSQL, http://www.postgresql.org
2. Agrawal, D., Ganesan, D., Sitaraman, R.K., Diao, Y., Singh, S.: Lazy-adaptive tree: An optimized index structure for flash devices. In: PVLDB (2009)
3. Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J., Manasse, M., Panigrahy, R.: Design Tradeoffs for SSD Performance. In: Usenix Annual Technical Conference, USENIX (2008)
4. Beckmann, N., Kriegel, H.-P., Schneider, R., Seeger, B.: The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In: SIGMOD (1990)
5. Bouganim, L., Jónsson, B., Bonnet, P.: uFLIP: Understanding Flash IO Patterns. In: CIDR (2009)
6. Gray, J., Fitzgerald, B.: Flash Disk Opportunity for Server Applications. ACM Queue (2008)
7. Guttman, A.: R-Trees: A Dynamic Index Structure For Spatial Searching. In: SIGMOD (1984)
8. Hellerstein, J.M., Naughton, J.F., Pfeffer, A.: Generalized search trees for database systems. In: VLDB (1995)
9. Katayama, N., Satoh, S.: The sr-tree: An index structure for high-dimensional nearest neighbor queries. In: SIGMOD (1997)
10. Kim, H., Ahn, S.: BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In: FAST (2008)
11. Lavenier, D., Xinchun, X., Georges, G.: Seed-based Genomic Sequence Comparison using a FPGA/FLASH Accelerator. In: ICFPT (2006)
12. Lee, S., Moon, B.: Design of Flash-Based DBMS: An In-Page Logging Approach. In: SIGMOD (2007)
13. Lee, S.-W., Moon, B., Park, C., Kim, J.-M., Kim, S.-W.: A case for Flash memory SSD in Enterprise Database Applications. In: SIGMOD (2008)
14. Lee, S.-W., Park, D.-J., sum Chung, T., Lee, D.-H., Park, S., Song, H.-J.: A Log Buffer-Based Flash Translation Layer Using Fully-Associate Sector Translation. In: TECS (2007)
15. Leventhal, A.: Flash Storage Today. ACM Queue (2008)
16. Li, Y., He, B., Luo, Q., Yi, K.: Tree indexing on Flash Disks. In: ICDE (2009)
17. Li, Y., He, B., Yang, R.J., Luo, Q., Yi, K.: Tree indexing on solid state drives. In: PVLDB, vol. 3(1) (2010)
18. Moshayedi, M., Wilkison, P.: Enterprise SSDs. ACM Queue (2008)
19. Nath, S., Gibbons, P.B.: Online Maintenance of Very Large Random Samples on Flash Storage. In: VLDB (2008)
20. Nath, S., Kansal, A.: Flashdb: Dynamic self-tuning database for nand flash. In: IPSN (2007)
21. Reinsel, D., Janukowicz, J.: Datacenter SSDs: Solid Footing for Growth (January 2008), http://www.samsung.com/us/business/semiconductor/news/downloads/210290.pdf
22. Sellis, T.K., Roussopoulos, N., Faloutsos, C.: The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In: VLDB (1987)
23. Shah, M.A., Harizopoulos, S., Wiener, J.L., Graefe, G.: Fast Scans and Joins using Flash Drives. In: International Workshop of Data Managment on New Hardware, DaMoN (2008)
24. White, D.A., Jain, R.: Similarity indexing with the ss-tree. In: ICDE (1996)
25. Wu, C., Chang, L., Kuo, T.: An Efficient R-tree Implementation over Flash-Memory Storage Systems. In: GIS (2003)
26. Wu, C., Kuo, T., Chang, L.: An Efficient B-tree Layer Implementation for Flash-Memory Storage Systems. In: TECS (2007)