Geoinformatica (2013) 17:417–448
DOI 10.1007/s10707-012-0164-9

Generic and efficient framework for search trees on flash memory storage systems

Mohamed Sarwat · Mohamed F. Mokbel · Xun Zhou · Suman Nath

Received: 16 February 2012 / Revised: 27 June 2012 /
Accepted: 12 July 2012 / Published online: 30 August 2012

Abstract Tree index structures are crucial components in data management systems. Existing tree index structure are 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 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 Search Trees) as a generic framework for flash-aware 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 tree structures including R-tree and its variants, and (2) FAST achieves both *efficiency* and *durability* of read and write flash operations through memory flushing and crash recovery techniques. Extensive experimental results, based on an actual implementation of FAST inside the GiST

The research of M. Sarwat and M. F. Mokbel 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.

M. Sarwat (✉) · M. F. Mokbel · X. Zhou
Department of Computer Science and Engineering, University of Minnesota - Twin Cities,
200 SE Union Street, Minneapolis, MN 55455, USA
e-mail: sarwat@cs.umn.edu

M. F. Mokbel
e-mail: mokbel@cs.umn.edu

X. Zhou
e-mail: xun@cs.umn.edu

S. Nath
Microsoft Research, One Microsoft Way - Redmond, Redmond, WA 98052, USA
e-mail: sumann@microsoft.com

index structure in PostgreSQL, show that FAST achieves better performance than its competitors.

Keywords Flash memory · Tree · Spatial · Index structure · Storage · Multi-dimensional · Data · System

1 Introduction

Data partitioning 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 B-tree [4], with its variants [10, 27], for one-dimensional indexing, and R-tree [14], with its variants [5, 17, 32, 34], for multi-dimensional indexing. Data partitioning 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 [11, 12, 31]. 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., in forms of SSDs) [3, 21, 23, 28, 33]. Recently, several data-intensive applications have started using custom flash cards (e.g., ReMix [19]) 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 [16, 18, 20, 22, 29].

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 [19]), 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* [7, 20]. 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 [7].

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 sub-optimal performance due to the random writes encountered by these index structures. To remedy this situation, several approaches have been proposed for

[1] In a typical flash memory, the cost of *read*, *write*, and *erase* operations are 25, 200 and 1,500 μs, respectively [3].

flash-aware index structures that either focus on a specific index structure, and make it a flash-aware, e.g., flash-aware B-tree [30, 36] and R-tree [35], or design brand new index structures specific to the flash storage [2, 24, 25].

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 [30], which is designed to use a B-Tree, does not support R-Tree functionalities. RFTL [35] 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 [24, 25, 30, 35, 36]. 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 [30]. In summary, no existing flash-aware tree structure achieves both strict durability and efficiency.

In this paper, we address the above two limitations by introducing FAST; a framework for Flash-Aware Search 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. Such an important property makes FAST a very attractive solution to database industry as it is practical to port it inside the database engine with minimal disturbance to the engine code. (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 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 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 [15] inside PostgrSQL. As GiST is a generalized index structure, FAST can support any tree index structure that GiST is supporting,

including one-dimensional tree index structures (e.g., B-tree [4]) and including but not restricted to R-tree [14], R*-tree [5], SS-tree [34], and SR-tree [17], 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 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 two *flushing policies* that smartly select 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: An overview of Flash Memory storage is given in Section 2. Section 3 highlights related work to FAST. Section 4 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 5–8, respectively. Section 9 gives experimental results. Finally, Section 10 concludes the paper.

2 Flash memory storage overview

Figure 1 gives an overview of a typical flash memory storage device. In flash memory, data is stored in an array of flash blocks. Each block contains ≈64–128 pages, where a page is the smallest unit of access. Flash memory supports three types of operations: *read*, *write*, and *erase*. The *Erase* operation is the most expensive one where it can only be done at the block level and results in setting of all bits within a block to ones. *Read* is a low latency page level operation and can occur randomly anywhere in the flash memory without incurring any additional cost. *Write* is also a page level operation and can be performed only once a page has been previously erased since it sets the required bits to zeros. In that sense, writing on a previously erased block is a low latency operation, and termed as a sequential write, while writing on an already written block will result in a block *erase* operation before the actual write operation, and thus, would incur much higher cost. Current generation flash memory-based storage devices have varying access latencies for each of these operations. On average, compared to read operations, write operations are eight times slower, while erase operations are 60 times slower [6]. The typical access latencies for read, write, and erase operations in flash memory devices are 25, 200 and 1,500 μs, respectively [3].

The Flash Translation layer (FTL) [26] is a layer on top of NAND flash memory that makes the flash memory device acts like a virtual disk. The FTL layer receives read and write commands for logical pages addresses from the application layer and converts them to the internal flash memory commands (i.e., read page, write page, erase block) on physical pages/blocks addresses. To emulate disk like in-place update operation for a logical page P_{logical}, the FTL writes data into a new physical page P_{physical}, maintains a mapping between logical pages and physical pages, and marks

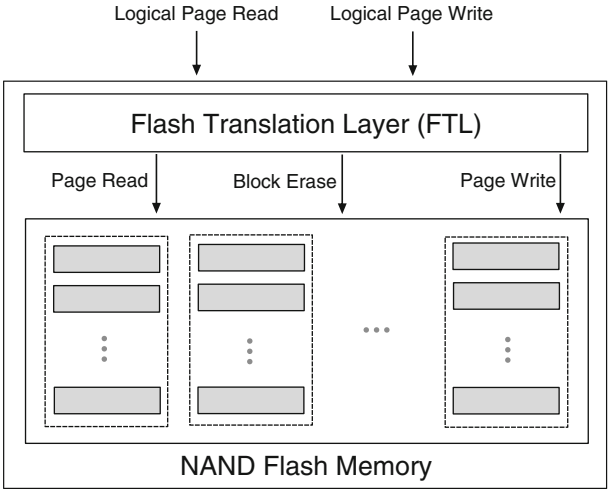


Fig. 1 Flash memory storage. *Grey rectangles* represent pages that are contained in blocks represented by *dotted rectangles*

the previous physical location of P_{physical} as invalid for future garbage collection. Even though FTL allows existing disk based applications to use flash memory without any modiPcations, it needs to internally deal with flash physical constraint of erasing a block before updating a page in that block. Besides this asymmetric read and write latency issue, a flash memory block can only be erased for a limited number of times (e.g., 10^5–10^6), after which it acts like a read-only device [3]. FTL employs various wear-leveling techniques to even out the erase counts of different blocks in the flash memory to increase its longevity [8]. However, still early wear-out of flash memory is one of the big concerns in widely deploying flash memory storage devices [31], and thus, it is of essence that flash memory avoids block erases as much as possible. Recent studies show that current FTL schemes are very effective for the workloads with sequential access write patterns. However, for the workloads with random access patterns, these schemes show very poor performance [9]

3 Related work

Previous approaches for flash-aware index structures can be classified into two categories: (1) Making an existing *specific* index structure flash-aware, which includes flash-aware B-tree (e.g., FlashDB [30] and BFTL [36]) and flash-aware R-tree (e.g., RFTL [35]). The main idea of these index structures is to save the B-tree (R-tree) operations in a reservation buffer residing on main memory. When the reservation buffer is full, its content is totally flushed to flash memory. For instance, BFTL and RFTL are adding a buffering layer on top of the flash translation layer in order to make B-trees work efficiently on flash devices. (2) Designing brand new *one-dimensional* index structures specific to the flash storage, e.g., the LA-tree [2] and the FD-tree [24, 25]. LA-tree is flash friendly 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. FD-tree is also 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.

In terms of the performance-durability trade-off, previous approaches either: (a) achieve efficiency, yet sacrifice durability, by buffering updates in main memory and flush them in batches to flash memory to amortize the cost of random writes [24, 25, 30, 35, 36]. However, storing updates in memory without taking into account system failures, which leads to durability issue, where in-memory updates may be lost if the system crashes, or (b) achieve durability, yet sacrifice efficiency, by writing all the recent updates in a sequential log file [2], hence retrieving the updates from the log file in case of a system crash. However, doing this increases the cost of search for many log entries that need to be read from flash in order to access each tree node with search and update operations [30].

FAST distinguishes itself from all previous techniques in three main aspects: (1) FAST is a general framework for data-partitioning tree index structures built inside GiST [15]. As GiST is a generalized index structure that can instantiate a wide set of data-partitioning trees that include B-tree [4], R-tree [14], R*-tree [5], SS-tree [34], and SR-tree [17]), FAST can support any tree that GiST is supporting. (2) FAST ensures both the *efficiency* and *durability* of system transactions where updates are buffered in memory, yet, an efficient crash recovery technique is triggered in case of a system crash to ensure the durability. (3) FAST is not a brand new index structure, hence does not need to replace existing tree indexes. However, it complements the existing tree index structures in database management systems to make them work efficiently on flash storage devices, with much less implementation cost.

4 Fast system overview

Figure 2 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, which are dominant in embedded systems and custom flash cards (e.g., ReMix [19]), are getting popular for data-intensive applications.

4.1 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

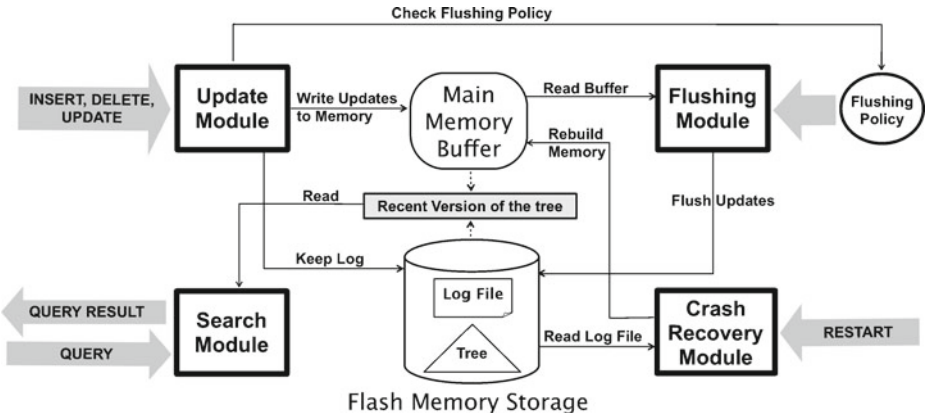


Fig. 2 FAST system architecture

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 5.

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 6.

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 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 employs *flushing policies* 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 7.

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., [24, 25, 35, 36]. 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 8.

4.2 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 [24, 25, 30, 35, 36], 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.

4.3 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 Fig. 3) 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*) 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 quadruple (*TimeStamp*, *type*, *index*, *value*) where *TimeStamp* represents the time at

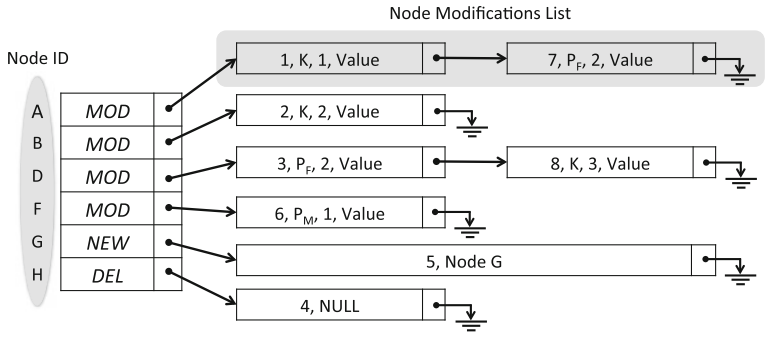


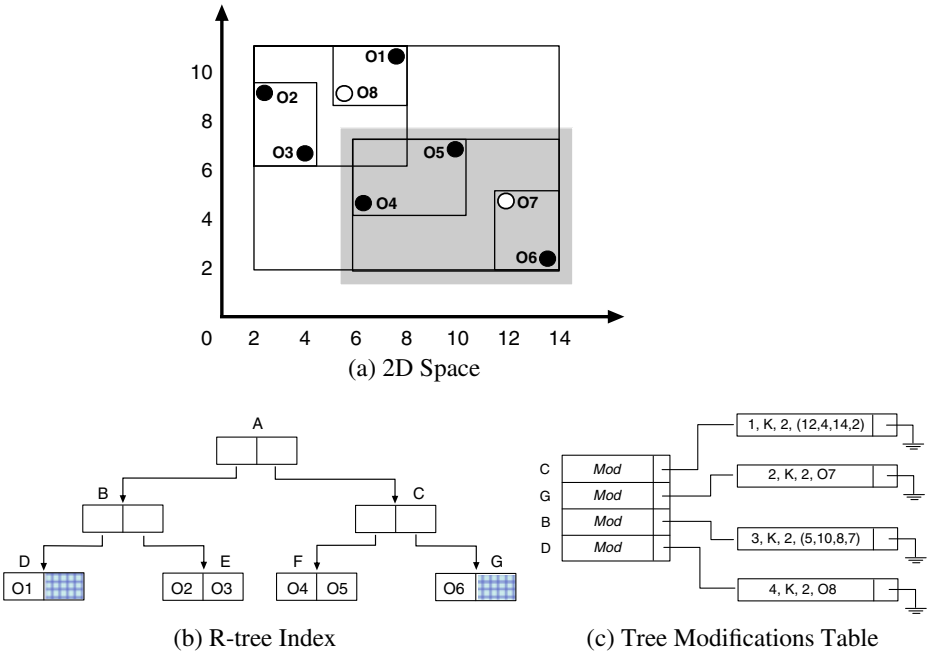
Fig. 3 Tree modifications table

which the update happened, *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 Fig. 3, 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.

4.4 Running example

Throughout the rest of this paper, we will use Fig. 4 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 4a depicts the eight objects in the two-dimensional space domain, while Fig. 4b gives the flash-resident R-tree with only the six objects that made it to the



(a) 2D Space

(b) R-tree Index

(c) Tree Modifications Table

Fig. 4 Illustrating example for search and update operations in FAST

Table 1 Cost analysis parameters

Parameter	Definition
T	The underlying tree index structure to which FAST has been applied
R_M	The average time to read a node update entry from the tree modifications table
W_M	The average time to write a node update entry to the tree modifications table
R_F	The average time to read a tree node from the underlying tree T residing on flash memory
W_F	The average time to write a tree node to the underlying tree T residing on flash memory
E_F	The average time to erase a whole block on the flash memory device

flash memory. Finally, Fig. 4c gives the in-memory buffer (*tree modifications table*) upon the insertion of O_7 and O_8 in the tree.

4.5 Operations cost parameters

For each FAST module, we analyze the cost model of its main operations, including search, update, flushing, crash recovery and log compaction. To this end, we define the parameters given in Table 1.

5 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 6, 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

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
```

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 3–22 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 Fig. 4, 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

Fig. 5 FAST logging and recovery example

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

(a) FAST Log File

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

(a) FAST Log File after Crash Recovery

in Fig. 4c. The log entries for this operation are depicted in the first two entries of the log file of Fig. 5a. Similarly, inserting O_8 results in modifying nodes, D and B

Cost analysis For a given update operation U applied to a tree index structure T, let $y_{i,U} \in \{0, 1\}$ represent whether or not node i of T has been modified by U. Let N be the total number of nodes in T at the time U is applied, then the total cost C_U of update operation U applied on T is as follows:

$$C_U = C_Q + \sum_{i=0}^{N} y_{i,U} * [W_F + W_M + L] \tag{1}$$

The update operation (e.g., insert, delete, modify) requires first a search query Q for a proper leaf node in T. This also takes the same search time C_Q as illustrated above. For each updated node i due to applying U, $y_{i,U} = 1$, and for each of these updates we write a sequential log entry to the log file that each takes W_F time. Hence, the total time to write all log entries is equal to $\sum_{i=0}^{N} y_{i,U} * W_F$. For each updated node i, we also perform a lookup on the tree modifications table to get the entry for node i, which is performed in constant time L. In addition, the total time to write the modifications to all nodes (for which $y_{i,U} = 1$) in the tree modifications table is $\sum_{i=0}^{N} y_{i,U} * W_M$. All of the above sums up to give the update cost given in Eq. 1

6 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. 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 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

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
```

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
```

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 3–7 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 Fig. 4a, 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* (Fig. 4c), 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 by first fetching them from flash memory, and then reading their recent updates from the *tree modifications table*. Finally, the result of this query is $\{O_4, O_5, O_6, O_7\}$.

Cost analysis For a given search query Q applied to a tree index structure T, let $x_{i,Q} \in \{0, 1\}$ represent whether node i of T is visited or not when issuing query Q. Let $M_{i,Q}$ be the number of modifications applied to node i and buffered in the tree modifications table at the time Q is issued. Let N be the total number of nodes in T at the time Q is issued, then the total search cost C_Q on T is as follows:

$$C_Q = \sum_{i=0}^{N} x_{i,Q} * [R_F + (M_{i,Q} \times R_M) + L] \tag{2}$$

Assuming a range query, the search operation returns a number of objects within the query range. In FAST, when reading a node i from the flash-resident R-tree, we also need to accommodate all the corresponding modifications on i that have been recorded in the tree modifications table. Then, the total cost of reading a node i would thus be $(R_F + T_m)$ where T_m is the in-memory processing time for each node. For the in-memory processing part, it first takes constant time L to locate the node in the tree modification table , and then takes a linear scan of the list to apply all the modifications. Given that the number of modifications associated with each node is $M_{i,Q}$, then $T_m = (M_{i,Q} \times R_M) + L$, where $M_{i,Q}$ is upper bounded by the memory size.

7 Memory flushing in FAST

As discussed in Section 5, the effect of all incoming updates in FAST has to be buffered in memory. As memory is a scarce resource, it will eventually be filled up with incoming updates. In that case, FAST triggers its flushing module, equipped with a *flushing policy*, 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.

The motivation of having a *flushing policy* that flushes only part of the memory is twofold: (1) Clearing the whole memory at once will cause a significant pause to the system due to the need of erasing all the flash blocks that include at least one update record in memory. As a result, it is better to consider clearing only part of the memory in a way that does not really pause the system. In this paper, we present two main flushing policies employed by the system, and we empirically evaluate both of them, (2) Considering that we need to flush only part of the memory, it is crucial to select that part in a way that reduces the overhead of the block erase operation.

7.1 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 [19]), for a *flushing unit* smaller than a flash block, C_1 decreases with the increase of the flushing unit size (see [29] 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 16 KB to 0.5 KB [7]. 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 16 KB is a good choice, as it gives a good balance between the values of C_1 and C_2. Note that a flushing unit size of 16 KB also matches the optimal size of a tree node, as suggested by Gray et al. [13]. Thus, with a tree of this optimal node size of 16 KB, we can simply flush one node at a time from the memory.

7.2 Flushing policies

FAST is designed so that different flushing policies can be plugged in to the system. In the rest of this section, we discuss two main flushing policies adopted by FAST: (1) FAST Flushing Policy, and (2) FAST* Flushing Policy.

7.2.1 FAST 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 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. Updates in max heap is O(n), where n is the number of flash blocks with in-memory updates. In the mean time, retrieving the flushing unit with maximum number of updates is an O(1) operation.

7.2.2 FAST flushing policy*

The *FAST* flushing policy* is an enhancement over the FAST flushing policy described in Section 7.2.1. FAST* flushing policy takes into account two parameters that helps in deciding which unit must be flushed: (1) Number of updates per flushing unit: It is the same parameter used by the FAST flushing policy explained in Section 7.2.1; which favors the flash unit that has the highest number of updates, and (2) Time stamp of the flushing unit: which represents the last time a flash block has been updated. When deciding which unit needs to be flushed, that parameter gives higher priority to the flushing unit that has the lowest time stamp (i.e., least recently updated).

FAST Flushing Policy* employs a Top-1 selection algorithm to select a flushing unit to be evicted to flash memory with the objective of maximizing the number of updates per flushing unit and minimizing the time stamp of the flushing unit. The intuition behind such a policy is that it is sometimes better to keep the block that has the highest number of updates in the tree modifications table (i.e., in memory)

and not to flush it, especially if that block is expected to receive more updates (i.e., recently updated block). On the other hand, it might be better to flush a flash block that has a bit less number of updates, but it is not expected to be updated frequently (i.e., least recently updated block). Hence, FAST* flushing policy makes that tradeoff between the two parameters in order to amortize the total number of erase operations on flash memory storage systems.

7.3 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 2–9 in Algorithm 4). This step starts by finding out the victim *flushing unit*, *MaxUnit*, using the flushing policy passed to the algorithm. 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 10–15 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.

Algorithm 4 Flushing Tree Updates

```
1:  Function FlushTreeUpdates(FlushPolicy)
    /* STEP 1: Finding out the list of flushed tree nodes */
2:  FlushList ← {φ}
3:  MaxUnit ← Retrieve Unit to be Flushed uisng FlushPolicy
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 a clean flash memory block
11: Add (Flush, All Nodes in FlushList) to the log file
12: Erase the old flash memory block and update the index pointer to refer the new
    block
13: for each Node F in FlushList do
14:   Delete F from the Tree Modifications Table
15: end for
```

Example In our running example given in Fig. 4, 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, as per FAST flushing policy, *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 Fig. 5a). Finally, the entries for nodes B, C, and D are removed from the *tree modifications table*.

Cost analysis For a given flushing operation F applied to a tree index structure T, Let P_{flush} be the set of tree nodes that belongs to the block selected to be flushed. Let M_p be the number of modifications applied to node p and buffered in the tree modifications table at the time F is applied. Hence, the total cost C_F of flushing operation F applied on T is as follows:

$$C_F = E_F + H + \sum_{p \in P_{\text{flush}}} [R_F + (M_p * R_M) + W_F + L] \tag{3}$$

We decide which unit to flush by employing the flushing policy passed to the algorithm. The cost of this in memory operation H varies based on which flushing policy is activated. For each node $p \in P_{\text{flush}}$, we first need to retrieve the node current value saved in flash memory which costs $\sum_{p \in P_{\text{flush}}} R_F$, and then lookup the node in the tree modifications table in $\sum_{p \in P_{\text{flush}}} L$. For each node $p \in P_{\text{flush}}$, we read all M_p modifications of p that are buffered in the tree modifications table, which sum up to $\sum_{p \in P_{\text{flush}}} (M_p * R_M)$. Before we write the new nodes values, we first erase the whole flash block which costs E_F time. For each node $p \in P_{\text{flush}}$, we write the flushed node new value, that costs $\sum_{p \in P_{\text{flush}}} W_F$. All of the above sum up to give the flushing operation cost given by Eq. 3.

8 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.

In this section, we will first describe the crash recovery module and logging mechanism in FAST. Then, we will follow by discussing the log compaction operation in FAST, which is mainly done to ensure that the log file is within a certain size limit. Log compaction has a very similar operation to the recovery module, and it is crucial to keep up the efficiency of FAST. For simplicity, we will not consider the case of having a system crash during the recovery process, as this can be handled in a similar way to traditional recovery modules in database systems.

8.1 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*.

Algorithm Algorithm 5 gives the pseudo code for crash recovery in FAST, which has two main steps: (1) *Bottom-Up scan* (Lines 2–11 in Algorithm 5). In this step, FAST 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 pass the *RedoStack* to the second step. (2) *Top-Down processing* (Lines 13–19 in Algorithm 5). At the beginning, we first create a new log file F_{new}. Then, 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 in the new Log File F_{new}. 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 new log file F_{new} will be exactly an image of the *tree modifications table* stored in the flash storage. Finally, we change the log file pointer to refer to the new log file F_{new} and we finally erase the old log file flash blocks.

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
   /* Phase 2: Top-Down Processing */
12: Create a new Log File F_new
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 new log file F_new
17: end while
18: Change the Log File pointer to refer to the new Log File F_new
19: Clean all the old log entries by erasing the old log file flash blocks
```

Example In our running example, the log entries of inserting Objects O_7 and O_8 in Fig. 4 are given as the first four log entries in Fig. 5a. Then, the last log entry in Fig. 5a 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 5b 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.

Cost analysis For a given crash recovery operation R applied to a tree index structure T, let Z be the set of operations recorded in the log file. Let α $(0 \leq \alpha \leq 1)$ be the fraction of operations in Z that had been flushed to T before the system fails. Let S_{page}, S_{block}, and S_{log} be the byte size of the flash page, flash block and flash log

file, respectively. Hence, the total cost C_R of a crash recovery operation R applied on T is as follows:

$$C_R = S_{\log} \times \left[\frac{R_F}{S_{\text{page}}} + \frac{R_F}{S_{\text{block}}} \right] + Z \times \alpha \times (W_M + W_F) \quad (4)$$

As all the Z entries in the log file have to be scanned, then the total cost to scan them is $R_F \times \frac{S_{\log}}{S_{\text{page}}}$. In addition, only $Z \times \alpha$ log file operations need to be redone (i.e., written back to the tree modifications table), which results to an additional cost of $Z \times \alpha \times W_M$. As all redone operations are written back to memory, an additional cost of logging them is $Z \times \alpha \times W_F$. The old log file blocks needs to be erased, which incurs a cost of $E_F \times \frac{S_{\log}}{S_{\text{block}}}$. All of the above sums up to give the recovery cost given in Eq. 4.

8.2 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, Fig. 5a and b 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.

Cost analysis The log compaction is almost the same as the crash recovery procedure. The only difference is that records are not redone (written to the tree modifications table). Similar to recovery cost, the log compaction cost $C_C$is as follows:

$$C_C = S_{\log} \times \left[\frac{R_F}{S_{\text{page}}} + \frac{R_F}{S_{\text{block}}} \right] + Z \times \alpha \times W_F \quad (5)$$

9 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 [24, 25]: 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 [35]: 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.

We instantiate B-tree and R-tree instances of FAST using both flushing policies (i.e., FAST flushing policy and FAST* flushing policy), termed FAST-Btree, FAST*-Btree, FAST-Rtree, and FAST*-Rtree , respectively, by implementing FAST inside the GiST generalized index structure [15], which is already built inside PostgreSQL [1]. 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 ≈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.

9.1 Effect of memory size

Figure 6 and b give the effect of varying the memory size from 128 KB to 1,024 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 choice of *flushing unit* and *flushing policy* used in FAST that amortize the block erase operations over a large number of updates. Also, for both experiments, the number of erase operations decreases with the increase of the memory size, which is intuitive as more memory means less frequent need for flushing, and hence less need for block erase operations.

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 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. Comparing the two workloads against each other, we can see that the workload W_U encounters much more erase operations than that of workload W_L. This is mainly because W_U is an update intensive workload which results in many in-memory updates that need to flushed. FAST*-Btree gives a slightly better performance than FAST-Btree as FAST*-Btree employs a flushing policy that does not only rely on the number of updates per flash block, but also takes into account the last time a flash block has been updated. Hence, FAST*-tree gives a chance for those flash blocks that has higher number of updates to stay in memory if more updates are expected to be applied to these blocks.

Figures 7a and b give similar experiments to that of Fig. 6 and 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

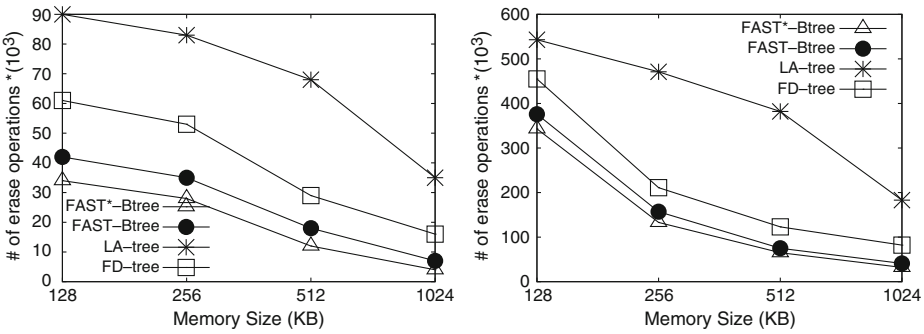
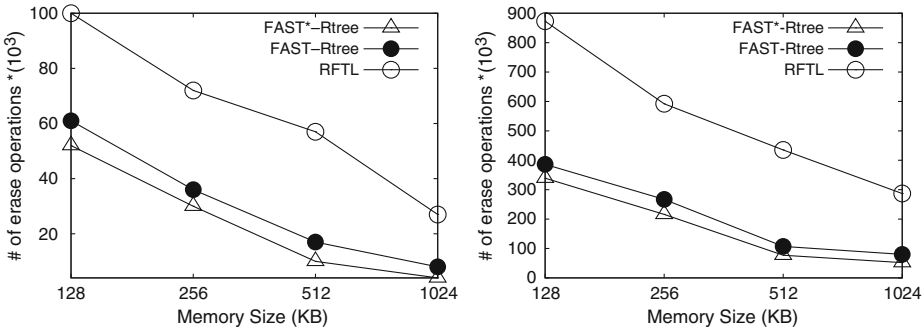


Fig. 6 Effect of memory size on one-dimensional index structure



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

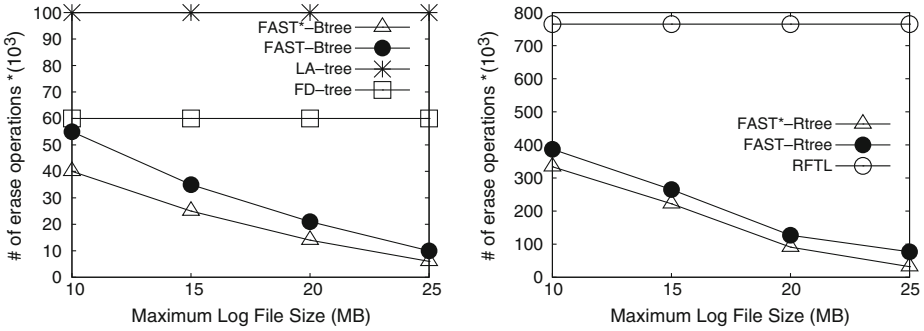
Fig. 7 Effect of memory size on multi-dimensional index structure

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. Similar to the one-dimesnional case, FAST*-Rtree slightly outperforms FAST-Rtree. Comparing the multi-dimensional workload to the one dimensional one shows that the multi-dimensional workload encounters more erase operations which is mainly due to the facts that the update operation may span more nodes. However, even with this, FAST still keeps its performance ratio over its counterparts.

The experiments in Figs. 6 and 7 not only shows that FAST has better performance than its counterparts LA-tree, FD-tree and RFTL, but it also shows the power of the FAST framework where it can be applied to both one-dimensional and multi-dimensional index structures with the same efficiency. In other words, it is not only that FAST is better than LA-tree and FD-tree, but it is also the fact that FAST has the ability to efficiently support multi-dimensional search and update operations in which LA-tree or FD-tree cannot even support.

9.2 Effect of log file size

Figure 8 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 (Fig. 8a) and FAST-Rtree and RFTL for workload *Spatial*-W_U (Fig. 8b). 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



(a) W_L (b) Spatial-W_U

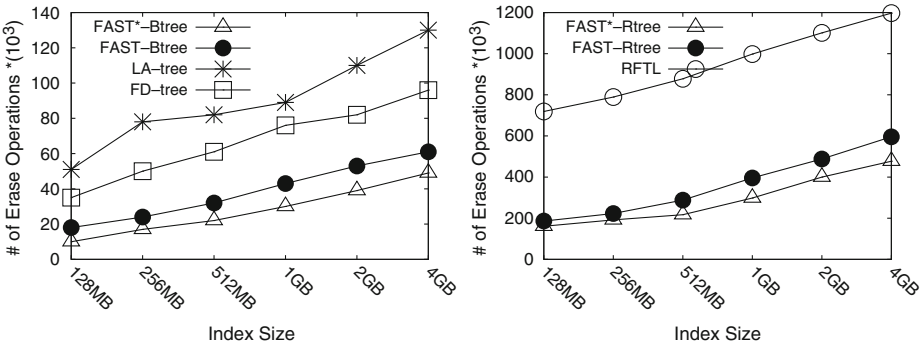
Fig. 8 Effect of FAST log file size

justification for this is that with the increase of the log file size, there will be less need for FAST to do log compaction. FAST*-Btree and FAST*-Rtree shows the same trend as FAST-Btree and FAST-Rtree except that they slightly give better performance due to the fact that they apply the FAST* Flushing policy.

Revisiting Figs. 6 and 7 in Section 9.1, the number of erase operations encountered in both LA-tree, FD-tree, and RFTL were only coming from flushing buffered updates, while the number of erase operations in FAST were coming from two sources, flushing in-memory updates, and log compaction. Then, the experiment in this section (Fig. 8) shows that a large fraction of the erase operations in FAST is coming from the log compaction operation, which can be significantly reduced with the slight increase of the log file. With this, we can see that FAST achieves close to an order of magnitude less erase operations than its counterparts for both one-dimensional and multi-dimensional index structures when having the log file as small as 5 % of the index size, i.e., 25 MB.

9.3 Effect of index size

Figure 9 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 (Fig. 9a) and FAST-Rtree and RFTL for workload *Spatial-*W_U (Fig. 9b). Same as in Section 9.2, 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. Moreover, FAST*-Btree and FAST*-Rtree give a bit better performance than FAST-Btree and FAST-Rtree, respectively. This is basically due to the fact that FAST* flushing policy handles the flash memory updates better than the original FAST flushing policy, hence when the index size increase the possibility that more blocks are updated increases leading to such performance gain for both FAST*-Btree and FAST*-Rtree. The take home message from this experiment is that FAST still maintains its performance gain over its counterparts even with the large increase of the index size.

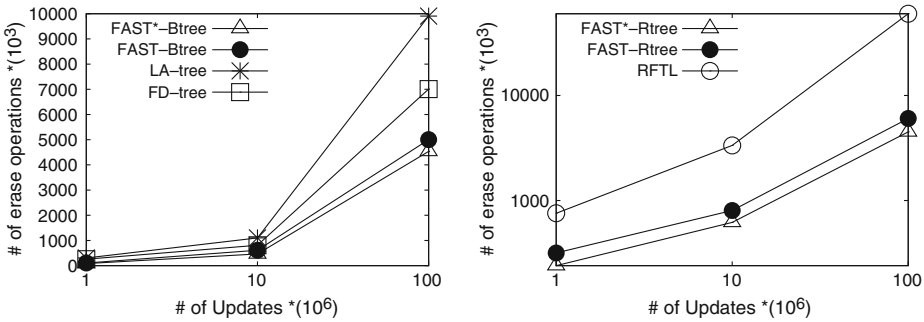


(a) W_L (b) Spatial-W_U

Fig. 9 Effect of tree index size

9.4 Effect of number of updates

Figure 9 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 Fig. 10a) and multi-dimensional index structures (i.e., FAST-Rtree and RFTL in Fig. 10b). 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 flushing policy, which flushes only the block with the highest number of updates. In addition, FAST* slightly outperforms FAST because FAST* flushing policy employs a Top-1 algorithm that maximizes the number of updates per block and minimize the timestamp at which the block has been updated, hence the total amortized update cost in FAST* is less than FAST.



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

Fig. 10 Effect of number of updates

9.5 Flushing performance

Figure 11 illustrates the performance of the *flushing policy* employed by FAST compared to a naive flushing policy, termed *flush-all* that just flushes all the memory contents to the flash storage once. We also compare FAST to a random flushing policy, termed *Rand-Flush* that chooses a block at random and flushed its contents to the flash storage The performance is given with respect to various memory sizes (Fig. 11a) and log file sizes (Fig. 11a). Both experiments were run for FAST-Btree under workload W_U. Running these experiments for FAST-Rtree and other workloads give similar performance, and thus omitted for brevity.

Figure 11a gives the effect of varying the memory size from 128 KB to 1,024 KB on the number of erase operations for flush-all policy, Rand-Flush policy and FAST flushing policy. In all cases, FAST has much lower erase operations than the flush-all and Rand-Flush policies, which is about one fourth of the erase operations for a memory size of 512 KB. The main reason behind this gain in FAST is that it amortizes the cost of the block erase operation over a large number of updates, and hence, will free more memory with each flushing operation. On the other side, in the flush-all or Rand-Flush policy, a block may be erased just because it has only one single update in the memory. In this case, although a block is erased, it does not free much memory space. The Rand-Flush policy performance is slightly better than that of flush-all policy because the Rand-Flush flushes only one block and hence keeping all other blocks in memory, which decrease the cost of random writes on these blocks.

FAST* flushing policy is better than FAST flushing policy as it better amortizes the update cost. FAST* policy may still keep a block that has the highest number of updates in memory if this block has higher potential to be updated soon, and hence the decreasing the number of erase operations applied to that block.

Figure 11b gives a similar experiment to that of Fig. 11a with the exception that we study the effect of changing the log size from 10 MB to 25 MB on the number of erase operations. In all cases, FAST flushing policy is superior, which is intuitive given the above explanation for Fig. 11a. However, an interesting observation from Fig. 11b is that the gain from FAST flushing policy over the flush-all and Rand-Flush policies increases with the increase of the log file size. This means that FAST flushing policy makes better use of the log file than the flush-all and Rand-Flush policies. A justification for this is as follows: As FAST flushing policy evicts a block

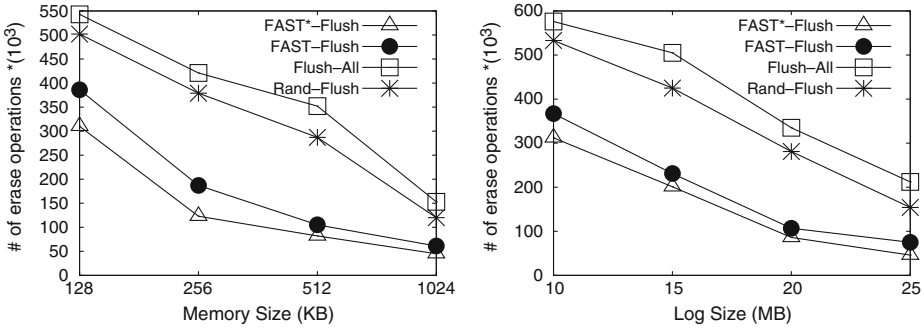


Fig. 11 Flushing performance

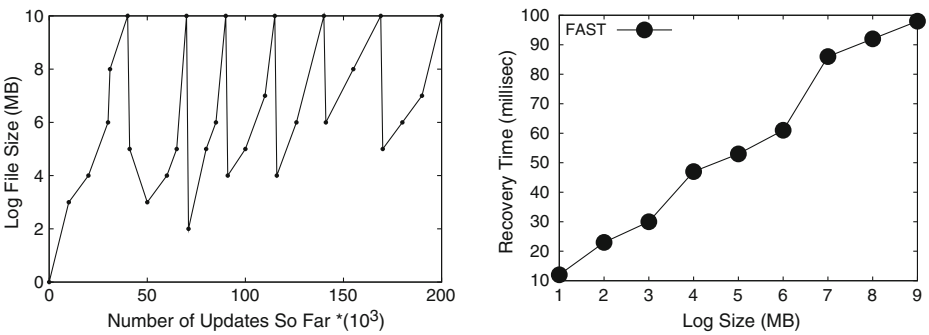
to the storage only if it has high number of updates, the log entry for this flushing operation will include many updated nodes. Then, in the log compaction process, there will be a lot of space for compaction. This would not be the case for the flush-all and Rand-Flush policies where a log entry for a flush operation may include only one flushed node. Then, at the time of log compaction, there will be nothing much to compact, which means that the log compaction will be called again. As discussed in Section 9.2 and Fig. 9, log compaction is a major factor in the number of erase operations. Reducing the frequency of log compaction makes FAST flushing policy more superior than the flush-all policy. Moreover, FAST* flushing policy slightly outperforms FAST flushing policy because of the fact that FAST* may prefer to keep the block that has the highest number of updates in memory leading to less erase operations on the flash memory storage.

9.6 Log compaction

Figure 12a 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.

9.7 Recovery performance

Figure 12b 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



(a) Log Compaction (b) Recovery

Fig. 12 Log compaction and recovery

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 ms 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*.

10 Conclusion

This paper presented FAST; a generic framework for flash-aware data-partitioning 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 tree index 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.

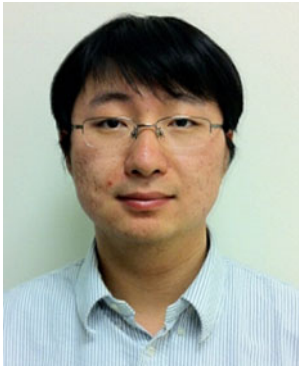
References

1. PostgreSQL. http://www.postgresql.org
2. Agrawal D, Ganesan D, Sitaraman RK, Diao Y, Singh S (2009) Lazy-adaptive tree: an optimized index structure for flash devices. PVLDB
3. Agrawal N, Prabhakaran V, Wobber T, Davis J, Manasse M, Panigrahy R (2008) Design tradeoffs for SSD performance. In: Usenix annual technical conference, USENIX
4. Bayer R, McCreight EM (1972) Organization and maintenance of large ordered indices. Acta Inform 1:173–189
5. Beckmann N, Kriegel H-P, Schneider R, Seeger B (1990) The R*-tree: an efficient and robust access method for points and rectangles. In: SIGMOD
6. Birrell A, Isard M, Thacker C, Wobber T (2007) A design for high-performance flash disks. ACM SIGOPS Oper Syst Rev 41(2):88–93
7. Bouganim L, Jónsson B, Bonnet P (2009) uFLIP: understanding flash IO patterns. In: CIDR
8. Chang Y-H, Hsieh J-W, Kuo T-W (2007) Endurance enhancement of flash-memory storage systems: an efficient static wear leveling design. In: Proceedings of the annual ACM IEEE Design Automation Conference, DAC, pp 212–217
9. Chen S (2009) FlashLogging: exploiting flash devices for synchronous logging performance. In: SIGMOD. New York, NY
10. Comer D (1979) The ubiquitous B-tree. ACM Comput Surv 11(2):121–137
11. Gray J (2006) Tape is dead, disk is tape, flash is disk, RAM locality is king. http://research.microsoft.com/~gray/talks/Flash_is_Good.ppt. Accessed Dec 2006
12. Gray J, Fitzgerald B (2008) Flash disk opportunity for server applications. ACM Queue 6(4):18–23
13. Gray J, Graefe G (1997) The five-minute rule ten years later, and other computer storage rules of thumb. SIGMOD Rec 26(4):63–68
14. Guttman A (1984) R-trees: a dynamic index structure for spatial searching. In: SIGMOD
15. Hellerstein JM, Naughton JF, Pfeffer A (1995) Generalized search trees for database systems. In: VLDB
16. Hutsell W (2007) Solid state storage for the enterprise. Storage Networking Industry Association (SNIA) Tutorial, Fall

17. Katayama N, Satoh, S (1997) The sr-tree: an index structure for high-dimensional nearest neighbor queries. In: SIGMOD
18. Kim H, Ahn S (2008) BPLRU: a buffer management scheme for improving random writes in flash storage. In: FAST
19. Lavenier D, Xinchun X, Georges G (2006) seed-based genomic sequence comparison using a FPGA/FLASH accelerator. In: ICFPT
20. Lee S, Moon B (2007) Design of flash-based DBMS: an in-page logging approach. In: SIGMOD
21. Lee S-W, Moon B, Park C, Kim J-M, Kim S-W (2008) A case for flash memory SSD in enterprise database applications. In: SIGMOD
22. Lee S-W, Park D-J, sum Chung T, Lee D-H, Park S, Song H-J (2007) A log buffer-based flash translation layer using fully-associate sector translation. TECS
23. Leventhal A (2008) Flash storage today. ACM Queue 6(4):24–30
24. Li Y, He B, Luo Q, Yi K (2009) Tree indexing on flash disks. In: ICDE
25. Li Y, He B, Yang RJ, Luo Q, Yi K (2010) Tree indexing on solid state drives. Proceedings of the VLDB Endowment 3(1–2):1195–1206
26. Ma D, Feng J, Li G (2011) LazyFTL: A page-level flash translation layer optimized for NAND flash memory. In: SIGMOD
27. McCreight EM (1977) Pagination of B*-trees with variable-length records. Commun ACM 20(9):670–674
28. Moshayedi M, Wilkison P (2008) Enterprise SSDs. ACM Queue 6(4):32–39
29. Nath S, Gibbons PB (2008) Online maintenance of very large random samples on flash storage. In: VLDB
30. Nath S, Kansal A (2007) Flashdb: dynamic self-tuning database for NAND flash. In: IPSN
31. Reinsel D, Janukowicz J (2008) Datacenter SSDs: solid footing for growth. http://www.samsung.com/us/business/semiconductor/news/downloads/210290.pdf. Accessed Jan 2008
32. Sellis TK, Roussopoulos N, Faloutsos C (1987) The R+-tree: a dynamic index for multi-dimensional objects. In: VLDB
33. Shah MA, Harizopoulos S, Wiener JL, Graefe G (2008) Fast scans and joins using flash drives. In: International Workshop of Data Managment on New Hardware, DaMoN
34. White DA, Jain R (1996) Similarity indexing with the SS-tree. In: ICDE
35. Wu C, Chang L, Kuo T (2003) An efficient R-tree implementation over flash-memory storage systems. In: GIS
36. Wu C, Kuo T, Chang L (2007) An efficient B-tree layer implementation for flash-memory storage systems. TECS

Mohamed Sarwat is a PhD candidate at the Computer Science and Engineering department, University of Minnesota, where he also received his master's degree in computer science in 2011. His research interest lies in the broad area of Database systems, spatio-temporal databases, distributed graph databases, social networking, cloud computing, large-scale data management, data indexing and storage systems. He has been awarded the University of Minnesota Doctoral Dissertation Fellowship in 2012/2013. He has been a recipient of Best Research Paper Award in the 12th international symposium on spatial and temporal databases 2011.

Mohamed F. Mokbel is an associate professor in the Department of Computer Science and Engineering, University of Minnesota. His current main research interests focus on providing database and platform support for spatial data, moving objects, and location-based services. Mohamed is the main architect for the PLACE, Casper, and CareDB systems that provide a database support for location-based services, location privacy, and personalization, respectively. His research work has been recognized by two best paper awards at IEEE MASS 2008 and MDM 2009 and by the NSF CAREER award 2010. Mohamed is currently the general co-chair of SSTD 2011 and program co-chair for MDM 2011, DMSN 2011, and LBSN 2011. Mohamed was also the proceeding chair of ACM SIGMOD 2010, and the program co-chair for ACM SIGSPATIAL GIS 2008, 2009, and 2010. He serves in the editorial board of IEEE Data Engineering Bulletin, Distributed and Parallel Databases Journal, and Journal of Spatial Information Science. Mohamed is an ACM and IEEE member and a founding member of ACM SIGSPATIAL.



Xun Zhou received his B.Eng., and M.Eng., in Computer Science and Technology from Harbin Institute of Technology, Harbin, China in 2007 and 2009 respectively. He is currently a Ph.D. student in Computer Science at the University of Minnesota, Twin Cities. His research interests include spatiotemporal data mining, spatial databases and Geographical Information Systems (GIS). His current application focus is understanding climate change from data.

Suman Nath is a researcher in the Sensing and Energy Research Group at Microsoft Research Redmond. He works on various data management problems in mobile and sensing systems. He received his PhD from Carnegie Mellon University in 2005. He has authored 20+ patents (granted or pending), 70+ papers in various computer science conferences and journals, and received Best Paper Awards at BaseNets 2004, USENIX NSDI 2006, IEEE ICDE 2008, and SSTD 2011. At Microsoft, he received the Gold Star Award, which recognizes excellence in leadership and contributions for Microsoft's long-term success.