Noname manuscript No.
(will be inserted by the editor)

Online Maintenance of Very Large Random Samples on Flash Storage

Suman Nath · Phillip B. Gibbons

Received: January 2009 / Revised: July 2009 / Accepted: date

Abstract Recent advances in flash storage have made it an attractive alternative for data storage in a wide spectrum of computing devices, such as embedded sensors, mobile phones, PDA's, laptops, and even servers. However, flash storage has many unique characteristics that make existing data management/analytics algorithms designed for magnetic disks perform poorly with flash storage. For example, while random reads can be nearly as fast as sequential reads, random writes and in-place data updates are orders of magnitude slower than sequential writes. In this paper, we consider an important fundamental problem that would seem to be particularly challenging for flash storage: efficiently maintaining a very large random sample of a data stream (e.g., of sensor readings). First, we show that previous algorithms such as reservoir sampling and geometric file are not readily adapted to flash. Second, we propose B-File, an energy-efficient abstraction for flash storage to store self-expiring items, and show how a B-File can be used to efficiently maintain a large sample in flash. Our solution is simple, has a small (RAM) memory footprint, and is designed to cope with flash constraints in order to reduce latency and energy consumption. Third, we provide techniques to maintain biased samples with a B-File and to query the large sample stored in a B-File for a subsample of an arbitrary size. Finally, we present an evaluation with flash storage that shows our techniques are *several orders of magnitude faster and more energy-efficient* than (flash-friendly versions of) reservoir sampling and geometric file. A key finding of our study, of potential use to many flash algorithms beyond sampling, is that "semi-random" writes (as defined in the paper) on flash cards are over two orders of magnitude faster and more energy-efficient than random writes.

Keywords flash storage · random sample · sensor networks · semi-random writes

Suman Nath
Microsoft Research
Tel.: +1-425-706-8072
Fax: +1-425-936-7329
E-mail: sumann@microsoft.com

Phillip B. Gibbons
Intel Labs Pittsburgh
Tel.: +1-412-297-4114
Fax: +1-412-297-4110
E-mail: phillip.b.gibbons@intel.com

1 Introduction

Recent technological trends in flash storage have made it an attractive choice for non-volatile data storage in a wide spectrum of computing devices such as PDA's, mobile phones, MP3 players, embedded sensors, etc. The success of flash storage for these devices is due mainly to its superior characteristics such as smaller size, lighter weight, better shock resistance, lower power consumption, less noise, and faster read performance than disk drives [3,8,20]. While flash has been the primary storage media for embedded devices from the very beginning, many market experts expect that it will soon dominate the market of personal computers too [13]. Indeed, several companies including Samsung and Dell have already launched new lines of laptops containing only flash storage [11]. Several companies including SimpleTech and STec offer 512GB flash-based 3.5 inch solid state disk (SSD) drives with claims of 200× performance over 15K RPM enterprise hard drives and better reliability [21,33]. Several Internet service companies are planning to use SSDs in high-end servers, for SSD's

higher throughput, higher energy efficiency, and lower cooling cost in data centers hosting the servers [27].

Flash storage has fundamentally different read/write characteristics than magnetic disks. For example, reading pages at random is nearly as fast as reading pages sequentially, unlike magnetic disks where seek times and rotational latencies make random disk reads many times slower than sequential disk reads (which are in turn many times slower than any flash read). On the other hand, flash writes are immutable and one-time—once written, a data page must be erased before it can be written again. Moreover, the unit of erase often spans a *block* of 32–64 pages—if any of the other pages in the block contain useful data, that data must be copied to new pages before the block is erased. (We will discuss flash characteristics in more detail in Section 2.1.) For this reason, it is well-known that *in-place update*, i.e., overwriting a page that has already been written since the last erase, is very slow on flash storage. Efforts to overcome this limitation (such as via a Flash Translation Layer (FTL) [12]) suffer from another well-known problem: random writes are very slow [3]. Indeed, *the latency and bandwidth (and energy-efficiency) of both random page writes and in-place page updates are over two orders of magnitude worse than sequential page writes.*

In this paper, we consider an important fundamental problem that would seem to be particularly challenging for flash storage: efficiently maintaining a very large (i.e., at least an order of magnitude larger than the available main memory) random sample of a stream of data items. Such very large random samples are useful in a variety of applications. For example, consider a sensor network where each sensor node collects too many readings to store them all locally (because its onboard and attached flash storage is limited) or to transmit them all to a base station (because doing so would rapidly deplete its limited battery). Having each sensor node maintain a random sample of its readings, perhaps biased towards more recent readings, is an attractive approach for addressing the limits of both storage and battery life. Queries can be pushed out to the sensor nodes, and answered (approximately) using the sample points falling within a specified time window. Similarly, random samples are often required in data mining, approximate query answering, statistical analysis, machine learning, and various other streaming applications, which may run in SSD equipped servers. Note that, in all these applications, a very large sample is often required in order to have highly-accurate answers with high-confidence. Specifically, whenever the underlying data has high variance, the query predicate is highly selective, and/or the query contains joins (which can amplify variance), the sample size needs to be in the GBs [15].

There are a number of existing algorithms for maintaining a bounded-size random sample of a stream of data items. Unfortunately, these algorithms were not designed for the unique characteristics of flash storage and hence, not surprisingly, they are ill-suited for flash. For example, *reservoir sampling* [9,29] and *geometric file* [15] are state-of-the-art algorithms for maintaining a large fixed-size sample in memory and on magnetic disk, respectively. However, both rely heavily on in-place updates and/or random writes (details in Sections 3 and 8.2). Moreover, simple optimizations of these algorithms in order to make them more flash-friendly are unable to overcome their flash-unfriendly structure (details in Section 3.3). Indeed, intuitively, maintaining a bounded-size sample seems challenging for flash because new items that are selected for the sample must replace *random* items currently in the sample; this can entail both in-place updates and random writes.

In this paper, we present the first flash-friendly algorithm for maintaining a large bounded-size random sample of a stream of data items. Our algorithm is based on an efficient abstraction, called B-File (Bucket File), for flash storage to store self-expiring items. A B-File consists of multiple buckets, and each item included in the sample is stored in a random bucket according to a distribution dependent (in a non-trivial way) on both the desired sample properties (uniform, biased, etc.) and various overhead trade-offs. When the size of the B-File grows to reach the maximum available flash storage, the B-File automatically shrinks by discarding the largest bucket.

The main efficiency of B-File comes from three properties. First, it always appends data to existing buckets, instead of overwriting any existing data on flash—appending data is far more efficient than updating in place. Second, although these writes are not sequential (because they jump from bucket to bucket), the buckets are structured so that the writes conform to a "semi-random" pattern (where blocks can be selected in any order, but individual pages within blocks are written sequentially from the start of the block; more details in Section 4). A key finding of our study, of potential use to many flash algorithms beyond sampling, is that "semi-random" writes on flash cards are over two orders of magnitude faster and more energy-efficient than random writes. Third, it solves the above "random replace" problem by storing sampled items in buckets according to a preselected random replacement order, so that later all the items in a bucket can be deleted at the same time (i.e., the items are *self-*

expiring). Moreover, B-File's bucketing strategy ensures there are no sub-block deletions.

Another key feature of B-File is that, like reservoir sampling, B-File is effective even when the amount of main memory (RAM) available to the algorithm is very small (e.g., tens of KBs for a 1 GB sample). This contrasts with geometric file, which performs poorly for a 1 GB sample on flash even with 1–10 MBs of RAM. Because embedded devices typically have very limited RAM (e.g., the iMote and SunSpot sensor nodes have 32 KBs and 512 KBs of RAM, respectively), and this RAM must be shared across all sensor node functionalities, B-File's small memory footprint is critical to its suitability for a range of embedded devices.

We also provide efficient techniques to maintain biased samples with a B-File, and to query the large sample stored in a B-File for the sample points within an arbitrary time window.

Our evaluation with flash storage from several vendors shows that our sampling techniques are three orders of magnitude faster and more energy-efficient than previous techniques, including our flash-friendly variants of reservoir sampling and geometric file.

In summary, this paper makes the following contributions.

1. We propose B-File, an energy-efficient abstraction for flash storage to store self-expiring items, and show how B-File can be used to efficiently maintain a large uniform random sample (in particular, a simple random sample) in flash. Our solution has a small (RAM) memory footprint, and is designed to cope with flash constraints in order to reduce latency and energy consumption. We determine several important parameters of B-File that optimize the performance of our algorithm.
2. We define the notion of a *semi-random write*, and show that such writes are over two orders of magnitude more efficient on flash cards than completely random writes. This is an important refinement to the conventional wisdom that random writes are slow on flash, and is a key enabler for B-File.
3. We show how our techniques can be extended to (weighted and age-decaying) biased samples. We also present (flash-friendly, skip-list-based) subsampling techniques for answering ad hoc time-range queries.
4. Using a variety of flash storage, we evaluate our B-File algorithm versus existing state-of-the-art algorithms. Our results show that B-File is *three orders of magnitude* faster and more energy-efficient than existing techniques. Moreover, the number of I/Os and block erases are close to the idealized optimal.

The rest of the paper is organized as follows. Section 2 discusses flash characteristics and design principles for flash-friendly algorithms. Section 3 describes related work. Section 4 discusses semi-random writes. Section 5, 6, and 7 present our basic sampling algorithm, querying algorithm, and several extensions to basic algorithm, respectively. We present evaluation results in Section 8 and conclude in Section 9.

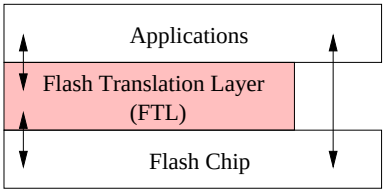


Fig. 1 A flash-based storage system

2 Flash Preliminaries

In this section, we discuss flash storage characteristics and present several well-known design principles for flash-friendly algorithms.

2.1 Flash Characteristics

Figure 1 shows the architecture of a flash-based system. The system consists of flash chips, an optional Flash Translation Layer (FTL), and applications.

2.1.1 Flash Chips

Flash chips are primarily of two types: NOR and NAND. While NOR flash has faster and simpler access procedures, its storage capacity is lower and hence it is used primarily for program storage. NAND flash offers significantly higher storage capacity (e.g., 4GB in a single chip) and is more suitable for storing large amounts of data.

The key properties of NAND flash that directly influence storage design are related to the method in which the media can be read or written, and are discussed in [23]. In summary, all read and write operations happen at page granularity (or for some chips down to $\frac{1}{8}$th of a page granularity), where a page is typically 512–2048 bytes. Pages are organized into blocks, typically of 32 or 64 pages. A page can be written only after erasing the entire block to which the page belongs. However, once a block is erased, all the pages in the block can be written once with no further erasing. Page write cost (ignoring block erase) is typically higher than read, and the block erase requirement makes some writes even more expensive. In particular, for an in-place update, before the erase and write can proceed,

any useful data residing in other pages in the same block must be copied to a new block; this *internal copying* incurs a considerable overhead, e.g., a two orders of magnitude slowdown in our experiments in Section 2.2. A block wears out after 10,000–100,000 repeated writes, and so the write load should be spread out evenly across the chip. Because there is no mechanical latency involved, random read/write is almost as fast (and consumes as much energy) as sequential read/write (assuming the writes are for erased pages).

2.1.2 Flash Cards and the FTL

Portable flash packages such as solid state disks (SSDs), compact flash (CF) cards, secure digital (SD) cards, mini SD cards, micro SD cards and USB sticks provide a disk-like ATA bus interface on top of flash chips. The interface is provided through a Flash Translation Layer (FTL) [12], which is implemented within the microcontroller of the device. Many embedded devices such as cell phones use internal flash chips instead of FTL equipped packages. In such cases, the operating system, e.g. Windows Mobile, implements the FTL in software.

FTL emulates disk-like in-place update for a (logical) address L by writing the new data to a different physical location P, maintaining a mapping between each logical address (L) and its current physical address (P), and marking the old data as invalid for later garbage collection. Thus, although FTL enables disk-based applications to use flash without any modification, it needs to internally deal with flash characteristics (e.g., erasing an entire block before writing to a page). Many recent studies have shown that FTL-equipped flash devices, although a great convenience, suffer many performance problems. In particular, both random writes and in-place updates are very slow, typically two orders of magnitude slower than sequential writes to an erased page [3] (see also Section 2.2). Similar to previous work [16,23], our algorithms address performance problems in today's FTL-equipped flash devices. If future FTL technology eliminates such problems, the algorithms may need to be revisited.

2.2 Design Principles for Flash Algorithms

Because of the above characteristics of flash storage, algorithms designed for flash should follow three key well-known design principles. For completeness, we here present a series of experiments supporting the principles through concrete measurements on both flash chips and flash cards.

The *flash chip* experiments report measurements on a 128MB Toshiba TC58DVG02A1FT00 NAND flash chip [20]. We report only energy numbers here; however, because different micro-ops on a flash chip draw approximately the same amount of current, the energy costs are approximately proportional to the latencies. Each page is 2KBs and each block contains 64 pages. The *flash card* experiments report energy and latency measurements on a 2GB Lexar compact flash (CF) card. We also experimented with a few other flash cards from Kingston and SanDisk, flash chips from FujiFilm XD cards, as well as SSD drives from Samsung and SanDisk, and the conclusions were identical; hence we omit results for those cards and drives here. For concreteness, we assume that the data of interest is comprised of a collection of 32-byte records.

Table 1 Costs of different types of I/Os in a Lexar CF card

Access Pattern	Latency/page (ms)		Energy/page (μJ)	
	Read	Write	Read	Write
Sequential	0.408	0.425	12.7	13.7
Random	0.594	127.1	26.4	7854
Semi-Random	0.463	0.468	13.5	14.9

Principle P1: *Avoid in-place updates.* Flash does not allow updating data in place. Updating some in-flash data d involves several steps: (a) all other data d' from the flash block b containing d first needs to be moved to other locations, (b) the page containing d is read and modified in memory, (c) block b is erased, and (d) d' and the modified d are written back to b. (The last step can be avoided by using an FTL.) In contrast, appending data to flash is cheap, because it can be done by erasing a block and sequentially writing to it until the block is full, without requiring any data movement. With the Toshiba flash chip, overwriting a 32-byte record costs $8554.76\mu J$, while appending a record costs only $1.17\mu J$, nearly four orders of magnitude cheaper. With the Lexar CF card, the costs are $7880.4\mu J$ and $0.21\mu J$, respectively, over four orders of magnitude difference.

Principle P2: *Avoid random writes in flash cards.* A flash chip is a purely electronic device and thus has no mechanically moving parts like disk heads in a magnetic disk drive. Therefore, a raw flash memory chip can provide similar sequential and random access speed. Therefore, one may think that it is not essential to avoid random writes in flash, as many algorithms designed for magnetic disks try to do [15].

However, the situation is different in flash cards and SSD drives (or flash chips with a software FTL driver): such devices provide very poor random write performance. As shown in Table 1, random writes on the

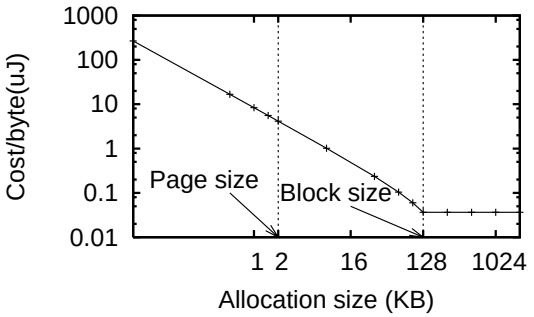


Fig. 2 Allocation/deallocation cost on a Toshiba flash chip

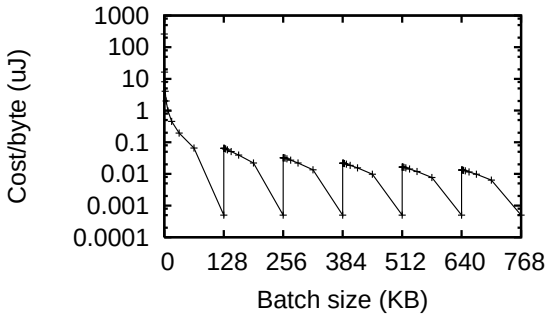


Fig. 3 Random and batch deletion cost on a Toshiba flash chip

Lexar CF card is over two orders of magnitude more expensive than sequential writes. The performance problem stems from an artifact of the FTL design, as discussed in Section 4. In that section, we will introduce semi-random writes, which can be almost as efficient as sequential writes, and show that this design principle should be modified as follows: **Principle P2':** *Avoid random writes unless they are semi-random.*

Principle P3: *Avoid sub-block allocations and sub-block deletions.* Possible choices for an allocation/deallocation size include: (i) sub-page granularity, where fractions of a single flash page are allocated independently (i.e., the same flash page can contain multiple independent data units), (ii) page granularity, where each entire page is allocated independently, and (iii) block granularity, where each entire flash block is allocated independently.

Our experiments using sub-page and page-based allocation show that they suffer from high overhead, for the following reason. As allocated units are deallocated, storage space is freed up. However, reusing deallocated space requires an erase operation on flash. In sub-page and page-level allocations, other data units that reside within a page or the erase block may still contain valid data, and thus, must be moved elsewhere before the entire block can be erased and reused. Figure 2 shows the cost of deallocating and allocating a random data unit on flash of different sizes. It shows that allocation/deallocation at block granularity is two orders of magnitude more efficient than at page or sub-page granularity.

A similar effect arises when an algorithm wishes to delete data without an explicit deallocation. Deleting data in a flash requires a block erase operation. Before erasing a block, valid data in the block needs to be copied to some other location, which requires reading and writing all the valid data. The amortized cost of deleting an item can be made orders of magnitude smaller by deleting multiple items with a single erase operation. This can be done by clustering data that will be deleted together in the same block.

Figure 3 shows the deletion cost per byte when multiple records are clustered and deleted together in a batch (with the minimum number of erase operations). The per-byte cost is highest ($261\mu J$) when only one 32-byte record is deleted, and lowest ($0.0005\mu J$) when all the records in an entire block are deleted together. Even if the total size of records to be deleted in a batch is not a multiple of the 128KB block size, the deletion cost per byte is very small ($< 0.013\mu J$) compared to the maximum cost of deleting only one record within a block. This result implies that if an application needs to delete, say 1000 32-byte records, clustering them into a minimum number of blocks and deleting them in batch can be several orders of magnitude cheaper than deleting them independently.

Additional Goal: *Minimize (DRAM) memory footprint.* In addition to achieving the design principles mentioned above, we add a further goal that our sampling algorithm should minimize its memory footprint. Today's flash-based computing platforms vary widely in the size of available memory (DRAM)—from as few as several kilobytes (e.g., low-end embedded sensors) to many megabytes (e.g., high-end sensors, PDAs) to several gigabytes (e.g., laptops). For generality, an algorithm should optimize for stringent memory constraints, but should be able to exploit greater memory availability in order to improve latency and energy-efficiency.

3 Related Work

In this section, we first present related work, and then show how the most relevant previous work on sampling can be made somewhat more flash-friendly.

3.1 Algorithms for Flash

Recent studies [1,3,16] have proposed application-independent techniques to improve application performance, by optimizing the FTL itself, e.g., to improve the performance of random writes. However, this is quite challenging given that the FTL typically needs to run in a memory-constrained environment (e.g., within a micro-controller), to be sufficiently general to support multiple applications, and to provide fast recovery after a crash [1]. Moreover, in most practical scenarios, application developers do not have access to the FTL (it is either within the micro-controller, or in a proprietary software module), and therefore, the only feasible approach is to optimize the application to use algorithms that perform well on flash. We take this latter approach in this paper. Although, in general, the effort to optimize must be applied to each application, the performance benefits from restructuring an application's algorithm are often orders of magnitude larger than the benefits of application-independent optimizations.

Recent work has shown the feasibility of running a full database system on flash-only computing platforms [19] and running a light-weight database system on flash-based embedded computing devices [7,23] or smartcards [4]. Several other studies have proposed efficient data structures and algorithms for flash storage, including flash-optimized B trees [23], R-trees [32], stacks [20], queues [20], and hash tables [34]. These algorithms seek to follow the Design Principles P1, P2 and P3 discussed in Section 2.2, but they neither study our sampling problem nor propose anything analogous to the key ideas in this paper: B-File, semi-random writes, and a skip-list-based search structure. Moreover, most of these works are designed solely for memory-constrained embedded systems with raw flash chips, whereas our algorithm is *also* optimized for higher-end flash devices (e.g., CF cards or SSDs), where applications must access the flash through an FTL.

3.2 Sampling Algorithms

Because no prior work addressed the problem of maintaining a (bounded-size) random sample on flash, we discuss work related to maintaining bounded-size samples on disk. We omit previous work that deals with *using* a sample (e.g., [2,6]) instead of *maintaining* one, as well as existing streaming algorithms that maintain small random samples in main memory (e.g., [10]).

The fastest streaming algorithm for maintaining a large *fixed-size* random sample on a magnetic disk is due to Jermaine et al. [15]. The algorithm uses an abstraction called the *Geometric File*. The algorithm collects sample items in an in-memory buffer, randomly permutes the items in the buffer, and then divides them into *segments* of geometrically decreasing size. Larger segments are flushed to disk such that each flushed segment overwrites an on-disk segment of the same size. Smaller segments are maintained in memory to avoid small writes. The efficiency of geometric file comes from reducing the number of expensive random writes on disk: only one random access is required per segment and all items within a segment are written sequentially. However, because each new segment overwrites an existing segment, these in-place updates are expensive on flash (see Section 8). Moreover, in addition to the algorithm being more complex than our proposed algorithm, it has a higher in-memory footprint because (i) small segments and segment overflow stacks are maintained in memory, and more importantly, (ii) a large in-memory buffer is required for the algorithm to be effective (a smaller buffer implies smaller segments, which increases the number of random writes).

In [15], the authors also propose using multiple geometric files in parallel for reducing the number of disk head movements. While this scheme eliminates some of the problems of basic geometric file, it introduces additional overheads for maintaining a large number of files. Moreover, compared to a single geometric file, it has a bigger memory footprint for small segments and overflow stacks for all files and a higher space overhead due to higher internal fragmentation and special *dummy segments*.

Reservoir sampling [9,29] is a popular algorithm for maintaining a fixed-size sample of a stream of unknown size. In the basic version of the algorithm, a reservoir R is filled with the first n items (where n is the target size), and after that, the i'th item is selected for R with probability n/i. The selected item overwrites a random item in R. Many optimizations have been proposed to improve the performance of the basic algorithm [15,30]. Although the original algorithm is implicitly designed to maintain a sample in memory, it can be implemented on secondary storage. However, all variants of reservoir sampling require overwriting random sample items in R, and such overwrites are expensive in flash (see Section 8).

Olken and Rotem [24] present techniques for constructing samples in a database environment. However, in addition to not being designed for flash storage, the techniques assume we are sampling from disk-resident, indexed data. Single pass streaming is generally not the goal. When it is, the sample itself is assumed to be stored in main memory during the single pass—avoiding issues of efficiently maintaining the sample on disk. Several I/O efficient index structures such as LSM

Trees [25] and Y-Trees [14] can be used to maintain a large random sample on disk. However, like geometric file, they also require frequent in-place updates, making them unsuitable for flash. Moreover, as shown in [15], they require more random writes and hence perform worse than geometric file. Therefore, we do not consider them in the rest of the paper.

3.3 Adapting the Previous Algorithms to Flash

As neither geometric file nor reservoir sampling were designed for flash, it is natural to consider whether they can be readily modified to be more flash-friendly. We consider each in turn, and show how to improve their flash performance, at the cost of some extra space on the flash.

Geometric file The original geometric file algorithm (described above) can be adapted as follows for more efficient implementation in flash. First, to avoid copying valid data from a block before each erase, a flash block should store data for only a single segment. In this way, an on-flash segment can be overwritten (by erasing entire blocks and writing data to them), without moving data of other segments to other locations. This will introduce internal fragmentation in some blocks, because the last block of a segment can be partially full. However, we can trade additional space for performance in many situations. Second, to reduce fragmentation, very small segments should not be stored in individual blocks. In platforms where memory is limited, all these small segments cannot be maintained in memory. Therefore, these small segments can be stored as append-only log entries in flash. When the log becomes too big, they can be compacted by discarding segments which are supposed to be overwritten by newer segments.

Reservoir sampling The basic reservoir algorithm can be made more efficient by using some extra space E in addition to the reservoir R. Suppose the reservoir R contains a random sample of all the data items seen so far, and a newly-arriving item v gets selected to be added to R, replacing a random item w in R. Instead of overwriting w with v, which would be expensive, we cheaply append v as a log entry in E, deferring the selection of a random w. When the space E becomes full, we need to apply the log entries accumulated in E to R. Note that while the last entry in E must be in R, the second-to-last entry in E must be in R only if the last entry in E is not selected to overwrite it, and so on. In general, the i'th entry in E can be discarded without inserting it to R if *any* of the $(|E|-i)$ subsequent items in E is selected to overwrite it, which has a probability $(1-p^{|E|-i}), p = (|R|-1)/|R|$. By avoiding the insertion of items that get selected by subsequent items in the log, we save expensive replacement operations for them. More precisely, for each $|E|$ items, we incur an expected cost of $|E|/l \times (c_r + c_w) + \frac{(1-p^{|E|})}{(1-p)} \times (c_r + c_w)$ instead of $|E| \times (c_r + c_w)$. Here, l is the number of log entries in a flash page, and c_r (c_w) is the cost of reading (writing) a page. Given a sufficiently large E, the savings can be significant.

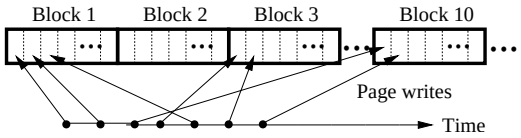


Fig. 4 Semi-random writes

The bottom line The above two sampling algorithms and their adapted versions still require frequent in-place updates. In Section 8, we will show that the adapted algorithms perform better than the original algorithms; however, our algorithm based on B-File can be three orders of magnitude more efficient than the adapted algorithms.

4 Semi-Random Writes

In addition to sequential and random writes, we have also investigated a *semi-random* write pattern where blocks can be selected in any order, but individual pages within blocks are written sequentially from the start of the block. We call the blocks currently selected for write as *open blocks*. Thus, in a semi-random write pattern, multiple sequential writes to open blocks are interleaved with one another. Figure 4 shows an example of a semi-random write pattern, indicated by a sequence of (block id, page id) pairs: (1,1), (1,2), (10,1), (3,1), (1,3), (3,2), (10,2), etc. This pattern has three open blocks with id 1, 3, and 10.

Interestingly, our experiments show that while random writes perform very poorly in existing FTL-equipped devices, semi-random writes perform very close to sequential writes. As shown in Table 1, random writes on a Lexar 2GB CF card are well over two orders of magnitude more expensive than sequential writes, while semi-random writes (with 16 open blocks) are almost as efficient as sequential writes. Similar results hold for several other flash cards and SSDs we tried. The result can be explained by the algorithms used in existing FTLs. The FTL maintains a mapping table between logical addresses and physical addresses. If this

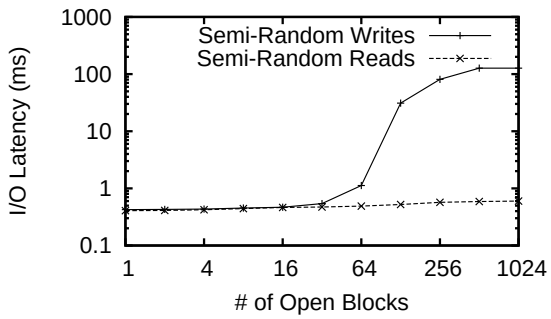


Fig. 5 Effect of number of open blocks in a Lexar CF card

table were to map logical pages to physical pages, the mapping table for a 2GB flash with a 2KB page size and 64 pages/block would be 64MB! Instead, existing flash packages maintain a mapping from logical blocks to physical blocks; for a 2GB flash, this reduces the mapping table to 1MB [3]. For all but the low-end platforms, this enables the mapping table to be stored in memory, which is crucial because its typical access pattern (frequent, random reads and in-place updates, at a word granularity) is very ill-suited for flash. Unfortunately, with a block-level mapping, even when a single page is modified, the entire logical block needs to be written to a new physical block, resulting in poor random write performance.

The performance benefits of semi-random writes are a result of several optimizations within existing FTLs. Many existing FTLs optimize write costs by being lazy; when the i'th logical page of a block is written, the FTL copies and writes the first i pages (instead of all the pages in the block) to a newly allocated block, leaving subsequent (unmodified) pages in the old block; later, when page $j > i$ is modified, pages $(i+1)$ to j are moved and written to the new block, and so on [3]. Semi-random writes do not require moving any unmodified page to the newly allocated block, resulting in a performance comparable to sequential writes. In many other existing FTLs, modified pages are temporarily maintained in logs; logged pages, along with unmodified pages in the same block, are later copied to newly allocated blocks [17,18]. With this strategy as well, semi-random writes do not require copying any unmodified pages across blocks, resulting in superior performance.

The performance of a semi-random write pattern, however, depends on the number N_{open} of open blocks. This is shown in Figure 5, which shows the average latency of a semi-random write operation on a Lexar CF card, as a function of N_{open}. The result shows that semi-random write is efficient for a relatively small $N_{open} \leq 32$; using a large $N_{open} \geq 256$ can make semi-random writes as expensive as random writes. Therefore, algorithms exploiting semi-random write patterns should limit N_{open} to a small number. For example, an application can benchmark its target flash device to generate a graph like Figure 5, pick a value $N_{MaxOpen}$ that gives the maximum tolerable performance of semi-random writes, and then limit $N_{Open} \leq N_{MaxOpen}$.

The inefficiency of a semi-random write pattern with a large value of N_{open} can be explained as follows. In the aforementioned lazy or log-based FTL algorithms, the first write to an open (logical) block goes to a newly allocated physical block, since flash does not allow in-place update of the old physical block. Thus, an open logical block occupies two physical blocks: B_{old} containing old pages and B_{new} containing newly written pages of the logical block. Valid pages from B_{new} and B_{old} are later consolidated into one physical block mapped by the logical block. In both lazy and log-based algorithms, B_{new} is allocated from a limited pool of *log blocks* that are dedicated to the FTL algorithm and are not exposed to applications. Semi-random writes are efficient when the N_{open} blocks fit within this pool of log blocks, because then the log blocks can be written to sequentially and consolidation is done by simply discarding old blocks for their later reuse as log blocks. However, if N_{open} is larger than the available pool of log blocks, a log block may need to be shared by multiple open blocks, which results in more frequent consolidation, e.g., after only a fraction of a logical block is updated. Moreover, this early consolidation is expensive, because an old physical block may contain valid pages that must be copied to the log block during consolidation. The frequency and cost of consolidation increases as the value of N_{open} increases. Similar results have been recently reported for solid state disks as well [5].

The good performance of semi-random writes is also likely to hold for applications directly accessing flash chips. Most such applications will maintain a block-level mapping between logical and physical addresses, resulting in performances similar to existing FTLs. Some applications may decide to maintain a page-level mapping, at the cost of a very large memory footprint and crash-recovery overheads [1]; however, this extreme case will make semi-random (and random) writes perform almost the same as sequential writes, as modified pages will be written sequentially irrespective of the write pattern.

In summary, the orders of magnitude performance benefits of semi-random writes hold across a broad range of flash configurations, including commercial offerings and research prototypes. However, in order to use semi-random writes, algorithms need to know where the block boundaries are, and hence what the block size is—the

Table 2 Notation used in this paper

N	Number of individual B-File buckets
B_T	The tail B-File bucket (a log)
$B_i, i = 1 \cdots N$	The i'th individual B-File bucket
L	Current minimum active level
S'	Data stream seen so far
S	Sample, i.e., $\cup_{i=1}^{N} B_i \cup B_T$
s_{min}	Expected minimum sample size
s_{max}	Guaranteed maximum sample size
α	s_{min}/s_{max}
v, l_v	A stream item and its assigned level
p	Probability of heads in each coin toss
N_{open}	Number of open blocks
$N_{MaxOpen}$	Performance threshold for N_{open}
R, W	Avg. cost to read/write an item in flash

Algorithm 1 *Sample*(s_{min}, s_{max}, N)

Require: Minimum and maximum sample sizes s_{min} and s_{max}, number of B-File buckets N (not counting the tail bucket)

```
1: L ← 1   {L is the current minimum active level}
2: bfile ← new B-File(N)
3: for each stream item v do
4:    l_v ← Level(v, s_min, s_max)   {compute the level}
5:    if l_v ≥ L   {if v selected for the sample} then
6:       bfile.AddItem(v, l_v−L+1)   {append v to B_{l_v−L+1}}
7:       if |bfile| = s_max   {if sample size at its max} then
8:          bfile.DiscardBucket(1)   {discard the items in B_1}
9:          bfile.LeftShift()   {rename each B_{i+1} to be B_i}
10:         L ← L + 1   {increment the minimum active level}
11:      end if
12:   end if
13: end for
```

block size can be readily obtained by querying the flash driver or the FTL.

5 Maintaining Samples on Flash

In this section, we describe our main algorithm and how it can be implemented with our B-File data structure. We will use the notation summarized in Table 2.

5.0.1 Algorithm Overview

At a high level, there are three salient aspects of our algorithm (see Algorithm 1). First, as in the adapted algorithms in Section 3.3, Algorithm 1 will incur some additional storage overhead beyond the sample itself, in order to improve performance. In our case, we allow the sample size to range between a specified expected lower bound (s_{min}) and a specified hard upper bound (s_{max}). This flexibility is useful because it enables us to decouple the addition of a new item to the sample from the deletion of an existing item (to make room). The difference between s_{max} and s_{min} represents the additional flash storage overhead incurred by our algorithm, in order to ensure (on expectation) a sample of size at least s_{min}. On the other hand, because the maintained sample is always a simple random sample (without replacement), any extra sample points beyond s_{min} are not really wasted, as they can be put to good use by applications.

Second, when an item is selected for the sample, we immediately determine its relative priority for deletion compared to other sample points (i.e., *we preselect its random relative replacement order*), and then store the item with sample points of the same priority. Specifically, each item selected for the sample is randomly assigned to one of a logarithmic number of "levels" (by the "Level" function in line 4 of Algorithm 1, details below). This partitions the sampled items into equivalence classes; all items in the same equivalence class are stored in the same "bucket" and will later get discarded at the same time. This allows block-wise erasure (as opposed to random overwrite) of data, and is the key behind the efficiency of our algorithm. We use our new B-File data structure (described in Section 5.2) to store the buckets.

Third, we use the same Level function and a rising threshold L to determine whether an item is selected for the sample. Consider the main loop of Algorithm 1. An item v is selected if its level l_v (computed in line 4) is at least the current threshold L (line 5). A selected item is added to the bucket for its level (line 6). Whenever the sample size reaches s_{max} (line 7), we make room by discarding all the sample points in the first bucket B_1 (line 8), i.e., discarding all items with level L but retaining all items with level $L+1$ or above. Conceptually, we then shift all the buckets to the left, so that the buckets containing sample points are always numbered starting at 1 (line 9). As we are no longer including in our sample any items with level L, but require at least level $L+1$, we increment the threshold (line 10).

Our algorithm is reminiscent at a high level of the sampling component of a previous algorithm for counting the number of 1's in the union of distributed data streams [10], with modest changes. However, at the next level of detail, the previous algorithm (which is designed for main memory and not flash) violates Design Principles P1–P3 from Section 2.2. Thus, our contribution is in (i) designing flash-friendly techniques in support of each step of the algorithm (finding the right data organization, etc.), and (ii) exploring how various parameter choices optimize performance.

5.1 Assigning Levels to Items to Obtain Overall Guarantees

The properties of the sample obtained by Algorithm 1 depend on the level function (Level()) in line 4. We say a level function generates *independent and identically distributed (i.i.d.)* levels if each invocation returns a random value according to the same probability distribution as the others, independent of all the others. We will show that i.i.d. levels imply that Algorithm 1 generates a *simple random sample* (without replacement), i.e., a random sample such that all samples of the given size are equally likely.

Lemma 1 *Consider any run of Algorithm 1 using a level function generating i.i.d. levels on a stream S' seen so far. Let S be the items currently in* B-File *buckets after processing S' and m be the number of items in S. Then S is a simple random sample (without replacement) of size m of the items in S'.*

Proof We must show that all subsets of size m of the items in S' are equally likely. Let n be the number of items in S', and let k be the current value of L in Algorithm 1 after processing S'. Because the levels are i.i.d., each item v in S' has a level $l_v \geq k$ with the same probability (call it P), independently of all other items. Let T be an arbitrary subset of size m of the items in S'. T is selected as the sample if and only if each item in T is assigned a level at least k and each item in S' but not in T is assigned a level less than k. Because levels are assigned independently, this probability is $P^m(1-P)^{n-m}$, which is independent of the choice of T. Thus, all such T are equally likely. □

In order to have only a logarithmic number of levels, we focus on *geometrically distributed* i.i.d. levels, i.e., where the probability of level i decreases exponentially with i. Such a random level can be obtained, for example, by tossing a biased coin—the level is determined by the number of tosses required to get the first head. Let p be the probability of heads on any given coin toss. An item is assigned level i (≥ 1) with probability $p(1-p)^{i-1}$.

Lemma 1 implies that we can maintain a simple random sample using any value of $p \in (0,1)$. However, the value of p determines how the sample size fluctuates, because it determines the expected number of items that are assigned to the current level L (i.e., it determines $|B_1|$) at the point that the total sample size hits the upper bound s_{max}. Because B_1 is discarded at this point, we have that the expected value of s_{min} is s_{max} minus the expected value of $|B_1|$. The following lemma provides a means to select p in order to keep the sample size within a target range.

Lemma 2 *Setting $p = 1-\alpha$, where $\alpha = s_{min}/s_{max}$, ensures that the expected sample size is at least s_{min} (the sample size is always at most s_{max}).*

Proof Suppose at a given point of time, the sample has been computed over a total of n data items. Then, on expectation, $(1-p)^{i-1}p \cdot n$ of these items are assigned to level i, and are placed in the $(i-L+1)$'th bucket $B_{(i-L+1)}$. This gives, on expectation, $|B_k| = (1-p)^{k-1} \cdot |B_1|$ and hence $s_{max} = \sum_{k=1}^{w} |B_k| = |B_1| \sum_{k=1}^{w} (1-p)^{k-1} = |B_1|/p$. Plugging this into our goal that, on expectation, $|B_1| = s_{max} - s_{min}$, we get $(1-p)s_{\max} = s_{\min}$, i.e., $p = 1-\alpha$, where $\alpha = s_{min}/s_{max}$. □

5.2 B-File Design

In this section, we present our main new data structure: the B-File. From the perspective of an application using B-File, a B-File consists of a potentially large set of buckets $\cup_i B_i$ stored on flash storage; denote these buckets as *application buckets*. Physically, however, a B-File stores these buckets in a collection of N *individual buckets* holding the first N application buckets and one *tail bucket* holding all the remaining (typically very small) buckets; denote these (individual and tail) buckets as B-File buckets. The use of a tail B-File bucket is a key optimization for flash, as discussed below.

From the application perspective, the B-File supports the following operators:

- *new* B-File*(N):* Create a new B-File with N individual B-File buckets plus one tail B-File bucket.
- *AddItem(v,i):* Add item v to application bucket B_i. Application buckets can be of arbitrary size.
- *size* and *size(i):* Return the number of items in the entire B-File or in application bucket B_i. (In Algorithm 1, we use "$|bfile|$" as a shorthand for the *size* operator.)
- *DiscardBucket(i):* Discard the items in application bucket B_i, and reclaim the space.

(Algorithm 1 also depicts a *LeftShift* operator, which is used only to simplify the notations and explanations in this paper.)

When used for our sampling algorithm, the sizes of individual application buckets exponentially decrease, with the first bucket B_1 being the largest. At any point of time, the contents of all the buckets represent a simple random sample S over the entire data stream S' seen so far (Lemma 1). Figure 6 depicts a snapshot of a B-File as used by Algorithm 1.

Before explaining the B-File in further detail, it is useful to motivate its design by considering its use

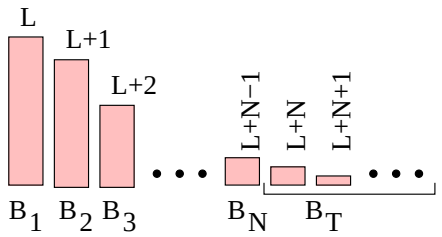


Fig. 6 A snapshot of B-File. Solid bars represent application buckets, text above a bar represents the level of the items in the bucket, and text below a bucket represents the B-File bucket number. The tail B-File bucket B_T contains items with level at least $L + N$.

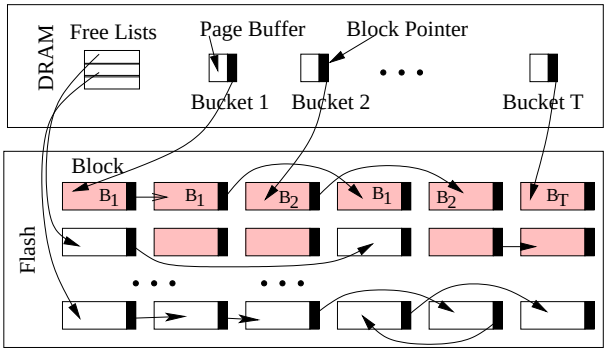


Fig. 7 Physical layout of the buckets in a B-File

in our sampling algorithm. Using the B-File enables the steps of the algorithm to be supported in a flash-friendly way, for the following reasons. First, new items are always appended into the appropriate buckets (either the tail bucket or the corresponding individual bucket)—we avoid in-place updates. Moreover, the B-File maintains an in-memory page of the most recently inserted items for each B-File bucket, which, when full, gets appended to a block associated with the bucket (as discussed in Section 5.2.1). These page flushes fit a semi-random access pattern, as defined in Section 4, with $N_{open} = N + 1$ open blocks. Namely, while the next flushed page can be for any of the $N + 1$ B-File buckets, the pages within a bucket's block are written sequentially. Thus, according to Design Principles P1 and P2′, the write operations are highly-efficient, provided we set $N < N_{MaxOpen}$.

Second, the algorithm clusters items with the same level together into application buckets. The first such bucket is mapped to the first individual B-File bucket. As we shall see, individual B-File buckets are stored using as few blocks as possible. According to the Design Principle P3, this enables highly-efficient deletion of B_1.

Third, the B-File maintains only a few (N) large application buckets as individual B-File buckets. Note that when we use geometrically distributed i.i.d. levels, the size of the application buckets exponentially decreases with level number. Thus, application buckets with higher levels contain very few items. Because storage on flash is best allocated in granularity of a block, allocating a whole block for those small application buckets would be wasteful. Instead, they are rolled into the tail B-File bucket.

Finally, the parameter N provides a tunable control over not only the number of open blocks (as discussed above) but also the B-File's (RAM) memory footprint. The number of memory words used by the B-File (and hence by the sampling algorithm) is linear in N, and otherwise constant. Thus, RAM-constrained embedded devices can use the algorithm with smaller values of N. On the other hand, as we show in Section 5.2.2, the I/O cost of maintaining buckets decreases with increasing N; hence, less constrained devices can take advantage of the larger available RAM by using larger values of N.

5.2.1 Bucket Layout and Maintenance

Figure 7 depicts the physical layout of B-File buckets. The top half shows the in-memory portion. For each B-File bucket B_i (including the tail bucket), we maintain an in-memory data structure called $B_i.header$. The header contains a *page buffer* that can temporarily hold one flash page worth of data, and a *block pointer* that points to the first flash block and page containing the items in that bucket. When an item is added to a bucket, it is temporarily put in its page buffer. When the page buffer holds one page worth of data, the buffer is flushed to the next available page, which is next to the page and within the block pointed to by the block pointer. Search or retrieval of items in a bucket starts with the block pointer.

To cope with the unique properties of flash, the physical layout of the buckets on the flash must be carefully designed in order to obtain high efficiency. Consider the following alternatives. If pages of a single block were used by different buckets, discarding a bucket would violate Design Principle P3, and hence be expensive in terms of energy and latency. Thus, instead, all pages of a block are dedicated to a single B-File bucket, as shown in the bottom half of Figure 7, where each shaded (pink) rectangle in the bottom half depicts a block and is labeled with its associated bucket name. Unshaded rectangles in the bottom half depict free blocks.

There is also a crucial choice as to how all the blocks for a bucket are organized. In RAM or magnetic disk, there are a variety of possible organizations (array, stack,

queue, singly- or doubly-linked list, etc.) that may be desirable depending on the context. However, on flash, certain organizations can be extremely expensive to maintain. For example, suppose a bucket were organized as a data structure with forward pointers (i.e., pointers from older elements to newer elements), such as a queue or a doubly-linked list. Older elements on the flash cannot be modified to point to newer elements without incurring high costs (Design Principle P1). An array, although efficient, is not a suitable choice because the precise size of a bucket cannot be determined a priori (see [15] for a discussion on the complexities of handling sampling variance in geometric file). A flat in-memory table that maps blocks to buckets is not attractive either, for its large memory footprint and inefficient bucket to block mapping. Thus, instead, we will chain the blocks of a bucket together with backward pointers (i.e., newer blocks point to older blocks), as depicted in the figure.

B-File uses two modules to maintain this layout, described next.

Bucket manager The Bucket Manager (BM) writes in-memory buffers to flash pages. When the buffer holds one page worth of data, the buffer is flushed to the next available page within the block h, as indicated by the block pointer. When no empty page is available in that block h, a new block h' is allocated by the Storage Manager (described below). A pointer to the block h is stored with the last page of block h' and the block pointer is updated to h'. Thus the blocks in a bucket are chained together with backward pointers and the address of the last block is maintained in the block pointer.

Storage manager The Storage Manager (SM) tracks the available blocks and allocates them to the Bucket Manager (BM) on demand. When BM discards a bucket, the block pointer of the bucket is returned to SM. Moreover, when the tail bucket B_T is unrolled (described in Section 5.2.2), the blocks used by B_T are also reclaimed by SM. When BM requests a new block, SM pops a block from a discarded bucket, erases it, and returns it to BM.

Note that because blocks are allocated dynamically to individual buckets, B-File can handle variable-size records. However, to simplify our cost analyses and parameter optimizations, we consider fixed-size records in the rest of this paper.

5.2.2 Maintaining the Tail Bucket

Note that the tail B-File bucket B_T is essentially a log of items with different levels, all of which are larger than the item levels in individual B-File buckets. Because items are discarded one level at a time, at some point the log must be scanned in order to separate out items with certain levels. We call this process *unrolling* B_T. For example, suppose $N = 10$ and $L = 3$. Then, all the items with level ≥ 13 are kept in B_T. The reason we decide to maintain these levels in one bucket is that very few items so far have these levels (with geometrically distributed i.i.d. levels, the numbers decrease exponentially with the level), and so maintaining a separate bucket (which must be at least one block in the flash) for each such level is wasteful. However, as more items arrive, level 13 becomes more frequent within B_T and at some point it may make sense to maintain a separate bucket for level 13. Separating level 13 items from B_T would make it easier to discard the level 13 items when $L = 13$ and $|bfile| = s_{max}$. Note that after unrolling, separated buckets can be accommodated within the individual buckets, because at least one individual bucket is discarded between any two unrollings.

Unrolling B_T requires reading all its items, writing items to be separated out into their appropriate buckets, writing the remaining items into a new B_T, and then freeing the old B_T. (We cannot update B_T in place since flash does not allow it.) This is the only occasion where we write the same item more than once to flash—there is no other such copying overheads in Algorithm 1.

One important design decision is when to unroll B_T in order to separate out one or more buckets from it. This decision can significantly affect the performance of the sampling algorithm. After each unrolling, all $N + 1$ buckets contain items. Now, on the one hand, B_T can be unrolled every time B_1 is discarded. This is feasible because discarding B_1 gives free space that can be used to unroll B_T. This has the advantage that B_T cannot grow very long before unrolling, keeping the cost of scanning it small. On the other hand, B_T can be unrolled lazily. In the extreme, it can be unrolled only when necessary, e.g., when discarding items of the lowest level in B_T, or when processing queries involving items in B_T. This has the advantage that B_T can be unrolled very infrequently, which may save the unrolling cost.

In general, suppose the algorithm maintains at most $N + 1$ buckets and B_T is unrolled after every u times B_1 is discarded; i.e., just before unrolling B_T, there are $(N - u)$ individual B-File buckets. (The two extreme scenarios above correspond to $u = 1$ and $u = N$). We now study the following optimization question: *What values of N and u optimize the cost of maintaining the sample?* This analysis will assume a level function that generates geometrically distributed i.i.d. levels, with α set according to Lemma 2.

Cost analysis Suppose the costs of reading and writing a data item to flash are R and W, respectively. For example, if y items can be stored in a flash page, the cost of writing a flash page is c_w, a block contains z pages, and the cost of erasing a block is c_e, then $W = (c_w + c_e/z)/y$. Suppose, the expected size of the largest bucket B_1 before it is discarded is s_1, and hence on expectation, s_1 items are inserted into the sample between two successive bucket discards. Thus, $u \cdot s_1$ items are inserted into the sample between two log unrolls. For these us_1 items, we incur the following I/O costs.[1]

1. All us_1 items are written (to individual buckets or to B_T), incurring a cost of $c' = us_1 \cdot W$.
2. The whole B_T needs to be read during unroll. Note that, just before unroll, there will be $N - u$ active buckets, and the expected size of B_T will be $s_T = \sum_{i=N-u}^{\infty} s_1 \alpha^i = s_1 \alpha^{N-u}/(1-\alpha)$, where $\alpha = s_{min}/s_{max}$ as before. Hence, reading B_T will incur a cost of $c'' = s_T \cdot R$
3. The items in B_T need to be written back, either to individual buckets or to a new B_T, incurring a cost of $c''' = s_T \cdot W$. In the special case $u = N$, the items with the smallest level in B_T can be discarded, and hence the cost would be $c''' = (s_T - s_1) \cdot W$

Thus the total cost *per item included in the sample* is

$$C = \frac{c' + c'' + c'''}{us_1} = W + \frac{(R+W)\alpha^{N-u}}{u(1-\alpha)}$$

Optimal values of N and u The above equation shows that the cost of maintaining the buckets decreases with increasing N. Intuitively, having a large N implies a smaller B_T and a small log unrolling cost. Therefore, it is preferable to have N be as large as possible. However, the size of the data structures in memory increases linearly with N. Moreover, the write performance degrades whenever $N_{open} = N + 1 > N_{MaxOpen}$. Hence, in practice, N is upper bounded based on these two considerations.

The cost equation also shows that, for a given N, the above cost function is convex in terms of u. Hence, the cost is minimized when the derivative $dC/du = 0$. This yields the following lemma.

Lemma 3 *Consider Algorithm 1 run with geometrically distributed i.i.d. levels such that its* B-File *maintains at most N individual buckets and B_T is unrolled after every u times that B_1 is discarded. Then the cost of maintaining the buckets is minimized when*

$$u = -1/\ln(\alpha) \ ,$$

where $\alpha = s_{min}/s_{max}$.

Our B-File implementation uses the above result to determine how often the tail bucket B_T is unrolled.

5.2.3 B-File Sizes

The size of a B-File ranges between two user-specified bounds s_{min} and s_{max}. Interestingly, there exists a non-trivial interaction between the cost of maintaining samples in a B-File, and the difference $\delta = s_{max} - s_{min}$. Consider a fixed N. Intuitively, a large value of δ is not desirable, since buckets are discarded less frequently and more items are added to the B-File (some of which are discarded later). A small value of δ is not desirable either, because then the tail bucket contains a large number of items, increasing the cost of log unroll. If a user has the freedom to choose a value of s_{max} (or s_{min}) for a given s_{min} (or s_{max}, respectively), the value must be chosen carefully to balance the trade-off.

We here briefly outline how to determine the optimal s_{max} given an s_{min} (or vice versa) if the stream size $|S'|$ (or an approximation of the size) is known a priori. As before, we assume geometrically distributed i.i.d. levels. Suppose the B-File is configured to maintain N individual buckets. Then, it is possible (applying Lemma 3) to compute

- t: the expected number of times the tail bucket is unrolled,
- s_T: the expected size of the tail bucket just prior to an unroll, and
- a: the expected number of items added to the B-File,

all as functions of s_{min}, s_{max}, $|S'|$, and N. Then, the total cost of maintaining the sample can be computed numerically as $C = a \cdot W + t \cdot s_T \cdot (R + W)$, where R and W are as defined in Section 5.2.2.

One can use the above cost function to search the design space of s_{min} and s_{max} for combinations that minimize the total cost. Our experiments show that for a given s_{min}, the cost function is convex with a single minima, thus the optimal s_{max} can be found by a simple binary search.

The cost function depends only logarithmically on the stream size $|S'|$. Thus, a very loose approximation of $|S'|$ suffices. Note also that, if desired (e.g., when $|S'|$ is not known a priori), one can increase s_{max} on-the-fly as the algorithm runs without sacrificing correctness:

[1] We here ignore the CPU cost of our algorithm because first, it is negligible compared to the flash I/O cost, and second, it does not affect the key parameters we seek to optimize.

Algorithm 2 *GetNext*()

```
1: while true do
2:     r = RAND(1, N + 1) {Select a random bucket}
3:     j = RAND(1, b*)   {b* is the size of the largest bucket}
4:     if j ≤ |B_r| then
5:         Return the j'th item in B_r
6:     end if
7: end while
```

As long as p is unchanged (which implies s_{min} also increases), the assignment to levels is unchanged and the sample will eventually grow to the new s_{max} using the current L.

6 Querying the Sample

In this section, we describe efficient techniques for extracting a subsample Q from the sample S generated by Algorithm 1, under two important scenarios. First, in Section 6.1, we seek a smaller random sample of the data stream than the one generated by Algorithm 1. When they suffice for the estimation problem at hand (e.g., the variance is low), smaller random samples are preferred because they cost less time and energy to transmit and process. Second, in Section 6.2, we seek a random sample of the portion of the data stream that arrived during an arbitrary query-specified time window. Such queries are common when remotely querying energy-constrained sensor nodes. In both scenarios, the key parameter—the sample size or the time window—is specified only at query time.

Caveat: As in all query processing scenarios that rely on a single stored sample to answer queries, the answers across queries (with overlapping time windows) will be correlated because they are all based on the same stored sample.

6.1 Random Subsampling

We first present techniques for choosing a simple random sample Q (of some target size m) from the on-flash sample S such that $m < |S|$. The most obvious way to implement such a sampling would be to use a reservoir sampling algorithm to draw a sample of size m from S. However, although simple, this naive algorithm has two major drawbacks. First, it would require scanning the entire sample S, which can be several gigabytes or more. Second, it would require $O(m)$ space in the memory, which may not be feasible in many memory-constrained devices. Instead, we develop techniques that exploit the randomized bucket structure of the B-File generated by Algorithm 1.

6.1.1 Iterative Sampling

We first consider *iterative sampling*, where we extract a single sample point from S at a time (i.e., $|Q| = 1$), with replacement. An estimator can use iterative sampling until a sufficiently high accuracy is achieved, adapting to the variance in the sampled data (the lower the variance, the smaller the required sample size). Algorithm 2 shows an iterative algorithm that uses an acceptance/rejection test, like [24], to produce a random sample from a B-File. Although many loops may be required before an acceptance, accessing the flash is required only on an acceptance (assuming the $N + 1$ bucket sizes are cached in memory). Note that to access the selected item in flash (line 5), one must traverse the chain of blocks of the corresponding bucket. We later discuss techniques to reduce the number of pointers required to follow in order to locate the selected item (Section 6.2), and the number of page reads required to extract the pointers (Section 7.3).

6.1.2 Batch Sampling

Because of the low efficiency of iterative sampling, however, we also consider *batch sampling* (i.e., $m = |Q| \gg 1$), which is less adaptive (the estimator must commit a priori to a batch size) but far more efficient. Each batch is a simple random sample. One possible approach would be to adapt Olken and Rotem's procedure of batch sampling from a hashed file [24]. The basic idea is first to determine how many samples need to be drawn from each bucket (using a multinomial distribution), and then to draw the target number of samples from each bucket with the acception/rejection algorithm or the reservoir sampling algorithm. However, this approach suffers from the overheads of extracting random items from each bucket. For example, when the expected number of sample points per page is around 1, then often entire pages are read from flash in order to extract a single sample point from the page. Instead, we can exploit our randomized bucket structure to develop an approach that uses *all* the sample points on most of the pages it reads from flash, as described next.

Consider Algorithm 1 run with a level function that generates i.i.d. levels. Let $\mathcal{B}$ be an arbitrary set of one or more B-File buckets. Then the items in $\mathcal{B}$ are a simple random sample Q of the data stream S'. If we can find a set of buckets that added together have the desired size $|Q| = m$, we can return the items in those buckets. On the other hand, if we must take only part of one bucket in the set in order to achieve size m, then we must be careful to ensure that the part is indeed random. Taking a prefix will not work, because the items in a

single bucket are in arrival time order. Instead, we use reservoir sampling on that one bucket, as follows.

1. Select a few buckets $\{B_{i_1}, B_{i_2}, \ldots, B_{i_k}\}$, where each $i_j \in [1, N+1]$ is a distinct integer, such that

   $$\sum_{j=1}^{k} |B_{i_j}| \geq m \text{ and } \sum_{j=1}^{k-1} |B_{i_j}| < m. \quad (1)$$

   That is, only a fraction of the last bucket B_{i_k} needs to be selected to have m items in all. The bucket selection must be done independently of which particular items are in which buckets.
2. Sample Q', a random set of $(m - \sum_{j=1}^{k-1} |B_{i_j}|)$ items, from B_{i_k}, using reservoir sampling. Return $Q = \cup_{j=1}^{k-1} B_{i_j} \cup Q'$ as the target subsample.

Lemma 4 *Consider the* B-File *buckets produced by an arbitrary run of Algorithm 1 using a level function generating i.i.d. levels on a stream S' seen so far. Consider any run of the above batch sampling algorithm on these buckets, with a target sample size m (smaller than the sum of the* B-File *bucket sizes). Then Q is a simple random sample (without replacement) of size m of the items in S'.*

Proof We must show that our combination of sampling procedures yields a simple random sample (SRS). Let n be the number of items in S'. Let $\mathcal{B} = \{B_{i_1}, \ldots, B_{i_k}\}$ be the buckets selected in step 1 of the batch sampling algorithm. Let A be the union of the items in $\mathcal{B}$, and let $m' = |A|$. Let X be the items in B_{i_k} that are *not* selected for Q', i.e., $Q = A - X$.

First, note that because items are assigned to buckets independently (i.i.d. levels) and buckets are selected for $\mathcal{B}$ (and specifically as B_{i_k}) independently of which particular items are assigned to a bucket, we have (i) A is an SRS of S' of size m', and (ii) B_{i_k} is an SRS of A.

Second, because reservoir sampling outputs an SRS, the complement of its output is also an SRS. Thus, (iii) X is an SRS of B_{i_k} of size $m' - m$.

Third, because an SRS of an SRS of a set is itself an SRS of the set, it follows from (ii) and (iii) that (iv) X is an SRS of A of size $m' - m$.

Finally, we consider the probability of returning a given Q. Set Q is returned if and only if A is a superset of Q and $X = A - Q$. There are $\binom{n-m}{m'-m}$ supersets of Q of size m'. By (i), each occurs with probability $1/\binom{n}{m'}$. By (iv), the precise X such that $X = A - Q$ for the choice of A occurs with probability $1/\binom{m'}{m'-m}$. Thus, Q is returned with probability

$$\frac{\binom{n-m}{m'-m}}{\binom{n}{m'}\binom{m'}{m'-m}} = \frac{1}{\binom{n}{m}},$$

which completes the proof. □

The cost of the above algorithm depends on the size of bucket B_{i_k} selected in step 1, because the reservoir sampling in step 2 takes $|B_{i_k}|$ time. Thus, one would like to select the smallest bucket that satisfies the inequalities in (1). We use a greedy heuristic for selecting buckets in step 1: we consider all B-File buckets in increasing order of their size, including them until the inequalities in (1) are satisfied, then removing the smallest buckets as long as (1) remain satisfied, and finally, designating the smallest remaining selected bucket to be B_{i_k}. Under the experimental setup described in Section 8, the size of the smallest bucket selected by this approach is, on average, within 12% of the size of the smallest B-File bucket. The size of the smallest bucket is clearly a lower bound on the cost for the best possible bucket selection.

One caveat is that because the greedy heuristic selects buckets for Q deterministically, the same subsample (up to the random choice of items from B_{i_k}) is selected each time the procedure is called for a given m. When $m \ll |S|$, this issue can be mitigated somewhat by altering the heuristic to consider buckets in a random order, at a potential cost of increasing the size of B_{i_k}.

6.2 Samples Within a Time Window

Given arbitrary t_1 and t_2 at query time, $t_1 < t_2$, our goal is to return a simple random sample of the items in the part of the original stream that arrived within the time window $[t_1, t_2]$. For the purposes of this section, we assume that each item in S is labeled with its timestamp. It is easy to show that all the items in S whose arrival timestamps are in $[t_1, t_2]$ satisfy our goal.

A naive approach to find the desired subset of items is to scan all the buckets in the B-File and return the items with the desired timestamps. However, we can do much better by exploiting the fact that B-File fills page buffers and flushes them to flash in such a way that scanning through the chained set of blocks in a bucket visits the items in descending timestamp order. Therefore, we just require a suitable data structure to locate, for each bucket, its most recent item I_0 with

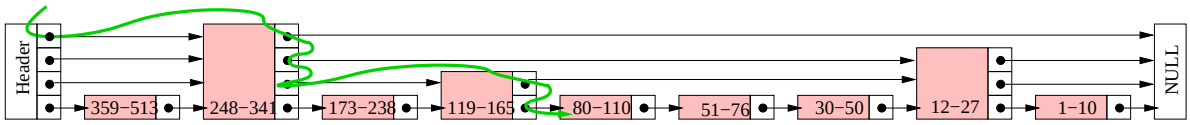


Fig. 8 Blocks of a bucket are organized as a skip list. Solid rectangles are blocks and the text within each solid block denotes the time window of the items stored in that block. Items are stored in descending timestamp order.

Algorithm 3 *InsertBlock*(Block f_b, Bucket m_bkt)

Require: $m\_bkt.header.forward[i]$ initialized to $NULL$ for all $i \in [1, MaxLevel]$; $m\_bkt.level$ initialized to 0

1: $lvl \leftarrow RandLevel()$
2: **if** $lvl > m\_bkt.level$ **then**
3: $\quad m\_bkt.level \leftarrow lvl$
4: **end if**
5: **for** $i = 1$ to lvl **do**
6: $\quad f\_b.forward[i] = m\_bkt.header.forward[i]$
7: $\quad m\_bkt.header.forward[i] = f\_b.address$
8: **end for**

Algorithm 4 *Search*(Bucket m_bkt, Time t_1, Time t_2)

1: $x \leftarrow m\_bkt.header$
2: **for** $i = m\_bkt.level$ **downto** 1 **do**
3: $\quad$ **while** the first item in block $x.forward[i]$ has timestamp $> t_2$ **do**
4: $\quad\quad x \leftarrow x.forward[i]$
5: $\quad$ **end while**
6: **end for**
7: $x \leftarrow x.forward[1]$
8: Binary search block x for the page p containing the item I_0 with the largest timestamp $\leq t_2$
9: Sequentially read the bucket starting from page p, for as long as the timestamp is $\geq t_1$; if needed jump to the next block by using $forward[1]$ of the current block

a timestamp $\leq t_2$. We can then sequentially scan the bucket for as long as we find items with timestamps $\geq t_1$.

To facilitate quickly locating I_0, we organize blocks within a bucket as a *skip list* [26]. A skip list is an ordered linked list with additional forward links, added in a randomized way with a geometric/negative binomial distribution, so that a search in the list may quickly skip parts of the list. In terms of efficiency, it is comparable to a binary search tree ($O(\log n)$ average time for most operations, under the standard RAM model). Figure 8 shows an example bucket as blocks organized as a skip list.

Implementing a general skip list, which allows inserting items in the middle of the list, would be expensive in flash. For example, consider inserting a node (a block) with time range $[112, 117]$ into the skiplist in Figure 8. This would require changing forward pointers of some of the existing skip list nodes. Because these pointers cannot be updated in place, these nodes, with pointers to the new node, must be written to new locations. However, this would require updating forward pointers of nodes that point to the updated nodes, and so on. Thus, recursively, many nodes would be required to be written to new locations due to a single insertion operation. Similarly, a deletion operation can be very expensive.

Fortunately, blocks in a bucket of B-File are always inserted at the front of the bucket and inserting a new node at the front of a skip list can be efficiently implemented in flash. Algorithm 3 depicts the steps to insert a new block at the front of a bucket. In memory, each bucket maintains a header that keeps $maxLevel$ number of *forward* pointers. (Here, "level" refers to the skip list pointers, and is not to be confused with the notion of level in Algorithm 1.) To insert a block into the list, a level, lvl, is generated for it such that all blocks have level ≥ 1, and a fraction p (a typical value for p is $\frac{1}{2}$) of the nodes with level $\geq i$ have level $\geq (i+1)$. (See [26] for more details.) For each level i, the bucket header maintains the most recent block with level $\geq i$. For each level i up to lvl, the new block copies the level i pointer from the bucket header into its level i pointer and then writes a pointer to itself as the new level i pointer in the bucket header. Thus, inserting a block requires writing to just the bucket header and the first page of a new block, both of which are in memory. This takes constant time.

Searching for items having timestamps within a time window uses a combination of skip search and binary search (Algorithm 4). Skip search is used to locate the block containing I_0 in logarithmic time, as follows. Starting from the header of the bucket, we search for a block by traversing forward pointers that do not overshoot the block containing the item with timestamp t_2 (recall that items are sorted in descending order of timestamps). When no more progress can be made at the current level of forward pointers, the search moves down to the next level. When we can make no more progress at level 1, we must be immediately in front of the block that contains the desired item (if it is in the list). The green/gray curvy line in Figure 8 shows the search path for locating the block containing timestamp 90. After we locate the block, we use binary search to locate the page that contains the most recent item with timestamp $\leq t_2$. After locating the page, subsequent pages

are read sequentially from the same block. If the last page of the block does not contain a timestamp $< t_1$, the read continues from the first page of the next block of the bucket (the pointer $forward[1]$ of a block gives the next block of the bucket). The scan halts as soon as a timestamp $< t_1$ is encountered.

7 Extensions to the Basic Algorithm

In this section, we present a few extensions to our basic sampling algorithm with B-File.

7.1 Weighted Sampling

Thus far, we have described how the B-File can be used to efficiently maintain a very large unbiased random sample. Our algorithm guarantees that each item produced by the stream has an equal probability of being sampled. In many applications, however, the relative importance of the data items to be sampled is not uniform, in which case the random sample should overrepresent the more important records. Such *weighted sampling* is desirable in many sensor network applications where different sensed events have different importance. The database literature also contains many applications of weighted sampling [2,6].

In this setion we present a weighted sampling algorithm where each item i in the stream has a weight $w_i \geq 1$, and at a given point in time, the probability that the item is included in the sample is proportional to w_i. Interestingly, to ensure this property, the only thing we need to change in Algorithm 1 is how the level of an item is generated—the rest of the algorithm remains the same.

Recall that, for unbiased sampling, the level function must generate i.i.d. levels, and we use a coin tossing experiment to generate levels according to a geometric distribution. Let p be the probability of heads and let $\widehat{l}_u$ be the outcome of the coin tossing experiment (i.e., the number of coin tosses required to get the first head). Then, if we assign $\widehat{l}_u + \log_{1/(1-p)} w_i$ as the level of an item i with weight w_i, the level function has the following desired weighting property:

Lemma 5 *Suppose each item i is assigned a level $\widehat{l}_w = \widehat{l}_u + \log_{1/(1-p)} w_i$, where $w_i \geq 1$ is the weight of item i and $\widehat{l}_u$ is an independent, geometrically distributed random variable with parameter p. Then for all items i and all levels $L \geq 1$, $Pr\{\widehat{l}_w \geq L\} = w_i(1-p)^{L-1}$, independent of all other items.*

Proof $\Pr\{\widehat{l}_w \geq L\} = \Pr\{\widehat{l}_u \geq L - \log_{1/(1-p)} w_i\} = (1-p)^{(L+\log_{(1-p)} w_i - 1)} = w_i(1-p)^{L-1}$. □

Thus, Algorithm 1 with this level function maintains a sample S such that at any point in time, the probability that the item i is in S is proportional to w_i. Note that when $w_i = 1$, we have $\widehat{l}_w = \widehat{l}_u$, and hence the level function is the same as in unbiased sampling.

One caveat is that the level of an item must be an integer. This is achieved if the weights are restricted to powers of $z = \frac{1}{1-p}$, i.e., $1, z, z^2, z^3$, etc. Otherwise, $\widehat{l}_w$ may be fractional. A heuristic for dealing with such cases is to probabilistically round each w_i up or down to the next power of z, so that the expectation of its weight is w_i. More precisely, let $j = \lfloor \log_z w_i \rfloor$, so that $z^j \leq w_i < z^{j+1}$. We round up to z^{j+1} with probability $(w_i - z^j)/(z^{j+1} - z^j)$, and otherwise we round down to z^j.

7.2 Age-Decaying Sampling

Another important type of sampling is where the probability of an item to be included in the sample decays with its age; i.e., at any point in time, the sample includes more newer items than older items. Consider, for example, the problem of sensor data management—most queries will be over recent sensor readings. Another example is sampling-based techniques for network intrusion detection where recent events are more important than older events.

We here present a sampling algorithm where the most recent item is always in the sample and the probability that an item is included in the sample decays exponentially with its age. We define age age_i of an item i as the number of items in S' that arrived after i.[2] In our algorithm, the inclusion probability of items decays in discrete steps. More precisely, the inclusion probability of items stays the same for every s_1 item arrivals, where s_1 is the expected size of B_1 in B-File. Thus, the inclusion probability of an item i exponentially decreases with the number of item groups of size s_1 that arrived after i. Although this does not provide a smooth decay, this is acceptable in many practical scenarios. For example, it is perfectly fine for many applications to maintain a sample where all the items that arrived today have the same inclusion probability p_0, all the items that arrived yesterday have the same probability $p_1 < p_0$, and so on. In such a case and with a constant daily arrival rate, our algorithm can be used with a value of s_1 such that s_1 items arrive each day.

[2] This is in contrast to the definition of age in terms of time elapsed after the item i has arrived. Within our sampling framework, techniques for exponentially-decayed sampling with time-based age is still open. One can use weighted sampling with weight = arrival time, but as timestamps grow large, the decay becomes very slow.

Table 3 Generating levels for different sampling algorithms

Sampling scheme	Level of new item
Unbiased sampling (simple random sample)	$\widehat{l_u}$ = # tosses of a p-biased coin to get the first head
Weighted sampling	$\widehat{l_w} = \widehat{l_u} + \log_{1/(1-p)} w$
Exponentially decayed	$\widehat{l_e} = \widehat{l_u} + L - 1$
Weighted + Decayed	$\widehat{l_{we}} = \widehat{l_u} + \log_{1/(1-p)} w + L - 1$

As before, this sampling algorithm also requires generating the levels of newly arrived items in a special way, while everything else of Algorithm 1 remains the same. Now, on arrival of an item i, we assign it a level $\widehat{l_e} = \widehat{l_u} + L_i$, where $\widehat{l_u}$ is the level generated by the coin toss experiment for unbiased sampling, and L_i is the minimum active level at the time of the arrival of item i. Then the following lemma shows that our basic algorithm maintains a sample where the inclusion probability decreases exponentially.

Lemma 6 *Suppose an item i is assigned a level $\widehat{l_e} = \widehat{l_u} + L_i - 1$, where $\widehat{l_u}$ is an independent, geometrically distributed random variable and L_i is the minimum active level at the time of the arrival of item i. Then Algorithm 1 maintains a sample S such that at any point in time, (i) item i is in the sample if $L = L_i$ and (ii) the probability that item i is in the sample decreases exponentially with $(L - L_i)$ if $L > L_i$, where L is the current minimum active level.*

Proof Let p be the parameter of the geometric distribution (for the coin tossing experiment, p would be the bias on the coin). Suppose the current minimum active level is L. Then, $\Pr\{i \in S\} = \Pr\{\widehat{l_e} \geq L\} = \Pr\{\widehat{l_u} \geq L - L_i + 1\} = (1-p)^{(L-L_i)}$, as required. □

Note that the above two sampling techniques can be combined to maintain a sample where, at any point in time, the inclusion probability of an item is proportional to its weight and the probability decreases exponentially based on its age. Table 3 summarizes the level generation algorithms for different sampling schemes.

7.3 Optimizations with More Memory

We now propose three optimizations that can be used when more memory is available.

The first optimization reduces the cost of maintaining the sample, by using more buckets. As we mentioned in Section 5.2.2, the cost of maintaining the samples decreases monotonically with increasing number of buckets. This is due to the fact that with increasing number of buckets, fewer items are stored in the tail bucket, reducing the cost of log unrolling. However, each bucket maintains a page buffer, and hence more buckets will have a bigger memory footprint.

Table 4 Costs of different types of I/Os in a Fujifilm XD card flash chip

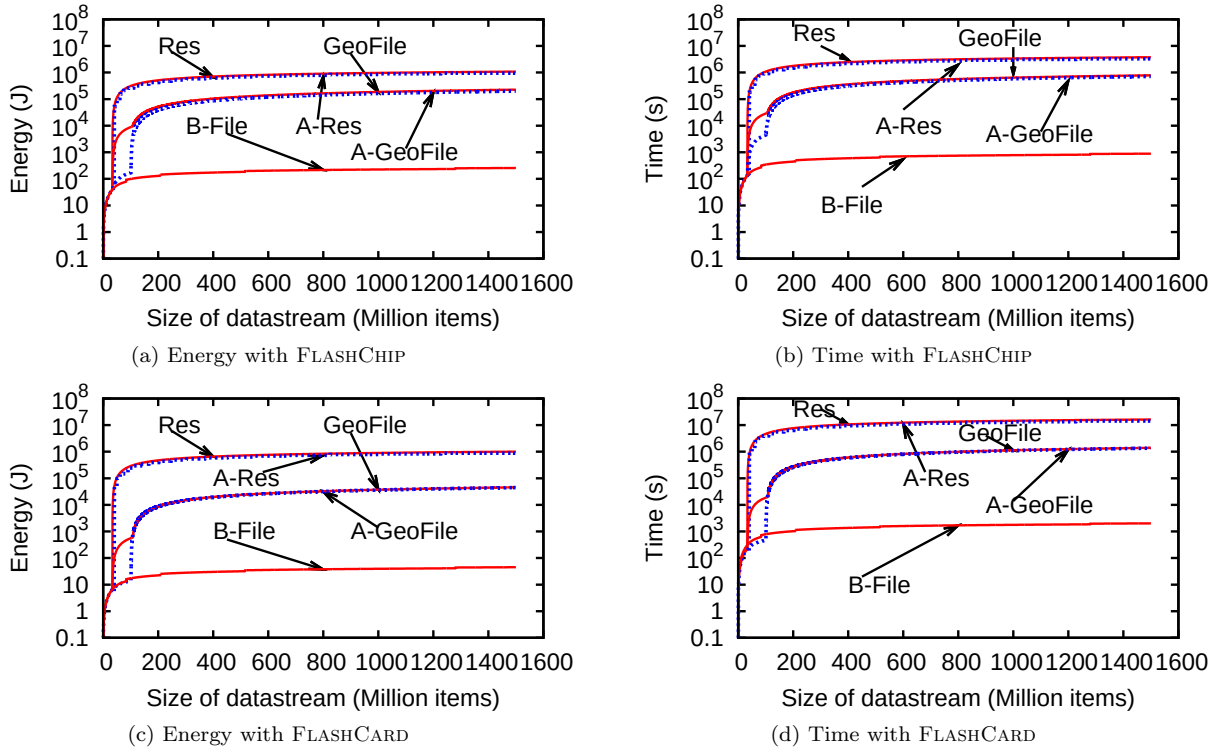
Operation	Latency (ms)	Energy (μJ)
Page read	0.203	34.3
Page write	0.209	34.8
Block erase	3.136	2904

The cost of maintaining the sample can further be reduced by using our second optimization: maintaining part of the tail bucket in memory. Since unrolling the tail bucket requires expensive flash I/O, keeping part of the bucket in memory can reduce this overhead.

Our third optimization reduces the cost of subsampling. In our original proposal in Section 6.2, skip pointers are stored in the last pages of individual blocks. Hence, if we need to retrieve and follow n skip pointers to locate the first record within the specified time window, we need to read n flash pages. This cost can be reduced by maintaining the skip lists in separate flash pages. In other words, we can pack skip pointers of successive blocks of a bucket in separate pages, instead of storing them in the end of every block. Because a flash page is large enough to hold several hundred skip pointers, and all of them can be read with a single flash page read, this optimization can reduce the number of page reads required to locate the first record. Note that this benefit comes at the cost of using a bigger memory footprint: because pages containing skip pointers must be written in flash at a page granularity, every bucket needs to maintain a page-sized in-memory buffer, in addition to its page buffer, for accumulating skip pointers before writing them to flash. The buffer can be flushed to the bucket's currently open block so that a flash block may contain pages having only skip-pointers and pages having only samples. Because the optimization affects neither the total number of writes nor the total number of open blocks for the B-File, its benefit for subsampling queries comes without increasing the time (or energy) for collecting/maintaining the overall sample.

8 Evaluation

In this section we experimentally compare our B-File-based sampling algorithm with a few existing algorithms and also study the impact of different parameters of a B-File. Most of our results are obtained by running the algorithms on an Intel P4 1.7 GHz PC. We also evaluate a B-File prototype on a resource-constrained embedded device in Section 8.8.



(a) Energy with FlashChip (b) Time with FlashChip

(c) Energy with FlashCard (d) Time with FlashCard

Fig. 9 Energy and time consumed by a flash chip and a flash card for each algorithm, under varying stream sizes

8.1 Experimental Setup

Flash Devices Unless otherwise stated, we use a s_{max} = 1.2GB flash device to maintain a s_{min} = 1GB sample ($\approx$ 32 million items) from a data stream consisting of 1.5 billion 32-byte records. We use two flash devices for our experiments: (1) FlashChip: a Fujifilm 2GB XD card flash chip, and (2) FlashCard: a Lexar 2GB CF card. Each flash page is 2KB and each block contains 64 pages. To measure energy, we connect the flash devices, through a CF Extend 180 Extender Card [28], to the PCMCIA slot of the PC. We then connect a low ohmage (1 Ohm) current sense resistor in series with the extender card and measure the current with an oscilloscope. Table 1 and Table 4 show the energy consumption and latency of both these flash devices. We also studied a few other flash cards from Kingston and SanDisk, flash chips from Toshiba [20], and SSD drives from Samsung (mode: MCBQE32G5MPP-0VA00) and SanDisk (mode; SDS5C-016G-000010), and the conclusions were identical; hence we omit results for those cards and drives here.

Workload We use a datastream coming from a set of sensors deployed in a large Microsoft datacenter, although the performance of a sampling algorithm does not depend on the content of the data items. We synthetically generate the weights of data items for biased sampling and the subsample lengths for subsampling experiments, as we will describe in Sections 8.5 and 8.6.

Algorithms Compared We evaluate the following five algorithms: (1) **Reservoir (Res)**: the original reservoir sampling algorithm [29]. (2) **Adapted Reservoir (A-Res)**: the adapted reservoir algorithm described in Section 3.3. (3) **Geometric File (GeoFile)**: the original geometric file based sampling algorithm [15]. However, to reduce its memory footprint, small segments are maintained in flash as a log, instead of in memory. (4) **Adapted Geometric File (A-GeoFile)**: the adapted geometric file based algorithm described in Section 3.3. Lastly, (5) **B-File (B-File)**: the main algorithm described in this paper. Based on our analysis in Section 5.2.2, we select $u = 5$. Note that, Res does not use any extra flash storage; i.e., 1GB space is used to maintain a 1GB sample. All other algorithms use the extra space ($s_{max} - s_{min}$) to maintain logs and/or to accommodate internal fragmentation.

Memory footprint We configure the B-File to use 15 buckets, and it incurs a memory footprint of 31KB. For Res and A-Res, we use a 2KB (= size of a flash page) buffer to temporarily hold samples before writing them

to the flash. (Increasing the footprint 100-fold has only a few percentage points performance impact.) GeoFile and A-GeoFile require a large in-memory buffer; we use 1MB and later discuss the impact of using an even larger buffer.

8.2 Cost of Maintaining Samples

Figure 9 shows the energy consumed and time elapsed to maintain a random sample from a data stream of varying length with FlashChip and FlashCard, using the different algorithms. Note that the relative performance of different algorithms with FlashChip and FlashCard is the same; moreover, the time consumed by different algorithms is proportional to the energy consumed. Therefore, we restrict our discussion below to energy consumption for FlashCard (Figure 9(c)); our conclusions will also hold for energy for FlashChip and for time for both FlashChip and FlashCard.

Figure 9(c) makes several important points. First, the per-item energy consumed by all algorithms decreases exponentially with the stream size. This is due to the fact that we are maintaining an unbiased sample and fewer new records are included into the existing sample as the stream size increases. Second, compared to Res, A-Res reduces energy consumption by $\approx 10\%$, which comes from the fact that some of the samples that need to overwrite random records in Res are discarded directly from the log in A-Res, avoiding expensive overwrite operations. Third, compared to GeoFile, A-GeoFile reduces energy consumption by $\approx$ 13%, highlighting the benefit of allocating entire blocks for individual segments. The last two points demonstrate the benefit of our adapted algorithms. However, their performance improvements look insignificant compared to the performance improvement of B-File. For a stream of size 1.5 billion records, B-File is 3 orders of magnitude more efficient than the best of all the other algorithms considered. Benefits of similar magnitude are observed for the time elapsed to maintain the sample, and with FlashChip instead of FlashCard.

8.3 Micro-benchmark Results

To better understand the relative performance of different algorithms, we show in Table 5 the total count of various primitive I/O operations incurred by different algorithms after sampling from 1.5 billion data items. A hypothetical optimal algorithm *OPT* would incur at least the cost of sequentially writing the minimum number of pages required to hold all the data items ever added to the sample and erasing the minimum number of blocks required to hold all the items ever deleted (to make space for new items) from the sample. *OPT*'s cost is shown in the last row of the table; this cost is a lower bound for any algorithm. In practice, the lower bound may not be achieved due to the following overheads: **C1)** random writes; **C2)** sub-block granularity deletion or in-place update, which require backing up valid data before and copying it back after the required block erase operation; and **C3)** multiple writes of a data item, because of log compaction. Both **C2** and **C3** increases the number of sequential reads and writes.

Table 5 Number, in millions, of basic operations for maintenance (Seq: sequential, Rnd: random, S-Rnd: semi-random)

	Read		Write			Erase
	Seq	Rnd	Seq	Rnd	S-Rnd	
Res	0	8160	0.52	8160	0	128
A-Res	1.89	7120	2.5	7121	0	111
GeoFile	2880	0	799	4.7	0	10.2
A-GeoFile	2747	0	511	0	0	12.6
B-File	0.4	0	0	0	3.42	0.048
OPT	0	0	2.7	0	0	0.032

Table 5 shows that B-File performs very close to *OPT* (note that semi-random and sequential writes have similar costs), because it follows our three design principles for flash. The additional reads and writes (compared to *OPT*) are due to **C3** overheads incurred while maintaining the tail bucket. In contrast, other algorithms incur significantly high overheads, even when they are allowed to use more memory than B-File. In the rest of the section, we point out the causes behind such inefficiencies of these algorithms.

Res performs poorly because of high **C1** and **C2** overheads. A-Res improves upon Res by reducing **C1** and **C2**, at a cost of a small **C3** overhead, as shown in Table 5 by A-Res's fewer random and slightly higher sequential I/Os than Res. GeoFile and A-GeoFile improve upon Res or A-Res by nearly eliminating **C1**. However, they still significantly suffer from other overheads, as explained below.

Ideally, a GeoFile would perform close to *OPT* if i) most of its segments are multiples of flash blocks (thus, avoiding **C2**), and ii) the total size of segments smaller than a block is small (thus, reducing **C3**, or completely avoiding it by buffering all small segments in RAM). This can be made possible only with a small *decay rate* α for its segments. Unfortunately, GeoFile does not allow one to arbitrarily choose a suitable value of α; rather the value of α is fixed as $(1 - \rho)$, where ρ is the ratio of the amount of main memory and the size of the sample. A small value of α is possible with a relatively

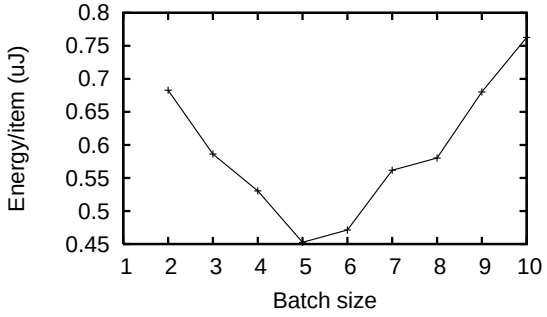


Fig. 10 Effect of batch unroll of B_T

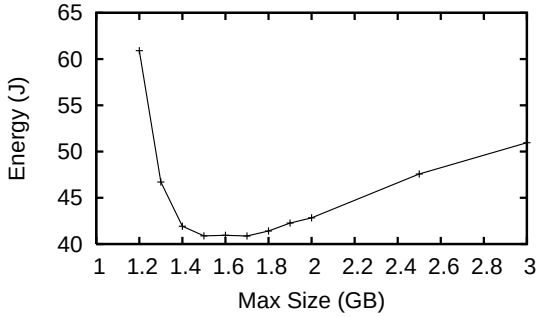


Fig. 11 Effect of s_{max}

large memory size. However, when the target sample size is several orders of magnitude larger than the main memory (i.e., a small ρ), applications are forced to use large values of α. For example, in our experiments, we use 1 MB of RAM for maintaining a 1GB of sample, giving a decay rate of $\alpha = 0.999$. A GeoFile with such a large α has a lot of different segment sizes, with most or all of them being not multiples of a block size. This results in segments being not aligned with flash block boundaries. In our experimental setup, we found that $> 90\%$ of the reads and writes of GeoFile are due to **C2**. GeoFile also incurs **C3** overheads, since the limited RAM of our experimental setup does not allow buffering small segments and overflow stacks associated with each segment in RAM; so they are maintained as a log.

As mentioned in Section 3.2, there is a variant of GeoFile that uses multiple geometric files in parallel. This variant enables using arbitrarily small values of α, and hence can reduce the overhead due to **C2**. However, with limited memory, each file will need to maintain small segments and segment overflow stacks as a log. Maintaining such logs for multiple files together will significantly increase the **C3** overhead. Moreover, the implementation of this approach is too complicated for resource constrained devices. Due to these reasons, we did not implement or evaluate this approach.

Table 5 also shows that A-GeoFile improves GeoFile by reducing **C2**. However, because allocating space at a block granularity in A-GeoFile may waste space and we use only $(s_{max} - s_{min}) = 0.2$GB of extra space, it is not possible to allocate entire blocks to all segments in A-GeoFile. Therefore, A-GeoFile cannot completely eliminate **C2**. Our experiments show that to significantly eliminate **C2** in A-GeoFile, we need to allocate full blocks to a large number of segments; and to afford the resulting internal fragmentation, we need to use $s_{max} = 10$GB instead of 1.2GB.

Note that the performance of both GeoFile and A-GeoFile can be improved by using a larger in-memory buffer; our experiments show that by using a 10MB memory, instead of our default 1MB memory, the performance can be improved by around 10%. In other words, significantly improving the performance of A-GeoFile requires using both a very large (e.g., 10× the sample size) flash and and a large memory (e.g., $> 1\%$ of the sample size).

8.4 B-File Parameters

Figure 10 shows the effect of doing log unroll in batch (unless otherwise noted, from this point on we use energy measure as the cost and FlashCard as the flash storage). As discussed in Section 5.2.2, the cost per sampled item depends on how many unrolls, u, are batched together and there is an optimal value for u. For our particular experimental setup, our analysis in Section 5.2.2 gives $u_{opt} = 5$, same as what we see from our experiment (Figure 10).

Figure 11 shows the effect of different s_{max} with a fixed $s_{min} = 1GB$. As explained in Section 5.2.3, the cost is optimized for a certain value of s_{max}, and for our experimental setup, the optimal value is $\approx 1.5GB$. The numerical analysis outlined in Section 5.2.3 also gives the value $1.5GB$. Note as well that using $s_{max} = 1.5GB$ improves the performance of our B-File algorithm by 33% over the default $s_{max} = 1.2GB$ studied in Figure 9.

8.5 Biased Sampling

Figure 12 shows the cost of maintaining different types of random biased samples on a B-File. For the weighted samples, the weights of individual records have a Gaussian distribution with mean 3 and variance 1 (negative weights are replaced with zero weight). As shown, the cost of maintaining a weighted sample is slightly higher than maintaining an unbiased sample; moreover, the per-item cost exponentially decreases with stream size because fewer items are included into the sample as

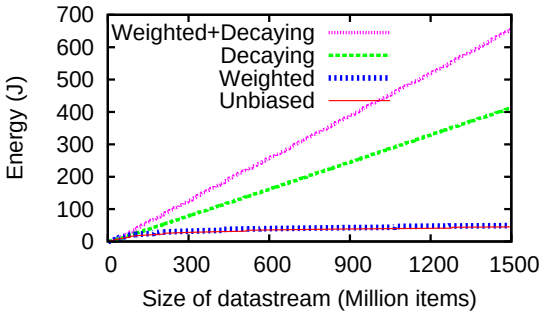


Fig. 12 Biased sampling

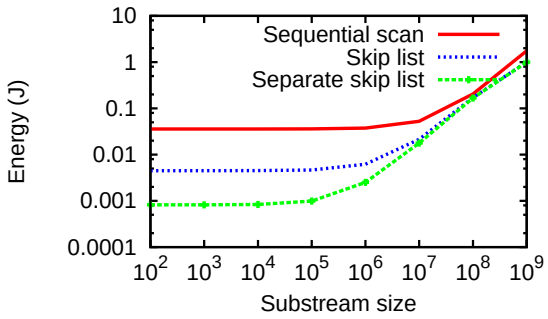


Fig. 13 Energy consumed to subsample

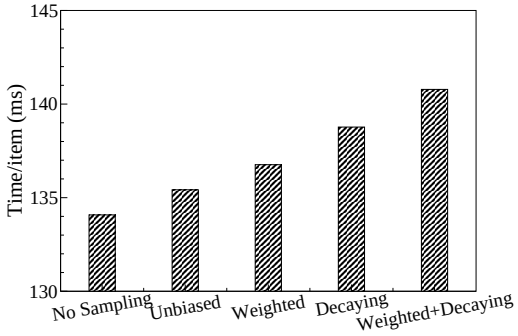


Fig. 14 Cost of subsampling queries concurrent to sampling

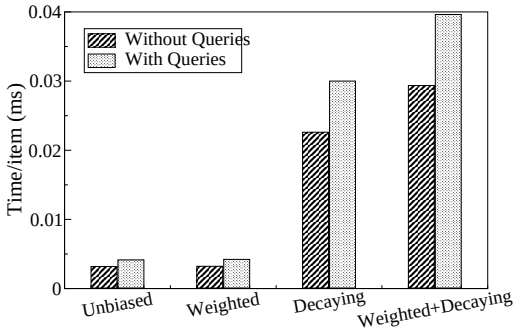


Fig. 15 Cost of sampling concurrent to subsampling queries

the stream size grows. However, maintaining an age-decaying sample is expensive, because every new record needs to be added to the sample (and possibly discarded later as the record grows older). The effect is that the cost increases linearly with stream size, as shown by the Decaying and the Weighted+Decaying curves in Figure 12. Note that the performance difference for B-File and other algorithms (e.g., A-Res and A-GeoFile) will be even greater for age-decaying sampling than for unbiased sampling; all algorithms will add roughly the same number of records into the sample, but adding a new record is much cheaper in B-File than in the other algorithms.

8.6 Subsampling

To evaluate subsampling cost, we first construct a 1GB random unbiased sample from 1.5 billion records, with exponentially distributed inter-arrival times. We then measure the energy consumed to extract all the records in the sample that arrived within a time window $[t_1, t_1 + length]$, where t_1 is uniformly randomly distributed in the window $[0, 1.5 \times 10^9 - length]$. Figure 13 shows the energy consumed for different values of $length$. We consider three alternatives to locate the first record ($\geq t_1$) in each bucket: sequentially scanning the bucket, using skip lists with pointers stored at the end of data blocks, and using skip lists with pointers stored in separate pages (recall Section 7.3). The results show that the cost of extracting subsamples increases with the substream size $length$ (which is proportional to subsample size). For smaller substreams ($< 10^5$), using skip lists provides roughly an order of magnitude greater energy savings than sequential scan, and using the skip lists in separate pages provides roughly another order of magnitude greater energy savings. The benefit comes from the small number of page reads required to locate the first record in the subsample. However, the benefit diminishes as the subsample is taken over a longer substream, as the cost of locating the first record becomes insignificant compared to the cost of reading subsequent records in the subsample.

8.7 Concurrent Sampling and Querying

So far we have evaluated our sampling and subsampling algorithms in isolation. Now, we consider them together; i.e., we query (subsample) the sample on a flash device at the same time the sample is being collected. Because a flash device can support a limited number of concurrent I/Os, concurrently sampling and querying the sample may affect the performance (especially the latency) of both operations. We consider the following scenario: data items arrive at the maximum rate that our most expensive sampling scheme (Weighted + Decaying) can handle, and we run one sampling thread

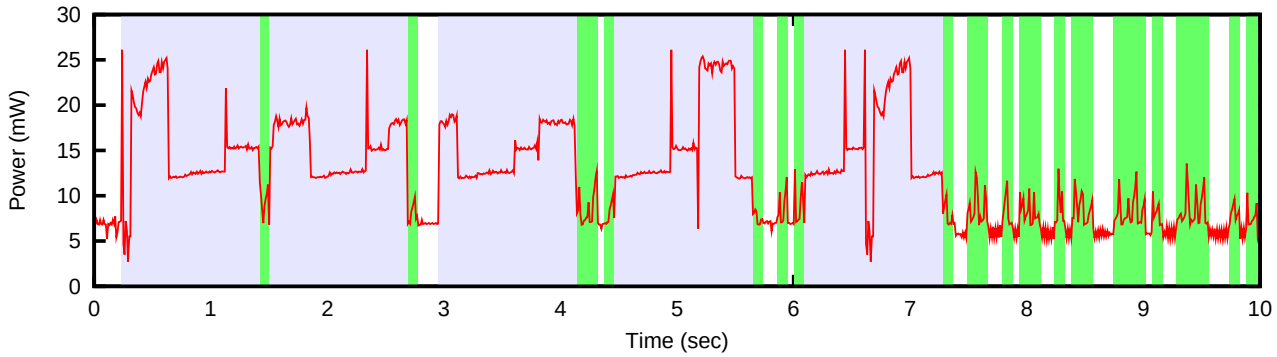


Fig. 16 Energy measurement for the Tmote Sky sensor platform. The graph shows power drawn by a Tmote device in the first 10 seconds of an unroll operation. The white, green (darkly shaded), and blue (lightly shaded) backgrounds denote times when B-File performs read, write, and erase operations on flash, respectively.

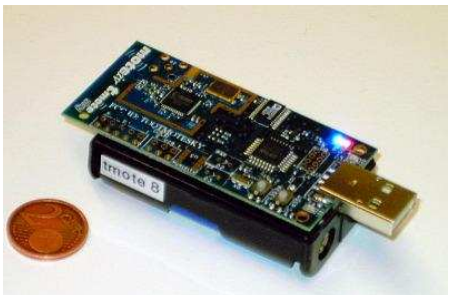


Fig. 17 The Tmote Sky embedded sensor platform

to maintain the sample on flash and 10 query threads each of which continuously asks for random subsamples of size 10,000 items. For the sampling thread, we report the average cost after seeing 100 million(M) data items; the cost would decrease with additional items. (Note that the flash device can hold around 40M items, so even after 100M items, the sampling cost per arriving item is reasonably high.)

Figure 14 shows the average subsampling cost reported by the query threads when the sampling thread uses different sampling schemes. It shows that the impact of concurrent sampling on query latency is very small ($< 5\%$). In contrast, as shown in Figure 15, the impact is more significant on sampling latency (up to 35%). This is because with concurrent reads and writes on flash, write performance suffers more than read performance [1].

8.8 B-File on an Embedded Device

To demonstrate the feasibility of using our B-File algorithm in resource-constrained embedded devices, we have implemented the algorithm on a Moteiv Tmote Sky sensor platform [22], shown in Figure 17, running TinyOS 2.1. The device is suitable for very low power, high data-rate sensor network applications. It has integrated sensors, radio, microcontroller, 10KB RAM, and programming capabilities. The device is also equipped with an ST Microelectronics M25P family flash chip of size 1MB. Each flash page is 512B and each block contains 64 pages. Our prototype is implemented using approximately 500 lines of code in nesC, a version of the C language designed for programming embedded systems. The prototype has around a 14KB ROM footprint and a 1.5KB RAM footprint. It captures data from Tmote's on-board temperature sensor and stores samples in the local flash chip.

TinyOS provides several abstractions to store data in a local flash chip. The LogStorage abstraction allows sequentially appending data to a log, while the BlockStorage abstraction allows accessing random flash blocks. TinyOS also allows the local flash chip to be statically divided into multiple *volumes*, so that different volumes can store different types of data with different storage abstractions. The simplicity of B-File enables us to implement it using the LogStorage abstraction: We statically divide the local flash into multiple volumes and use each volume as a separate bucket of B-File. Within each volume, we use the LogStorage abstraction to append data to existing buckets. In this experiment, we use a B-File with 8 buckets; using more buckets is not useful, since the flash chip contains only 32 blocks. Each record (i.e., sensor reading) is 4 bytes and we use $s_{max} = 1024KB$ and $s_{min} = 800KB$.

Our key findings are as follows. First, a block erase, a page read, and a page write consume $15.7mJ$, $0.32mJ$, and $0.78mJ$, respectively. The latencies of these three operations are $1.18s$, $0.06s$, and $0.09s$, respectively. Second, unrolling the tail bucket takes around 86 seconds, and consumes $\approx 758mJ$. Figure 16 shows the timeline of energy consumed in the first 10 seconds of an unroll operation. It shows different flash I/O operations,

as well as the power drawn for those operations. More precisely, the unroll operation starts by reading records from the tail bucket (time 0s to 0.23s in Figure 16) and writing them to other buckets. Writing to other buckets may first require erasing a block; in Figure 16, time 0.24s to 1.41s represents erasing a block, and time 1.42s to 1.50s represents writing a page in that block. Figure 16 also shows that an erase operation has longer latency and it requires more power than write and read operations.

The unroll operation may appear expensive; but an unroll operation happens only when the B-File runs out of space, and therefore it is an infrequent operation. More precisely, the above experimental setup requires only 8 unroll operations to collect samples from a stream of 1.5 billion records. Thus, the amortized cost (including costs for adding sampled records to buckets and unrolling the tail bucket) per record is only $0.0126\mu J$. For our three types of biased samples, using the parameter settings of Section 8.5, the cost per record is $0.014\mu J$, $0.115\mu J$, and $0.183\mu J$ for weighted, decaying, and weighted+decaying, respectively. We have also implemented Res on a Tmote node; the cost per record is $139.69\mu J$, which is *five orders of magnitude* more expensive than the B-File implementation. We have not considered implementing GeoFile on a Tmote as the algorithm is not useful with Tmote's limited RAM.

The large overhead of an unroll operation in this experiment also comes from our use of a small number of buckets. As mentioned in Section 5.2.2, the cost of maintaining samples increases monotonically with a decreasing number of buckets. Intuitively, with a small number of buckets, the tail bucket accumulates a large number of records before an unroll operation, thus, increasing the unrolling overhead. To demonstrate this, we have also run experiments with a B-File with five buckets and found that its unroll operation takes 171 seconds and consumes $1475mJ$, which is almost twice as expensive as an unroll operation of a B-File with eight buckets. The amortized cost per record is $0.02\mu J$, which is 58% higher than the cost with eight buckets.

9 Conclusion

In this paper, we have presented the first flash-friendly algorithm for maintaining a very large (100 MBs or more) random sample of a data stream. We proposed B-File, an energy-efficient abstraction for flash media to store self-expiring items and showed how B-File can be used to efficiently maintain a large sample in flash. We also provided techniques to maintain biased samples with a B-File and to query the large sample stored in a B-File for a subsample of an arbitrary size. Evaluation with flash media shows that our techniques are three orders of magnitude (or more) faster and more energy-efficient than existing techniques.

We believe that the B-File is a general abstraction and can be used for many purposes other than sampling. For example, it can be used to archive data and to automatically *age* it, based on arrival time or priority of the data, to reclaim storage space for newly-arriving data (e.g., on a sensor node). Moreover, our study revealed an important subclass of random writes, which we called semi-random writes, that defy the common wisdom to avoid all random writes. We believe that semi-random writes can also be used for many purposes, e.g., it is the write pattern for external memory distribution sort [31]. Moreover, for some algorithms, sufficient write-buffering and scheduling might be able to transform most of the random writes to flash into semi-random writes. Exploring other uses for B-File and semi-random writes is part of our future work.

Acknowledgement

We thank Dimitrios Lymberopoulos for his assistance with our Tmote Sky experiments, Shimin Chen for interesting discussions on flash, and the anonymous reviewers for their detailed comments that helped to improve the paper.

References

1. N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *USENIX Annual Technical Conference*, 2008.
2. B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. In *ACM SIGMOD International Conference on Management of Data*, 2003.
3. A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high-performance flash disks. *SIGOPS Oper. Syst. Rev.*, 41(2), 2007.
4. C. Bobineau, L. Bouganim, P. Pucheral, and P. Valduriez. PicoDBMS: Scaling down database techniques for the smartcard. In *International Conference on Very Large Data Bases (VLDB)*, 2000.
5. L. Bouganim, B. Jónsson, and P. Bonnet. uFLIP: Understanding flash IO patterns. In *Fourth Biennial Conference on Innovative Data Systems Research (CIDR)*, 2009.
6. S. Chaudhuri, G. Das, M. Datar, R. Motwani, and V. R. Narasayya. Overcoming limitations of sampling for aggregation queries. In *IEEE International Conference on Data Engineering (ICDE)*, 2001.
7. Y. Diao, D. Ganesan, G. Mathur, and P. Shenoy. Rethinking data management for storage-centric sensor networks. In *Third Biennial Conference on Innovative Data Systems Research (CIDR)*, 2007.
8. F. Douglis, R. Cáceres, F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber. Storage alternatives for mobile computers. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 1994.

9. C. T. Fan, M. E. Muller, and I. Rezucha. Development of sampling plans by using sequential (item by item) selection techniques and digital computers. *American Statistical Association Journal*, June 1962.
10. P. B. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2001.
11. M. Hachman. New Samsung notebook replaces hard drive with flash. http://www.extremetech.com/article2/0,1558,1966644,00.asp, May 2006.
12. Intel-Corporation. Understanding the Flash Translation Layer (FTL) specification. www.embeddedfreebsd.org/Documents/Intel-FTL.pdf, 1998.
13. J. Janukowicz and D. Reinsel. SSDs: The other primary storage alternative. IDC White Paper, 2008.
14. C. Jermaine, A. Datta, and E. Omiecinski. A novel index supporting high volume data warehouse insertion. In *International Conference on Very Large Data Bases (VLDB)*, 1999.
15. C. Jermaine, A. Pol, and S. Arumugam. Online maintenance of very large random samples. In *ACM SIGMOD International Conference on Management of Data*, 2004.
16. H. Kim and S. Ahn. BPLRU: a buffer management scheme for improving random writes in flash storage. In *USENIX Conference on File and Storage Technologies (FAST)*, 2008.
17. J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho. A space-efficient flash translation layer for compact flash systems. *IEEE Transactions on Consumer Electronics*, 48(2), 2002.
18. J. Lee, S. Kim, H. Kwon, C. Hyun, S. Ahn, J. Choi, D. Lee, and S. H. Noh. Block recycling schemes and their cost-based optimization in NAND flash memory based storage system. In *ACM/IEEE International Conference on Embedded Software (EMSOFT)*, 2007.
19. S.-W. Lee and B. Moon. Design of flash-based DBMS: an in-page logging approach. In *ACM SIGMOD International Conference on Management of Data*, 2007.
20. G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Capsule: an energy-optimized object storage system for memory-constrained sensor devices. In *ACM International Conference on Embedded Networked Sensor Systems (SenSys)*, 2006.
21. P. Miller. SimpleTech announces 512GB and 256GB 3.5-inch SSD drives. http://www.engadget.com/2007/04/18/, April 2007.
22. Moteiv Corporation. Tmote sky platform. http://www.moteiv.com/community/Tmote_Sky_Downloads, 2007.
23. S. Nath and A. Kansal. FlashDB: dynamic self-tuning database for NAND flash. In *ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2007.
24. F. Olken, D. Rotem, and P. Xu. Random sampling from hash files. *SIGMOD Rec.*, 19(2), 1990.
25. P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4), 1996.
26. W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6), 1990.
27. D. Reinsel and J. Janukowicz. Datacenter SSDs: Solid footing for growth. Samsung white paper. www.samsung.com/global/business/semiconductor/products/flash/ssd/pdf/datacenter_ssds.pdf, January 2008.
28. SyCard. CF extend 180 CompactFlash Flexible Extender Card. http://www.sycard.com/cfext180.html, 2008.
29. J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1), 1985.
30. J. S. Vitter. An efficient algorithm for sequential random sampling. *ACM Trans. Math. Softw.*, 13(1), 1987.
31. J. S. Vitter. External memory algorithms and data structures. *ACM Comput. Surveys*, 33(2), 2001.
32. C.-H. Wu, L.-P. Chang, and T.-W. Kuo. An efficient R-tree implementation over flash-memory storage systems. In *ACM International Symposium on Advances in Geographic Information Systems (GIS)*, 2003.
33. Yahoo!-Finance. Zeus-IOPS solid state drives surge to 512GB. http://biz.yahoo.com/pz/070418/117663.html, April 2007.
34. D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar. Microhash: an efficient index structure for flash-based sensor devices. In *USENIX Conference on File and Storage Technologies (FAST)*, 2005.