Online Maintenance of Very Large Random Samples on Flash Storage

Suman Nath
Microsoft Research
sumann@microsoft.com

Phillip B. Gibbons
Intel Research Pittsburgh
phillip.b.gibbons@intel.com

ABSTRACT

Recent advances in flash media 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 media has many unique characteristics that make existing data management/analytics algorithms designed for magnetic disks perform poorly with flash storage. For example, while random (page) reads are as fast as sequential reads, random (page) 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 (100 MBs or more) 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 media 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 media 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.

1. INTRODUCTION

Recent technological trends in flash media 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 media 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, 6, 15]. 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. Indeed, several companies including Samsung and Dell have already launched new lines of laptops containing only flash storage [8]. Several companies including SimpleTech and STec have launched 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 [27, 16]. 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 [21].

Flash media has fundamentally different read/write characteristics than magnetic disks. For example, reading pages at random is 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 media. Efforts to overcome this limitation (such as via a Flash Translation Layer (FTL) [9]) 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 (e.g., 100 MBs or more) 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 on-board 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, ap-

VLDB '08, August 24-30, 2008, Auckland, New Zealand

proximate 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 [11].

There are many 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 media and hence, not surprisingly, they are ill-suited for flash. For example, *reservoir sampling* [23] and *geometric file* [11] 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 2.2 and 7.1). Moreover, simple optimizations of these algorithms in order to make them more flash-friendly are unable to overcome their basic flash-unfriendly structure (details in Section 2.3). 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—which 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 media 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 3). 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*). While geometric file also uses a preselection of the replacement order, B-FILE is much more efficient because, unlike geometric file, 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 standard (RAM) memory 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 functionality, B-FILE's small memory footprint is critical to its suitability for a range of embedded devices.

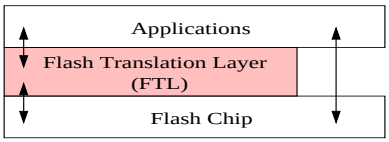


Figure 1: A flash-based storage system

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 media 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 media to store self-expiring items, and show how B-FILE can be used to efficiently maintain a large (guaranteed uniform) random 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. 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 media, 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 presents background and related work. Section 3 discusses semi-random writes. Section 4, 5, and 6 present our basic sampling algorithm, querying algorithm, and several extensions to basic algorithm, respectively. We present evaluation results in Section 7 and conclude in Section 8.

2. PRELIMINARIES

In this section, we discuss flash media characteristics, present related work, and show how the most relevant previous work can be made somewhat more flash-friendly.

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.

Flash Chips. 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 [17]. 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. Our experiments show that an in-place update is over two orders of magnitude more expensive than a write to an erased page (see Appendix B for measurement results.) 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).

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) [9], which is implemented within the micro-controller of the device. 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 Table 1 and Appendix B). Similar to previous work [12, 17], 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.

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.

Design principles for flash algorithms. Because of these characteristics of flash media, algorithms designed for flash should follow a few key well-known design principles, including **(P1) avoid in-place updates** and **(P2) avoid random writes**. Another natural design principle is **(P3) avoid sub-block deletions** (deleting only a portion of a block). Such deletions are over two orders of magnitude slower than block deletions (with or without an FTL), because they require internal copying: any undeleted data in the same block must first be copied to a new block (see Appendix B). Thus, each of these principles can effect both latency and energy consumption by two orders of magnitude or more. In Section 3, we will show that design principle 2 should be modified as follows: **(P2′) avoid random writes unless they are semi-random**.

2.2 Related Work

Algorithms for Flash. Recent studies [1, 3, 12] 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 general enough to support multiple applications, and to recover efficiently 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 [14] and running a light-weight database system on flash-based embedded computing devices [5, 17]. Several other studies have proposed efficient data structures and algorithms for flash storage, including flash-optimized B trees [17], R trees [26], stacks [15], queues [15], and hash tables [28]. These algorithms seek to avoid in-place updates and random writes, 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., SD cards or SSDs), where applications must access the flash through an FTL.

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, 4]) instead of *maintaining* one, as well as existing streaming algorithms that maintain small random samples in main memory (e.g., [7]).

The fastest streaming algorithm for maintaining a large *fixed-size* random sample on a magnetic disk is due to Jermaine et al. [11]. 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. The 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 7). Moreover, in addition to the algorithm being more complex than our proposed algorithm, it has a higher in-memory footprint because (i) small segments 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 [11], the authors also propose using multiple geometric files in parallel for reducing the number of disk head movements. However, on flash, this scheme may not add a significant benefit since it does not reduce the amount of data overwrite and flash devices do not have any mechanical head movements. Moreover, the scheme has a higher space overhead due to higher internal fragmentation and special *dummy segments*. Therefore, we do not consider this scheme in this paper.

Reservoir sampling [23] 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 [11, 24]. Although the original

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

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 7).

Olken and Rotem [18] present techniques for constructing samples in a database environment. However, in addition to not being designed for flash media, 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 [19] and Y-Trees [10] 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 [11], they require more random writes and hence perform worse than geometric file. Therefore, we do not consider them in the rest of the paper.

2.3 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 in Section 2.2) 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.

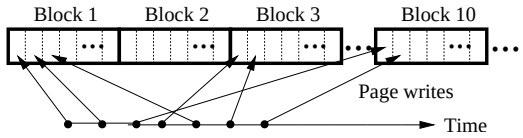


Figure 2: Semi-random writes

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.

The bottom line. The above two algorithms and their adapted versions still require frequent in-place updates. In Section 7, 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.

3. 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. In other words, multiple sequential writes to blocks are interleaved with one another. Figure 2 shows an example, 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.

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 are very efficient. 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 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 benefit of semi-random writes come from 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 [13]. With this strategy as well, semi-random writes do not require copying any unmodified pages across blocks, resulting in superior performance.

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: $L \leftarrow 0$ {L is the current minimum active level}
2: $bfile \leftarrow$ **new** B-FILE(N)
3: **for** each stream item v **do**
4: $l_v \leftarrow$ Level(v, s_{min}, s_{max}) {compute the level}
5: **if** $l_v \geq L$ {if v selected for the sample} **then**
6: $bfile.AddItem(v, l_v{-}L{+}1)$ {append v to bucket 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 bucket B_{i+1} to be B_i}
10: $L \leftarrow L + 1$ {increment the minimum active level}

The good performance of semi-random writes is likely to hold for applications directly accessing flash chips as well. 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 the block boundaries, and hence the block size—the block size can be readily obtained by querying the flash driver or the FTL.

4. MAINTAINING SAMPLES ON FLASH

4.1 The Basic Algorithm

Our basic algorithm (see Algorithm 1) combines ideas from several sampling algorithms (e.g., [7, 11]), in a novel way that is tailored to flash. When describing the algorithm, we will often highlight its adherence to the design principles (P1–P3) from Section 2.1. We will use the notation summarized in Table 2.

At a high level, there are three salient aspects of our basic algorithm. First, as in the adapted algorithms in Section 2.3, 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 lower bound (s_{min}) and a specified 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 uniformly random, 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 4.2) to store the buckets.

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}, s_{max}	Minimum and maximum sample size
α	s_{min}/s_{max}
v, l_v	A stream item and its level
p	Probability of heads in each coin toss
R, W	Avg. cost to read/write an item in flash

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).

Assigning levels to items to obtain overall guarantees. The fact that all items in unexpired buckets constitute a random sample holds as long as the random variable that determines an item's level is independent of its arrival order. As a specific implementation, and in order to have only a logarithmic number of levels, we assign items to levels such that the expected number of items having a level i decreases exponentially with i. Such a random level can be obtained 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.

LEMMA 4.1. *At any point of time, the items with level $\geq L$ (i.e., the items in the buckets), represent a uniform sample of all the items seen so far.*

PROOF. Consider an item v and denote the level assigned to it as l_v. Then Pr $\{l_v = i\} = (1-p)^{i-1}p$. Suppose the current value of L is k. The item will be part of the sample if $l_v \geq k$. Thus Pr $\{v$ is in sample $|L = k\} = \sum_{j=k}^{\infty}(1-p)^{j-1}p = (1-p)^{k-1}$, which is constant for given values of p and k. □

Lemma 4.1 implies that we can maintain a uniform 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 (proof in Appendix A) provides a means to select p in order to keep the sample size within a target range.

LEMMA 4.2. *Setting $p = 1 - \alpha$, where $\alpha = s_{min}/s_{max}$, ensures that the sample size is at least s_{min} on expectation and always at most s_{max}.*

Discussion. At this high level, the algorithm is reminiscent of the sampling component of our previous algorithm [7] for counting the number of 1's in the union of distributed data streams, with mod-

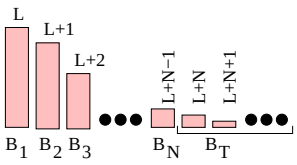


Figure 3: A snapshot of B-FILE. Solid bars represent application buckets, text above a bar represents the level of the items in the bucket, and text below a bucket represents the B-FILE bucket number. The tail B-FILE bucket B_T contains items with level at least $L + N$.

est changes. However, at the next level of detail, the previous algorithm (which is designed for main memory and not flash) suffers from excessive in-place updates, random writes and sub-block deletions, violating flash design principles P1–P3. Our main innovations are (i) in recognizing that the previous algorithm provides a basis for a flash-friendly algorithm, (ii) in designing novel flash-friendly techniques in support of each step of the algorithm (finding the right data organization, etc.), and (iii) in exploring how various parameter choices optimize performance.

4.2 B-File Design

In this section, we present our main new data structure: the B-FILE. Logically, a B-FILE consists of a potentially large set of application buckets $\cup_i B_i$ stored on a flash media. 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. To distinguish these two notions, we will call the former *application buckets* and the latter B-FILE *buckets*. The use of a tail B-FILE bucket is a key optimization for flash, as discussed below.

At a high level, 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 the random sample S over the entire data stream S' seen so far (Lemma 4.1). Figure 3 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 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 4.2.1). These page flushes fit a semi-random access pattern, as defined in Section 3. Namely, while the next flushed page can be for any bucket, the pages within a bucket's

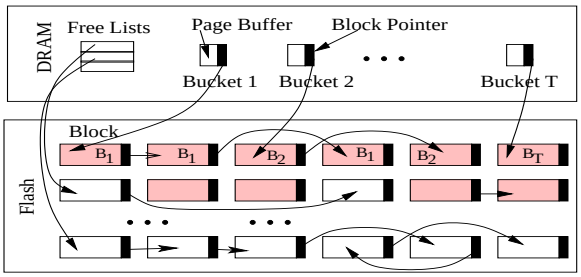


Figure 4: Physical layout of the buckets

block are written sequentially. Thus, according to Design Principles P1 and P2′, the write operations are highly-efficient. 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 the size of the application buckets exponentially decreases with level number, and therefore, 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 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 4.2.2, 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.

4.2.1 Bucket Layout and Maintenance

Figure 4 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 4, 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 [11] 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) keeps track of 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 4.2.2), 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 accommodate large and variable-size records. However, for simplicity of cost analyses and parameter optimizations, we consider fixed-size records in the rest of this paper.

4.2.2 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 (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?*

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, us_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 smaller B_T and 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. Hence, in practice, the desirable memory footprint places an upper bound on N.

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 4.3. *Suppose a* 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 $\alpha = s_{min}/s_{max}$.*

[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.

There also exists a non-trivial interaction between the total I/O cost of a B-FILE and the difference $\delta = s_{max} - s_{min}$. In Appendix A, we show how the optimal value of s_{min} or s_{max} can be determined if the size of the stream can be estimated a priori.

5. QUERYING SAMPLES

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 5.1, 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 5.2, we seek a random sample of the portion of the data stream that arrived during an arbitrary 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.

5.1 Random Subsampling

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

We will focus on *batch sampling*, where the desired sample points are extracted as a batch. In Appendix A, we present an alternative algorithm for *iterative sampling*, where a single sample point is extracted at a time, with replacement.

Batch sampling. One possible approach would be to adapt Olken and Rotem's procedure of batch sampling from a hashed file [18]. 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.

Because in Algorithm 1, all items are equally likely to have a given fixed level, each B-FILE bucket is a uniform random sample of the data stream S'. Thus, any combination of buckets is also a uniform random sample. If we can find a set of buckets that added together have the desired size $|Q|$, 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 $|Q|$, 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 $\sum_{j=1}^{k} |B_{i_j}| \geq |Q|$ and $\sum_{j=1}^{k-1} |B_{i_j}| < |Q|$. That is, only a fraction of the last bucket B_{i_k} needs to be selected to have $|Q|$ items in all. One greedy heuristic for selecting these buckets is to consider all B-FILE buckets in increasing order of their size and to designate the smallest selected bucket to be B_{i_k}.
2. Sample Q', a random set of $(|Q| - \sum_{j=1}^{k-1} |B_{i_j}|)$ items, from B_{i_k}, using reservoir sampling. Return $Q = \cup_{j=1}^{k-1} B_{i_j} \cup Q'$ as the target subsample.

It is easy to show that the above algorithm returns a random sample over the stream. All items have the same probability of being selected for a given bucket; let p_j be the probability that an item is selected for a bucket B_{i_j}. Now consider an item x in the entire stream S'. $\Pr\{x \in Q\} = \sum_{j=1}^{k-1} \Pr\{x \in B_{i_j}\} + \Pr\{x \in B_{i_k}\} \cdot \Pr\{x \text{ selected by reservoir sampling algorithm}\} = \sum_{j=1}^{k-1} p_j + p_k \cdot |Q'|/|B_{i_k}|$. As this expression is independent of x, every item in the stream has an equal probability of being included in Q.

The cost of the above algorithm depends on the size of the smallest bucket selected in step 1. Selecting an optimal bucket set (that will minimize the size of the smallest selected bucket) is part of our future work. Note that a lower bound on the cost for any algorithm is the size of the smallest B-FILE bucket. Under the experimental setup described in Section 7, the size of the smallest bucket selected by our greedy approach is, on average, within 12% of this lower bound.

One caveat is that because buckets are selected 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 $|Q|$.

5.2 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 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 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 efficiently locating I_0, we organize blocks within a bucket as a *skip list* [20]. 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 5 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 5. 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.

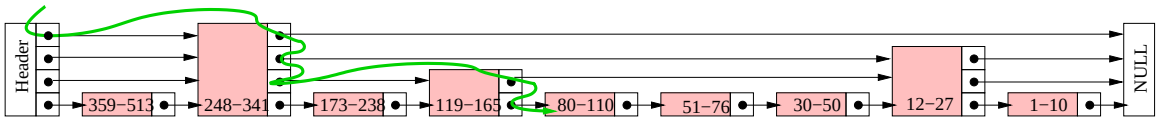


Figure 5: Blocks of a bucket are organized as a skip list. Solid rectangles are blocks and the text within each solid block denotes the time window of the items stored in that block. Items are stored in descending timestamp order.

Algorithm 2 $InsertBlock$(Block f_b, Bucket m_bucket)

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

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

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 a flash. Algorithm 2 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 [20] 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 within a time window uses a combination of skip search and binary search. 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 gray curvy line in Figure 5 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.

6. EXTENSIONS OF BASIC ALGORITHM

6.1 Weighted Sampling

Thus far, we have described how the B-FILE can be used to efficiently maintain a very large unbiased random sample. We have assumed 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 over-represent 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, 4].

We here present a weighted sampling algorithm where each item i in the stream has a weight w_i, 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 uniform sampling, we use a coin with $\Pr(head) = p$ and the level of an item is the number of coin tosses required to get the first head. Let us denote the outcome of such a coin tossing experiment as $\widehat{l}_u$. Then, if we assign $\widehat{l}_u + \log_{(1-p)} w_i$ as the level of an item i with weight w_i, then Algorithm 1 maintains a sample with the desired weighting property:

LEMMA 6.1. *Suppose an item i with weight w_i is assigned a level $\widehat{l}_w = \widehat{l}_u + \log_{(1-p)} w_i$. Then Algorithm 1 maintains a sample such that at any point in time, the probability that the item i is included in the sample S is proportional to w_i.*

PROOF. Suppose the current minimum active level is L. Then, $\Pr\{i \in S\} = \Pr\{\widehat{l}_w \geq L\} = \Pr\{\widehat{l}_u \geq L + \log_{(1-p)} w_i\} = (1-p)^{(L+\log_{(1-p)} w_i - 1)} = w_i(1-p)^{L-1}$. The lemma follows because $(1-p)^{L-1}$ is fixed independent of i. □

The level of an item must be an integer. However, $\widehat{l}_w$ may be fractional. To deal with this, we generate an integer level $\widehat{l}'_w$ which is either $\lfloor \widehat{l}_w \rfloor$ or $\lceil \widehat{l}_w \rceil$, depending the magnitude of the fractional part of $\widehat{l}_w$. More precisely, $\widehat{l}'_w = \lceil \widehat{l}_w \rceil$ with probability $(\widehat{l}_w - \lfloor \widehat{l}_w \rfloor)$, and $\widehat{l}'_w = \lfloor \widehat{l}_w \rfloor$ otherwise.

6.2 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. E.g., consider 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

[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 a newly arrived item
Uniform sampling	$\widehat{l}_u$ = # tosses of a p-biased coin to get the first head
Weighted sampling	$\widehat{l}_w = \widehat{l}_u + \log_{(1-p)} w$
Exponentially decayed sampling	$\widehat{l}_e = \widehat{l}_u + L$
Weighted + Decayed	$\widehat{l}_{we} = \widehat{l}_u + \log_{(1-p)} w + L$

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.

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 uniform 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 6.2. *Suppose an item i is assigned a level $\widehat{l}_e = \widehat{l}_u + L_i$, where 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, the probability that item i is included in the sample exponentially decreases with $(L - L_i)$, where L is the current minimum active level.*

PROOF. Suppose the current minimum active level is L. Then, $\Pr\{i \in S\} = \Pr\{\widehat{l}_e \geq L\} = \Pr\{\widehat{l}_u \geq L - L_i\} = (1-p)^{(L-L_i-1)}$, 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.

6.3 Optimizations with More Memory

We briefly outline two optimizations that can be used when more memory is available. The first optimization uses more buckets to reduce the cost of maintaining the sample. As mentioned in Section 4.2.2, more buckets reduce sampling overheads at the cost of a bigger memory footprint. The second optimization maintains the skip pointers (of the skip list described in Section 5.2) and timestamp ranges of blocks in separate flash pages, instead of storing them at the end of each block. This reduces subsampling cost by retrieving multiple successive skip pointers with a single page read. This requires an additional in-memory page buffer to temporarily hold skip pointers before they are written to flash.

7. EVALUATION

In this section we experimentally evaluate our B-File-based sampling algorithm and a few existing algorithms.

Flash Devices. Unless otherwise stated, we use a s_{max} = 1.2GB flash device to maintain a s_{min} = 1GB sample from a data stream consisting of 1.5 billion 32-byte records. We use two flash devices for our experiments: (1) FLASHCHIP: a Toshiba flash chip, and (2) FLASHCARD: a Lexar 2GB CF card. Each flash page is 2KB and each block contains 64 pages. To measure the performance numbers for the FLASHCARD experiments, we connect the flash card, through a CF Extend 180 Extender Card [22], to the PCMCIA slot of an Intel P4 1.7 GHz laptop. We then connect a low ohmage (1 Ohm) current sense resistor in series with the extender card and measure the current with an oscilloscope. For the FLASHCHIP experiments, we have written a "driver" that emulates a Toshiba TC58DVG02A1FT00 NAND flash chip, whose performance has been accurately measured and profiled in a recent work [15]. Table 1 shows the energy consumption and latency of both these flash media. We also studied a few other flash cards from Kingston and SanDisk and SSD drives from Samsung and SanDisk, and the conclusions were identical; hence we omit results for those cards 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 7.2 and 7.3.

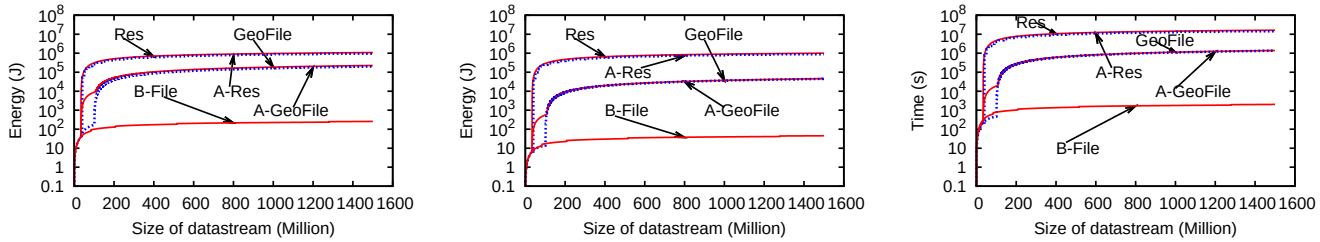
Algorithms Compared. We evaluate the following five algorithms: (1) **Reservoir (**RES**)**: the original reservoir sampling algorithm [23]. (2) **Adapted Reservoir (**A-RES**)**: the adapted reservoir algorithm described in Section 2.3. (3) **Geometric File (**GEOFILE**)**: the original geometric file based sampling algorithm [11]. 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 2.3. Lastly, (5) **B-File (**B-FILE**)**: the main algorithm described in this paper. Based on our analysis in Section 4.2.2, 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 even a larger buffer.

7.1 Cost of Maintaining Samples

Figure 6 shows the energy consumed by different algorithms to maintain a random sample from a data stream of varying length with FLASHCHIP and FLASHCARD and the time elapsed for the same with FLASHCARD. Note that the relative performance of different algorithms with FLASHCHIP and FLASHCARD is the same; moreover, the time consumed by different algorithms with FLASHCARD is proportional to the energy consumed. Therefore, in the rest of the section, we restrict our discussion to energy consumption for FLASHCARD; our conclusions naturally hold for energy for FLASHCHIP and for time for both FLASHCHIP and FLASHCARD.

Figure 6(b) makes several important points. First, the 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 com-



(a) Energy with flash chip (b) Energy with CF card (c) Time with CF card

Figure 6: Energy and time consumed by a flash chip and a flash card under varying stream sizes

pared 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.

To better understand the relative performance of different algorithms, we show in Table 4 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** result in increasing the number of sequential reads and writes.

A-RES improves upon RES by reducing **C1** and **C2**, at a cost of a small **C3** overhead, as shown in Table 4 by A-RES's fewer random and slightly higher sequential I/Os than RES. GEOFILE improves upon RES or A-RES by nearly eliminating **C1**. However, it still incurs a high **C2** overhead, because segments are not aligned to block boundaries. We found that $> 90\%$ of the reads and writes of GEOFILE are due to **C2**. 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 $(s_{max} - s_{min})$ = 0.2GB of extra space, it is not possible to allocate entire blocks to all segments in A-GEOFILE. Therefore, A-GEOFILE can not 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} = 10GB. Performance of GEOFILE and A-GEOFILE can also 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., $10\times$ the sample size) flash and and a large memory (e.g., $> 1\%$ of the sample size). In contrast, B-FILE naturally performs very close to OPT with very little extra space and memory. As shown in Table 4, B-FILE's I/O counts are quite close to optimal (note that semi-random and sequential writes have similar costs), thanks to its following all the desirable design principles; the additional reads and writes are due to **C3** overheads incurred while maintaining the tail bucket.

7.2 Biased Sampling

Table 4: 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	3.418	0.048
OPT	0	0	2.7	0	0	0.032

Figure 7 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. As shown, the cost of maintaining a weighted sample is slightly higher than maintaining an unbiased sample; moreover, the cost exponentially decreases with stream size because fewer items are included into the sample as 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 7. 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.

7.3 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 within the window $[0, 1.5 \times 10^9 - length]$. Figure 8 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 6.3). 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 an order of magnitude greater energy savings than sequential scan, and using the skip lists in separate pages provides another order of magnitude greater energy savings. The benefit comes from a 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 com-

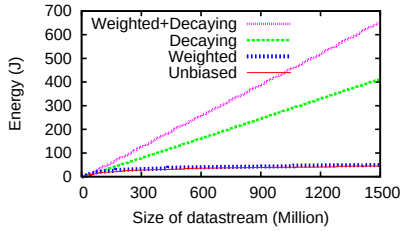


Figure 7: Biased sampling

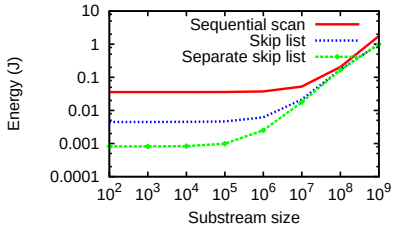


Figure 8: Energy consumed to subsample

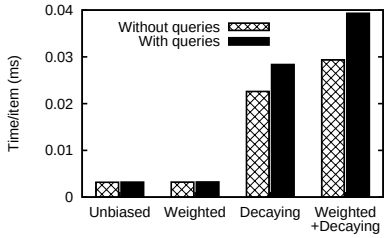


Figure 9: Sampling with queries

pared to the cost of reading subsequent records in the subsample.

7.4 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 our most expensive sampling scheme (Weighted + Decaying) can handle, and we run one sampling thread 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). Our results show that the impact of concurrent sampling on query latency is very small ($< 5\%$; details in Appendix B). In contrast, as shown in Figure 9, the impact is more significant on sampling latency (upto 35%). This is because with concurrent reads and writes on flash, write performance suffers more than read performance [1].

8. 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 [25]. 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.

Acknowledgements. The authors thank the anonymous reviewers for their detailed comments on how to improve the paper.

9. 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*, 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] S. Chaudhuri, G. Das, M. Datar, R. Motwani, and V. R. Narasayya. Overcoming limitations of sampling for aggregation queries. In *IEEE ICDE*, 2001.

[5] Y. Diao, D. Ganesan, G. Mathur, and P. Shenoy. Rethinking data management for storage-centric sensor networks. In *CIDR*, 2007.

[6] F. Douglis, R. Cáceres, F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber. Storage alternatives for mobile computers. In *USENIX OSDI*, 1994.

[7] P. B. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *ACM SPAA*, 2001.

[8] M. Hachman. New Samsung notebook replaces hard drive with flash. http://www.extremetech.com/article2/0,1558,1966644,00.asp, May 2006.

[9] Intel-Corporation. Understanding the Flash Translation Layer (FTL) specification. www.embeddedfreebsd.org/Documents/Intel-FTL.pdf, 1998.

[10] C. Jermaine, A. Datta, and E. Omiecinski. A novel index supporting high volume data warehouse insertion. In *VLDB*, 1999.

[11] C. Jermaine, A. Pol, and S. Arumugam. Online maintenance of very large random samples. In *ACM SIGMOD*, 2004.

[12] H. Kim and S. Ahn. BPLRU: a buffer management scheme for improving random writes in flash storage. In *Usenix FAST*, 2008.

[13] 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 EMSOFT*, 2007.

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

[15] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Capsule: an energy-optimized object storage system for memory-constrained sensor devices. In *ACM SenSys*, 2006.

[16] P. Miller. SimpleTech announces 512GB and 256GB 3.5-inch SSD drives. http://www.engadget.com/2007/04/18/, April 2007.

[17] S. Nath and A. Kansal. FlashDB: dynamic self-tuning database for NAND flash. In *ACM/IEEE IPSN*, 2007.

[18] F. Olken, D. Rotem, and P. Xu. Random sampling from hash files. *SIGMOD Rec.*, 19(2), 1990.

[19] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4), 1996.

[20] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6), 1990.

[21] 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.

[22] SyCard. CF extend 180 CompactFlash Flexible Extender Card. http://www.sycard.com/cfext180.html, 2008.

[23] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1), 1985.

[24] J. S. Vitter. An efficient algorithm for sequential random sampling. *ACM Trans. Math. Softw.*, 13(1), 1987.

[25] J. S. Vitter. External memory algorithms and data structures. *ACM Comput. Surveys*, 33(2), 2001.

[26] C.-H. Wu, L.-P. Chang, and T.-W. Kuo. An efficient R-tree implementation over flash-memory storage systems. In *ACM GIS*, 2003.

[27] Yahoo!-Finance. Zeus-IOPS solid state drives surge to 512GB. http://biz.yahoo.com/pz/070418/117663.html, April 2007.

[28] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar. Microhash: an efficient index structure for fash-based sensor devices. In *USENIX FAST*, 2005.

APPENDIX

A. ADDITIONAL B-FILE ALGORITHMS AND OPTIMIZATIONS

Randomized batch sampling from a B-File. The batch sampling algorithm in Section 5.1 is not truely randomized in the sense that there might be a significant overlap between two subsamples (since if both subsamples select a bucket, all items within the bucket will be in both subsamples). Although it might not be a problem for most applications, we here outline another batch sampling algorithm which is truely randomized, but less efficient than the algorithm in Section 5.1.

The algorithm is analogous to Olken and Rotem's procedure of batch sampling from a hashed file [18]. 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.

We would like to mention that geometric file can support a more efficient implementation of the above algorithm. Since the items in each segment are stored in a random order, the target number of samples from each segment can be drawn efficiently by sequential reads. In contrast, each B-FILE bucket keeps items stored in the order of their arrival times. Although requiring reading more flash pages during sampling within a bucket, the B-FILE approach has two advantages: (1) it avoids having to randomize data items before writing to flash, and (2) it enables efficiently producing a sample for any specified time window, as described in Section 6.2.

Algorithm 3 $GetNext()$

1: **while** $true$ **do**
2: $r = RAND(1, N+1)$
3: $j = RAND(1, b^*)$ {b^* is the size of the largest bucket}
4: **if** $j \leq |B_r|$ **then**
5: Return the j'th item in B_r

Iterative sampling from a B-File. Algorithm 3 shows an iterative algorithm that uses an acceptance/rejection test, like [18], 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. The number of pointers required to follow in order to locate the selected item can be reduced by using skip lists (Section 5.2), and the number of page reads required to extract the pointers can be reduced by maintaining the skip pointers in separate pages (Section 6.3).

Algorithm 4 $Search$(Bucket m_bucket, Time t_1, Time t_2)

1: $x \leftarrow m\_bucket.header$
2: **for** $i = m\_bucket.level$ **downto** 1 **do**
3: **while** the first item in block $x.forward[i]$ has timestamp $> t_2$ **do**
4: $x \leftarrow x.forward[i]$
5: $x \leftarrow x.forward[1]$
6: Binary search block x for the page p containing the item I_0 with the largest timestamp $\leq t_2$
7: 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

Searching within a skip list bucket. Algorithm 4 shows the pseudocode of the algorithm, described in Section 5.2, to locate the first record within a bucket satisfying a time range query.

Table 5: Energy consumed by different types of I/Os on two flash devices

Operation	*Energy (μJ)*
Device: Toshiba NAND TC58DVG02A1FT00 flash chip (128MB)	
Basic operations	
Page read [15]	57.83
Page write [15]	73.79
Block erase [15]	65.54
Append vs. update of a 32-byte record	
Append	1.17
In-place update	8854.21
Batch deletion of 32-byte records	
Delete 1 record	8357.6/record
Delete 16 consecutive records (1 page)	1.96/record
Delete 4096 consecutive records (1 block)	0.02/record
Device: a Lexar CF card (2 GB)	
Append vs. update of a 32-byte record	
Append	0.21
Update	7880.41

Proof of Lemma 4.2. We now present a proof of Lemma 4.2 from Section 4.1. 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 $\alpha = s_{min}/s_{max}$. □

Optimizing s_{min} and s_{max} of B-File. The size of a B-FILE fluctuates 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.

If the approximate size of the stream is known a priori, it is possible to determine the optimal s_{max} given an s_{min} (or vice versa). We here briefly outline how this can be done. Suppose the sample is being collected over a stream of size $|S'|$ and the B-FILE is configured to maintain N individual buckets. Then, it is possible (applying Lemma 4.3) 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 4.2.2.

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 simple binary search.

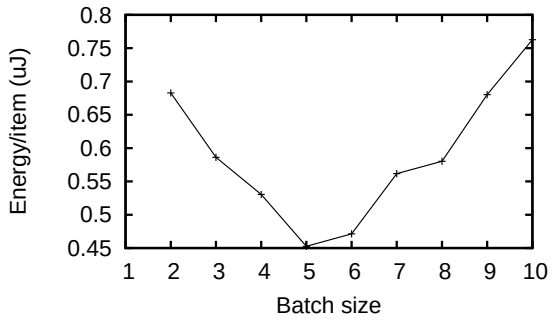


Figure 10: Effect of batch unroll of B_T

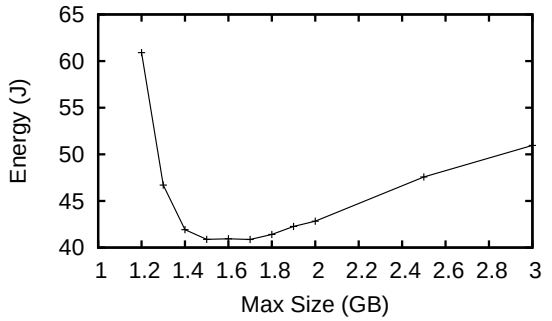


Figure 11: Effect of s_{max}

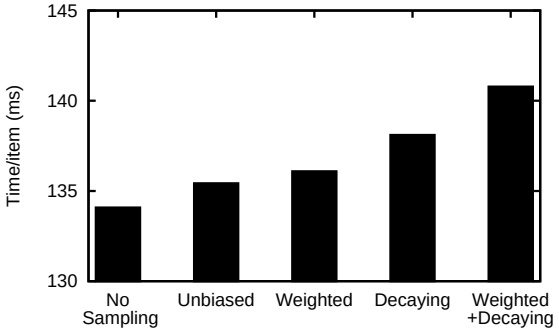


Figure 12: Cost of queries concurrent to sampling

B. ADDITIONAL EXPERIMENTAL RESULTS

Additional Measurements of Flash Devices. Table 5 shows the energy costs of the different flash I/Os discussed in Section 2. It presents the basic I/O costs, demonstrates the advantage of appending over overwriting data, and demonstrates the advantage of batching record deletions. Note that consumed energy is roughly proportional to latency, and hence any advantage in energy naturally translates to an advantage in latency.

Experimental Validation of Optimal B-File Parameters. Figure 10 shows the effect of doing log unroll in batch (from this point on we use energy measure as the cost and FLASHCARD as the flash media). As discussed in Section 4.2.2, 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 4.2.2 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 Appendix A, 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 Appendix A also gives the value 1.5GB.

Cost of queries with concurrent sampling. Figure 12 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 ($< 5\%$).