A Sequential Indexing Scheme for Flash-Based Embedded Systems

Shaoyi Yin [*,**,***]
*INRIA Rocquencourt
78153 Le Chesnay - France
Fname.Lname@inria.fr

Philippe Pucheral [*,**]
**PRiSM Lab, Univ. of Versailles
78035 Versailles - France
Fname.Lname@prism.uvsq.fr

Xiaofeng Meng [***]
***Renmin Univ. of China
100872 Beijing – China
{yinshaoy, xfmeng}@ruc.edu.cn

ABSTRACT

NAND Flash has become the most popular stable storage medium for embedded systems. As on-board storage capacity increases, the need for efficient indexing techniques arises. Such techniques are very challenging to design due to a combination of NAND Flash constraints (for example the block-erase-before-page-rewrite constraint and limited number of erase cycles) and embedded system constraints (for example tiny RAM and resource consumption predictability). Previous work adapted traditional indexing methods to cope with Flash constraints by deferring index updates using a log and batching them to decrease the number of rewrite operations in Flash memory. However, these methods were not designed with embedded system constraints in mind and do not address them. In this paper, we propose a new alternative for indexing Flash-resident data that specifically addresses the embedded context. This approach, called PBFilter, organizes the index structure in a purely sequential way. Key lookups are sped up thanks to two principles called Summarization and Partitioning. We instantiate these principles with data structures and algorithms based on Bloom Filters and show the effectiveness of this approach through a comprehensive performance study.

1. INTRODUCTION

Smart cards were equipped with kilobytes of EEPROM stable storage in the 90's and megabytes of NAND Flash in the 00's; mass-storage cards are coming soon that will link a microcontroller to gigabytes of NAND Flash memory [9]. All categories of smart objects (e.g., sensors, smart phones, cameras and mp4 players) benefit from the same storage capacity improvement thanks to high density NAND Flash. Smart objects are more versatile than ever and are now effective to manage medical, scholastic and other administrative folders, agendas, address books, photo galleries, transportation and purchase histories, etc. As storage capacity increases, the need for efficient indexing techniques arises. This motivates manufacturers of Flash modules and smart objects to integrate file management and even database techniques into their firmware.

Designing efficient indexing techniques for smart objects is very challenging, however, due to conflicting hardware constraints and design objectives.

On the one hand, although it is excellent in terms of shock resistance, density and read performance, NAND Flash exhibits specific hardware constraints. Read and write operations are done at a page granularity, as with traditional disks, but writes are more time and energy consuming than reads. In addition, a page cannot be rewritten without erasing the complete block containing it, which is a costly operation. Finally, a block wears out after about 10^5 repeated write/erase cycles. As a result, updates are usually performed "out-of-place" entailing address translation and garbage collection overheads. The more RAM is devoted to buffering and caching and the lazier garbage collection is, the better the performance.

On the other hand, smart object manufacturers are facing new constraints in terms of energy consumption (to increase device autonomy/lifetime), microcontroller size (to increase tamper-resistance) and storage capacity (to save production costs on billion-scale markets)[2]. In this context, performance competes with energy, RAM and Flash memory consumption. Co-design rules are therefore essential to help manufacturers calibrate the hardware resources of a platform and select the appropriate data-management techniques to match the requirements of on-board data-centric applications.

State of the art Flash-based storage and indexing methods were not designed with embedded constraints in mind and poorly adapt to this context. Database storage models dedicated to NAND Flash have been proposed in [11, 13] without specifically addressing the management of hot spot data in terms of updates, like indexes. Other work addressed this issue by adapting B+Tree-like structures to NAND Flash [4, 16, 18]. While different in their implementation, these methods rely on a similar approach: delaying index updates using a log dedicated to the index, and batching them with a given frequency so as to group updates related to the same index node. We refer to these methods as *batch methods*. The benefit of batch methods is that they decrease write cost, which is considered the main problem with using Flash in the database context. However, all these methods maintain additional data structures in RAM to limit the negative impact of delayed updates on lookup cost. All these methods also perform "out-of-place" updates, reducing Flash memory usage and generating address translation and garbage collection overheads. Such indirect costs have proven high and unpredictable [16].

Rather than adapting traditional index structures to Flash memory, we believe that indexing methods must be completely rethought if we are to meet the requirements of the embedded context, namely:

– *Low_RAM*: accommodate as little RAM as possible
– *Low_Energy*: consume as little energy as possible
– *Low_Storage*: optimize the Flash memory usage
– *Adaptability*: make resource consumption adaptable to the

EDBT'09, March 24-26, 2009, Saint Petersburg, Russia.

performance requirements of on-board applications

– *Predictability*: make performance and resource consumption fully predictable

Low_RAM emphasizes the specific role played by RAM in the embedded context. Due to its poor density, and as it competes with other hardware resources on the same silicon die, RAM is usually calibrated to its bare minimum [2]. Hence, the less RAM an indexing method consumes, the wider the range of devices that can be targeted. Low_Energy is also critical but concerns only autonomous devices. The objective of Low_Storage is to minimize not only the amount of Flash memory occupied by the index structure, but also, and above all, of Flash memory wasted by the obsolete data produced by index updates and leading to overprovisioning Flash memory. Adaptability conveys the idea that optimal performance is not the ultimate goal; rather optimality is reached when no resource is unduly consumed to get better performance than that strictly required by on-board applications. In other words, Adaptability means that Low_RAM, Low_Energy and Low_Storage must be considered in light of the applications' performance expectations. Finally, Predictability is a prerequisite to co-design.

In this paper, we propose a Flash-based indexing method, called PBFilter, specifically designed to answer these requirements. PBFilter organizes the index structure in a purely sequential way to minimize the need for buffering and caching and to avoid the unpredictable side effects incurred by "out-of-place" updates. But how to look up a given key in a sequential list with acceptable performance? We answer this question using two principles. Summarization consists of building an index summary used at lookup time to quickly determine the region of interest in the index. This introduces an interesting source of tuning between the compression ratio of the summary and its accuracy. Partitioning consists of vertically splitting the index list and/or its summary in such a way that only a subset of partitions need to be scanned at lookup time. This introduces a second trade-off between lookup performance and RAM consumption. The key idea behind Summarization and Partitioning is speeding up lookups without hurting sequential writes in Flash memory.

PBFilter gracefully accommodates files up to a few million tuples, a reasonable limit for embedded applications. PBFilter is optimized to support append-oriented files but deletion and updates can be supported without compromising the five requirements above.

The paper is organized as follows. Section 2 reviews the main characteristics of NAND Flash, studies the related work and introduces the metrics of interest for this study. Section 3 details the PBFilter indexing scheme. Section 4 presents an instantiation of the PBFilter scheme with partitioned Bloom Filter summaries. Section 5 presents a comprehensive performance study and Section 6 is the conclusion.

2. PROBLEM STATEMENT

2.1 NAND Flash Characteristics

Embedded Flash devices come today in various form factors such as compact flash cards, secure digital cards, smart cards and USB tokens. They share several common characteristics. A typical NAND Flash array is divided into blocks, in turn divided into pages (32-64 pages per block), and divided again into sectors (usually 4 sectors per page). Read and write operations usually happen at page granularity, but can also apply at sector granularity if required. A page is typically 512-2,048 bytes. A page can only be rewritten after erasing the entire block containing it (usually called the block-erase-before-rewrite constraint). Page write cost is higher than read, both in terms of execution time and energy consumption, and the block erase requirement makes writes even more expensive. A block wears out after about 10^5 repeated write/erase cycles, requiring write load to be evenly spread out across the memory.

These hardware constraints make update management complex, although this complexity is slightly mitigated by the decomposition of a page into sectors. Sectors can be written independently (albeit sequentially) in the same page, allowing one write per sector before the block must be erased. To avoid erasing blocks too frequently, "out-of-place" updates are usually performed by using a Flash Translation Layer (FTL) which combines: (1) a translation mechanism relocating the pages and making their address invariant through indirection tables and (2) a garbage collection mechanism that erases blocks, either lazily (waiting for all the pages of the block to become obsolete) or eagerly (moving the active pages still present in the block before erasing it).

As extensively studied in [16], the execution time and energy consumption of read and write operations vary greatly among the Flash devices. The high discrepancies between the platforms are partly due to the raw chip characteristics and partly to the firmware managing the FTL which is usually proprietary and constitutes the primary source of performance unpredictability [16].

2.2 Related Work

Some work [11, 13] has adopted the idea of log-structured file systems (LFS) [17] to design or improve database storage models dedicated to NAND Flash, without proposing new index methods. For example, the primary objective of IPL [13] is to hide Flash peculiarities from the upper layers of the DBMS. Updates in Flash are delayed using a log stored in each physical block and an accurate version of each page is rebuilt at load time. Updates are physically applied to a page when the corresponding log region becomes full. While elegant, this general method is not well suited to managing hot spot data in terms of updates, like indexes, because of frequent log overflows.

Other work has specifically considered the indexing problem in NAND Flash. Hash-based and tree-based index families can be distinguished. So far, little attention has been paid to hash-based methods in Flash. This is probably because hashing only performs well when a large number of buckets can be used and when the RAM can accommodate one buffer per bucket, which is a rare situation in the targeted context. One exception is Microhash designed to speed up lookups in sensor devices [22]. However, this method is not general and only applies to sensed data varying within a small range (e.g., temperature).

Within the tree-based family, one work has also considered indexing sensed data [14]. This work proposes a tiny index called TINX based on a specific unbalanced binary tree. The performance demonstrated by the authors (e.g., 2,500 page reads to retrieve one record from 0.6 million records) disqualifies this method for large files. To the best of our knowledge, all other papers suggest adaptations of the well-known B+Tree structure. Regular B+Tree techniques built on top of FTL have been shown to be poorly adapted to the characteristics of Flash memory [18]. Indeed, each time a new record is inserted into a file, its key must be added to a B+Tree leaf node, causing an out-of-place update of the corresponding Flash page. To avoid such updates, BFTL [18] constructs an “index unit” for each inserted primary key and organizes the index units as a kind of log. A large buffer is allocated in RAM to group the various insertions related to the same B+Tree node in the same log page. To maintain the logical view of the B+Tree, a node translation table built in RAM keeps, for each B+Tree node, the list of log pages which contain index units belonging to this node. In order to limit the size of these lists and therefore RAM consumption as well as lookup cost, each list is compacted when a certain threshold (e.g., 5 log pages in the list) is reached. At this time, logged updates are batched to refresh the physical image of the corresponding B+Tree node.

FlashDB [16] combines the best of Regular B+Tree and BFTL using a self-tuning principle linked to the query workload. JFFS3 proposes a slightly different way of optimizing B+Tree usage [4]. Key insertions are logged in a journal and are applied in the B+Tree in a batch mode. A journal index is maintained in RAM (recovered at boot time) so that a key lookup applies first to the journal index and then to the B+Tree.

In short, all B+Tree-based methods rely on the same principle: (1) delay index updates using a log and batch them with the purpose of grouping updates related to the same index node; (2) build a RAM index at boot time to speed up lookup of a key in the log; (3) commit log updates with a given commit frequency (CF) in order to limit log size. The differences between batch methods mainly include the way index nodes and log are materialized, which affect the way CF is managed.

In their attempt to decrease the number of writes, batch methods are in line with the Low_Energy requirement introduced in Section 1. By allowing trading reads, RAM and Flash memory usage for writes using CF, they also provide an answer to Adaptability. However, all batch methods fail in satisfying Low_RAM. Indeed, the higher the CF, the greater the RAM consumption. However, the primary objective of batch methods is to decrease the number of writes in Flash memory, leading to a higher CF. Section 5 will demonstrate that good write performance for batch methods requires RAM consumption incompatible with most embedded environments (in any case, not the objective they claim). Regarding Predictability, even if the number of writes is reduced, writes still generate out-of-place updates in Flash memory. This results in an indirect and unpredictable garbage collection cost linked to the strategy implemented in the underlying FTL [16]. Flash memory usage is also difficult to predict because it depends on the distribution of obsolete data in the pages occupied by the index.

2.3 Metrics of Interest

In light of the preceding discussion, more complete and accurate metrics appear necessary to help in assessing the adequacy of an indexing method for the embedded context. To this end, we propose the following metrics to capture the five requirements introduced in Section 1:

- *RAM consumption*: as already stated, RAM consumption is of utmost importance in the embedded context, since several devices (e.g., smart cards, sensors and smart tokens) are equipped with RAM measured in kilobytes [2]. This metric, denoted hereinafter *RAM*, comprises the buffers to read from and write to the Flash memory as well as the main memory data structures required by the indexing method.
- *Read/write cost*: this metric distinguishes between read cost *R* of executing a lookup and read cost *IR* and write cost *W* for inserting keys into the index. Depending on the objective, the metric can be execution time (wrt Adaptability) or energy consumption (wrt Low_Energy). To address both concerns, R, IR and W will be expressed in terms of number of operations. Note that this metric does not directly capture the Adaptability requirement, but rather tells whether the performance expected by on-board applications can be achieved.
- *Flash memory usage*: the objective is to capture the Flash memory usage, both in terms of space occupancy and effort to reclaim obsolete data. We distinguish between two values: *VP* is the total number of valid pages occupied by the index (i.e., pages containing at least one valid item); *OP* is the total number of pages containing only obsolete data and which can be reclaimed without copying data. Comparing these two values with the raw size of the index (total size of the valid items only) gives an indication of the quality of the Flash memory usage and the effort to reclaim stale space, independent of any FTL implementation.
- *Predictability*: as claimed in the introduction, performance and resource consumption predictability is a prerequisite for co-design. Predictability is mandatory in calibrating the RAM and Flash memory resources of a new hardware platform to the performance requirements of the targeted on-board applications. Another objective is to predict the limit (i.e., in terms of file size or response time) of an on-board application on existing hardware platforms. Finally, predictability is also required to build accurate query optimizers. To avoid making this metric fuzzy by reducing it to a single number, we express it qualitatively using two dimensions: (1) whether the indexing method is dependent on an underlying FTL or can bypass it, (2) whether the values measured for RAM, read/write cost and Flash memory usage can be accurately bounded independent of their absolute value and of the uncertainty introduced by the FTL, if any.

This paper aims to define a Flash-based indexing method that behaves satisfactorily in all of these metrics at once.

3. PBFILTER INDEXING SCHEME

As an alternative to the batch indexing methods, PBFilter performs index updates eagerly and makes this acceptable by organizing the complete database as a set of sequential data structures, as presented in Figure 1. The primary objective is to transform database updates into append operations so that writes are always produced sequentially, an optimal scenario for NAND Flash and buffering in RAM.

The database updating process is as follows. When a new record is inserted, it is added at the end of the record area (RA). Then, a new index entry composed by a couple <key, pt> is added at the end of the key area (KA), where key is the primary key of the inserted record and pt is the record physical address[1]. If a record

[1] Like all state of the art methods mentioned in Section 2, we concentrate the study on primary keys. The management of secondary keys is discussed in [21].

is deleted, its identifier (or its address) is inserted at the end of the delete area (DA) but no update is performed in RA nor KA. A record modification is implemented by a deletion (of the old record) followed by an insertion (of the new record value). To search for a record by its key, the lookup operation first scans KA, retrieves the required index entry if it exists, check that pt $\notin$DA and gets the record in RA. Assuming a buffering policy allocating one buffer in RAM per sequential data structure, this updating process never rewrites pages in Flash memory.

The benefits and drawbacks provided by this simple database organization are obvious with respect to the metrics introduced in Section 2. *RAM:* a single RAM buffer of one page is required per sequential structure (RA, KA and DA). The buffer size can even be reduced to a Flash sector in highly constrained environments. *Read/write cost:* a lower bound is reached in terms of reads/writes at insertion time (IR and W) since: (1) the minimum of information is actually written in Flash memory (the records to be inserted and their related index entries and no more), (2) new entries are inserted at the index tail without requiring any extra read to traverse the index. On the other hand, the lookup cost is dramatically high since R = (|KA|/2 + |DA| + 1) on the average, where | | denotes the page cardinality of a structure. *Flash memory usage:* besides DA, a lower bound is reached in terms of Flash usage, again because the information written is minimal and never updated. Hence, the number VP of valid pages containing the index equals the raw size of this index and the number OP of obsolete pages is null. Hence, the garbage collection cost is saved. *Predictability:* since data never moves and is never reclaimed, PBFilter can bypass the FTL address translation layer and garbage collection mechanism. RAM and Flash memory consumption is accurately bounded as discussed above. However, performance predictability is not totally achieved since the uncertainty on R is up to (|KA| - 1).

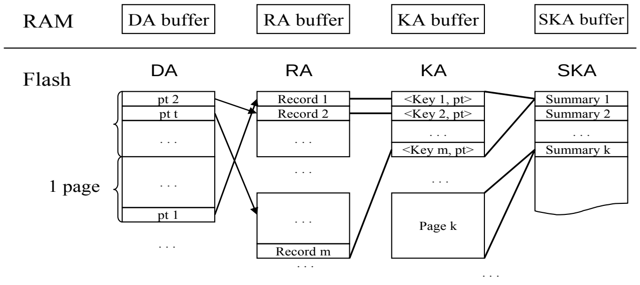


Figure 1. Database organization

The objective is now to decrease the lookup cost R to an acceptable value with a minimal degradation of the benefits listed above. Summarization and Partitioning are two principles introduced to reach this goal.

Summarization refers to any method which can condense sequentially the information present in KA. Let us consider an algorithm that condenses each KA page into a summary record. Summary records can be sequentially inserted into a new structure called SKA through a new RAM buffer of one page (or sector) size. Then, lookups do a first sequential scan of SKA and a KA page is accessed for every match in SKA in order to retrieve the requested key, if it exists. Summarization introduces an interesting trade-off between the compression factor *c* (c= |KA|/ |SKA|) and the fuzziness factor *f* (i.e., probability of false positives) of the summary, the former decreasing the I/O required to traverse SKA and the latter decreasing the I/O required to access KA. The net effect of summarization is reducing R to (|KA|/2c + f||KA||/2) on the average, where || || denotes the element cardinality of a structure. The positive impact on R can be very high for favorable values of *c* and *f*. The negative impact on the RAM consumption is limited to a single new buffer in RAM. The negative impact on the write cost and Flash memory usage is linear with |SKA| and then depends on *c*. Different algorithms can be considered as candidate "condensers", with the objective to reach the higher *c* with the lower *f*, if only they respect the following property: *summaries must allow membership tests with no false negatives*.

The idea behind *Partitioning* is to vertically split a sequential structure into *p* partitions so that only a subset of partitions has to be scanned at lookup time. Partitioning can apply to KA, meaning that the encoding of keys is organized in such a way that lookups do not need to consider the complete key value to evaluate a predicate. Partitioning can also apply to SKA if the encoding of summaries is such that the membership test can be done without considering the complete summary value. The larger *p*, the higher the partitioning benefit and the better the impact on the read cost and on Predictability. On the other hand, the larger *p*, the higher the RAM consumption (*p* buffers) or the higher the number of writes into the partitions (less than *p* buffers) with the bad consequence of reintroducing page moves and garbage collection. Again, different partitioning strategies can be considered with the following requirement: *to increase the number of partitions with neither significant increase of RAM consumption nor need for garbage collection*.

4. PBFILTER INSTANTIATION

4.1 Bloom Filter Summaries

The Bloom Filter data structure has been designed for representing a set of elements in a compact way while allowing membership queries with a low rate of false positives and no false negative [5]. Hence, it presents all the characteristics required for a condenser.

A Bloom filter represents a set $A=\{a_1, a_2, \dots a_n\}$ of *n* elements by a vector *v* of *m* bits, initially all set to 0. The Bloom filter uses *k* independent hash functions, h_1, h_2, … h_k, each producing an integer in the range [1,m]. For each element $a_i \in A$, the bits at positions $h_1(a_i)$, $h_2(a_i)$, ..., $h_k(a_i)$ in *v* are set to 1. Given a query for element a_j, all bits at positions $h_1(a_j)$, $h_2(a_j)$, ..., $h_k(a_j)$ are checked. If any of them is 0, then a_j cannot be in *A*. Otherwise we conjecture that a_j is in *A* although there is a certain probability that we are wrong. The parameters *k* and *m* can be tuned to make the probability of false positives extremely low [8].

Table 1. False positive rate under various m/n and k

m/n	k=3	k=4	k=5	k=6	k=7	k=8
8	.0306	.024	.0217	.0216	.0229	
12	.0108	.0065	.0046	.0037	.0033	.0031
16	.005	.0024	.0014	.0009	.0007	.0006

This probability, called the *false positive rate* and denoted by *f* in the sequel, can be calculated easily assuming the k hash functions are random and independent. After all the elements of *A* are hashed into the Bloom filter, the probability that a specific bit is still 0 is $(1-1/m)^{kn} \approx e^{-kn/m}$. The probability of a false positive is then $\left(1-(1-1/m)^{kn}\right)^k \approx \left(1-e^{-kn/m}\right)^k = (1-p)^k$ for $p=e^{-kn/m}$. The salient feature of Bloom filters is that three performance metrics can be traded off against one another: computation time (linked to the number k), space occupancy (linked to the number m), and false positive rate *f*. Table 1 illustrates these trade-offs for some values of k and m. This table shows that a small increase of m may allow a dramatic benefit for f if the optimal value of k is selected. We consider that k is not a limiting factor in our context, since methods exist to obtain k hash values by calling only three times the hash function, while giving the same accuracy as by computing k independent hash functions [7].

Bloom filters can be used as a condenser algorithm in PBFilter as follows. For each KA page, a Bloom filter summary is built by applying *k* hash functions to each index key present in that page. This computation is performed when the KA page is full, just before the RAM buffer containing it is flushed to the Flash memory. The computed Bloom filter summary is stored in the RAM buffer allocated to SKA. In turn, the SKA buffer is flushed to the Flash memory when full. At lookup time, the searched key a_i is hashed with the *k* hash functions. Then, SKA is scanned to get the first Bloom filter summary having all bits at positions $h_1(a_i)$, $h_2(a_i)$, ..., $h_k(a_i)$ set to 1. The corresponding page of KA is directly accessed and the probability that it contains the expected index entry (a_i,pt) is (1-f). Otherwise, the scan continues in SKA. The last step is to check that pt $\notin$DA before accessing the record in RA.

4.2 Dynamic Partitioning

Despite the benefits of summarization, the lookup performance remains linked to the size of SKA (on the average, half of SKA needs to be scanned). The lookup performance can be improved by applying the partitioning principle suggested in section 3. Each Bloom filter is vertically split into *p* partitions (with $p \leq m$), so that the bits in the range [1 .. m/p] belong to the first partition, the bits in the range [((i-1)*m/p + 1) .. (i*m/p)] belong to the ith partition, etc. When the SKA buffer is full, it is flushed into *p* Flash pages, one per partition. By doing so, each partition is physically stored in a separate set of Flash pages. When doing a lookup for key a_i, instead of reading all pages of SKA, we need to get only the SKA pages corresponding to the partitions containing the bits at positions $h_1(a_i)$, $h_2(a_i)$, ..., $h_k(a_i)$. The benefit is a cost reduction of the lookup by a factor p/k. The larger *p*, the higher the partitioning benefit for lookups but also the greater the RAM consumption (p more buffers) or the greater the number of writes (because page fragments have to be flushed in the partitions in Flash memory instead of full pages) and then the need for garbage collection (because of multiple writes in the same page of Flash).

We propose below a partitioning mechanism which exhibits the nice property of supporting a dynamic increase of p with no impact on the RAM consumption and no need for a real garbage collection (as discussed at the end of the section, obsolete data is naturally grouped in the same blocks which can be erased as a whole at low cost). This dynamic partitioning mechanism comes at the price of introducing a few reads and extra writes at insertion time. The proposed mechanism relies on: (1) the usage of a fixed amount of Flash memory as a persistent buffer to organize a stepwise increase of p and (2) the fact that a Flash page is divided into *s* sectors (usually *s*=4) which can be written independently. The former point gives the opportunity to reclaim the Flash buffer at each step in its integrality (i.e., without garbage collection). The latter point allows *s* writes into the same Flash page before requiring copying the page elsewhere.

Figure 2 illustrates the proposed partitioning mechanism. The size of the SKA buffer in RAM is set to the size of a Flash page and the buffer is logically split into *s* sectors. The number of initial partitions, denoted next by L1 partitions, is set to *s* and one page of Flash is initially allocated to each L1 partition. The first time the SKA buffer in RAM becomes full (step 1), each sector s_i (with $1 \leq i \leq s$) of this buffer is flushed in the first sector of the page allocated to the ith L1 partition. The second flush of the SKA buffer will fill in the second sector of these same pages and so forth until the first page of each L1 partition becomes full (i.e., after *s* flushes of the SKA buffer). A second Flash page is then allocated to each L1 partition and the same process is repeated until each partition contains *s* pages (i.e., after s^2 flushes of the SKA buffer). Each L1 partition contains 1/s part of all Bloom filters (e.g., the ith L1 partition contains the bits in the range [((i-1)*m/s + 1) .. (i*m/s)]).

At this time (step 2), the *s* L1 partitions of *s* pages each are reorganized (read back and rewritten) to form s^2 L2 partitions of one page each. Then, each L2 partition contains 1/ s^2 part of all Bloom filters. As illustrated in Figure 2, each L2 partition is formed by projecting the bits of the L1 partition it stemmed from on the requested range, *s* times finer (e.g., the ith L2 partition contains the bits in the range [((i-1)*m/s^2 + 1) .. (i*m/s^2)]).

After another s^2 SKA buffer flushes (step 3), *s* new L1 partitions have been built again and are reorganized with the s^2 L2 partitions to form (s^2+s^2) L3 partitions of one page each and so forth. The limit is *p=m* after which there is no benefit to partition further since each bit of bloom filter is in a separate partition. After this limit, the size of partitions grows but the number of partitions remains constant (i.e., equal to m).

In the example presented in Figure 2, where s=4, the number of partitions grows in an approximately linear way (4, 16, 32…)[2]. Assuming for illustration purpose Flash pages of 2KB, bloom filters of size m=2048 bits in SKA and <key,pt> of size 8 bytes in KA, each page of L3 partitions gathers 1/32 part of 256 bloom filters summarizing themselves 65536 keys. Scanning one complete partition in SKA costs reading the corresponding page in L3 plus 1 to s pages in L1.

More precisely, the benefit of partitioning dynamically SKA is as follows. A lookup needs to consider only *k* Li partitions of one page each (assuming the limit p=m has not been reached and Li partitions are the last produced) plus min (*k ,s)* L1 partitions, the size of which vary from 1 to *s* pages. This leads to an average cost of (*k* + min (k, s) * *s*/2). This cost is both low and independent of the file size while p≤m.

The RAM consumption remains unchanged, the size of the SKA buffer being one page (note that extending it to *s* pages would save the first iteration). The impact on IR and W (Read and write cost at insertion time) is an extra cost of about $\sum_i 2^{\lceil \log_2 i \rceil} * s^2$ reads and writes (see the cost model for details). This extra cost may be considered important but is strongly mitigated by the fact that it applies to SKA where each page condenses B_p/d records, where B_p is the size of a Flash page in bits (B_p /d is likely to be

[2] In practice, it does not grow exactly linearly because the bloom filter cannot be equally divided into an arbitrary number of partitions. For the same reason, the bloom filter size is always a power of 2, so one bloom filter may summarize more than 1 (less than 2) KA pages. The impact of these implementation details have been taken into account in the cost model in Section 5, and the extra cost has proven low.

greater than 1000). Section 5 will show that this extra cost is actually low compared to existing indexing techniques. Section 5 will also show the low impact of partitioning on the Flash usage for the same reason, that is the high compression ratio obtained by Bloom filters making SKA small with respect to KA.

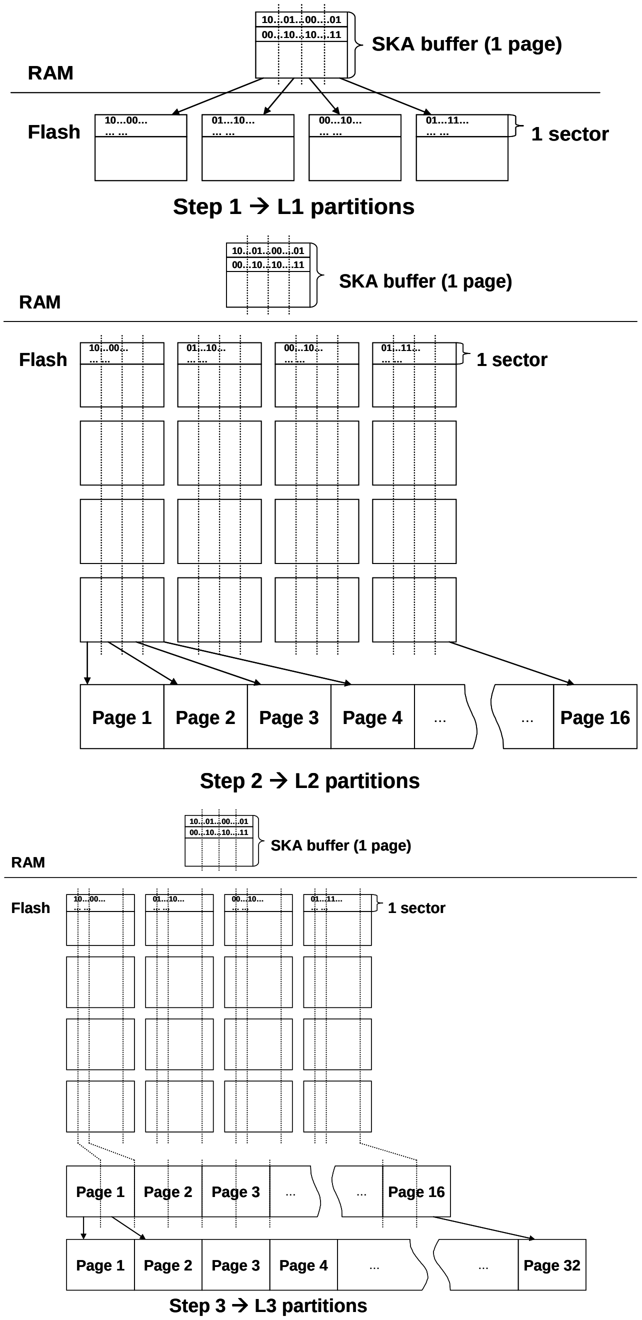


Figure 2. Dynamic partitioning

At the end of each step i, and after Li partitions have been built, the Flash buffer hosting L1 partitions and the pages occupied by L_{i-1} partitions can be reclaimed. Reclaiming a set of obsolete pages stored in the same block is far more efficient than collecting garbage crumbs spread over different pages in different blocks. The distinction between garbage reclamation and garbage collection is actually important. Garbage collection means that active pages present in a block elected for erasure must be moved first to another block. In addition, if at least one item is active in a page, the complete page remains active. In methods like BFTL, active index units can be spread over a large number of pages in an uncontrolled manner. This generates a worst situation where many pages remain active while they contain few active index units and these pages must be often moved by the garbage collector. PBFilter never generates such situations. The size of the Flash buffer and of the L_i partitions is a multiple of s^2 pages and these pages are always reclaimed together. Blocks are simply split in areas of s^2 pages and a block is erased when all its areas are obsolete.

4.3 Hash then Partition

As stated above, the benefit of partitioning is a cost reduction of the lookup by a factor p/k. The question is whether this factor can still be improved. When doing a lookup for key a_i in the current solution, the probability that positions $h_1(a_i)$, $h_2(a_i)$, ..., $h_k(a_i)$ fall into a number of partitions less than k is low, explaining the rough estimate of the cost reduction by the factor p/k. This situation could be improved by adding a hashing step before building the Bloom filters. Each Bloom filter is split into q buckets by a hash function h_0 independent of h_1, h_2, ..., h_k,. Each time a new key is inserted in KA, h_0 is applied first to determine the right bucket, then h_1, h_2, ..., h_k are computed to set the corresponding bits in the selected bucket. This process is similar as building q small Bloom filters for each KA page. The experiments we conducted led to the conclusion that q must remain low to avoid any negative impact on the false positive rate. Thus, we select $q=s$ (with s usually equals to 4). The benefit of this initial hashing is guaranteeing that the k bits of interest for a lookup always fall into the same L1 partition, leading to an average cost of (k + s/2) for scanning SKA.

4.4 An Illustration of Hashed PBFilter

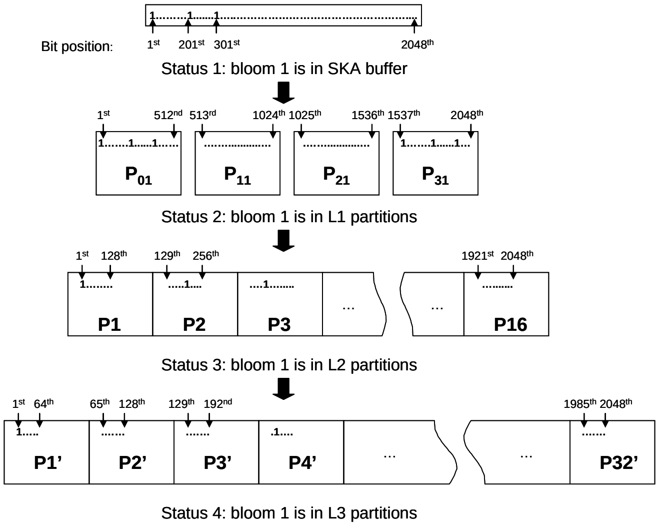
Now let us illustrate the key insertion and lookup processes of hashed PBFilter through an example (Figure 3). As pointed above, we set q=s=4 and m=2048, while supposing the size of <key, pt> is 8 bytes and the size of a page is 2048 bytes. To simplify the calculation, we use only 3 hash functions to build the bloom filters, denoted by h_1(key), h_2(key) and h_3(key). The hash function used in the pre-hashing step is denoted by h_0(key).

When the first key key1 is inserted, the hash bucket number is determined first by using h_0, and then h_1, h_2 and h_3 are computed. Suppose that: h_0(key1) = 0, h_1(key1) = 1, h_2(key1) = 201, and h_3(key1) = 301. Accordingly, the 1st, 201st and 301st bits in bucket 0 (the first 512 bits) of the first bloom filter bloom1 are set to 1 (Status 1 in Figure 3).

After inserting 2048 keys, the SKA buffer is full with 8 bloom filters (each bloom filter summarizes one KA page which contains 256 <key, pt> entries), so the bloom filters are partitioned and flushed into the L1 partitions: the first 512 bits (bucket 0) of each bloom filter are written into the first sector of page P_{01}, the second 512 bits (bucket 1) of each bloom filter are written into the first sector of page P_{11}, and so on (Status 2).

After inserting 32768 keys, the L1 partition pages are full, so the bloom filters are repartitioned into smaller pieces forming L2: the first 128 bits of all 128 bloom filters are written into the first L2 partition P1, the second 128 bits of all 128 bloom filters are written into the second L2 partition P2, and so forth (Status 3).

After inserting 65536 keys, the new L1 partitions are full again, so the bloom filters are repartitioned once more into ever smaller pieces forming L3: the first 64 bits of all 256 bloom filters (128 from the L2 partitions and 128 from the L1 partitions) are written into the first L3 partition P1', the second 64 bits of all 256 bloom filters are written into the second L3 partition P2', and so forth (Status 4).



Bit position:
1st 201st 301st 2048th
1.........1.......1
Status 1: bloom 1 is in SKA buffer
1st 512nd 513rd 1024th 1025th 1536th 1537th 2048th
1.......1......1......
1.......1......1...
P01
P11
P21
P31
Status 2: bloom 1 is in L1 partitions
1st 128th 129th 256th 1921st 2048th
1........
.....1....
....1........
P1
P2
P3
P16
Status 3: bloom 1 is in L2 partitions
1st 64th 65th 128th 129th 192nd 1985th 2048th
1.....
.1....
P1'
P2'
P3'
P4'
P32'
Status 4: bloom 1 is in L3 partitions

Figure 3. Storage status changing of a bloom filter

Now the bloom filters have been partitioned three times and have produced 32 L3 partitions each containing 64 bits of each bloom filter. Note that each of the L3 partitions still belongs to a single hash bucket set by h_0: the first 8 pages belong to bucket 0, the second 8 pages belong to bucket 1, and so on.

At this time, the process for looking up key1 is as follows. First, compute the hash functions to locate the required bit positions: h_0(key1) = 0, h_1(key1) = 1, h_2(key1) = 201, and h_3(key1) = 301, which means that, the 1st, 201st, and 301st bit positions of bucket 0 should be checked. In the L3 partitions, the three bit positions are stored in P1', P4' and P5' respectively, so only these pages have to be loaded into RAM. In this case, key1 will be found by only checking these pages. In other cases, if the searched key is not found in the L3 partitions, the current L1 partitions must be checked also: instead of scanning all the L1 partitions, only the pages in the corresponding bucket need to be checked (at most s pages), for example, if h_0(key) = 1, only P_{11}, P_{12}, P_{13} and P_{14} are scanned if they are not empty.

4.5 Deletes and Updates

PBFilter has been preliminary designed to tackle applications where insertions are more frequent and critical than deletes or updates. This characteristic is common in the embedded context. For instance, deletes and updates are proscribed in medical folders and many other administrative folders for legal reasons. Random deletes and updates are also meaningless in several applications dealing with historical personal data, audit data or sensed data. Note that cleaning history to save local space differs from deleting/updating randomly elements. While the latter impose to deal with a large DA area, the former can be easily supported. Indeed, cleaning history generates bulk and sequential deletes of old data. A simple low watermark mechanism can isolate the data related in RA, KA and SKA to be reclaimed together.

Let us now consider a large number of random deletes and updates enlarging DA and thereby decreasing the lookup performance. The solution to tackle this situation is to index DA itself using the same strategy, that is building bloom filters on the content of DA pages and partitioning them. The lookup cost being non linear with the file size, there is a great benefit to keep a single DA area for the complete database rather than one per file. This will bound the extra consumption of RAM to s more buffers for the whole architecture. The extra cost in Flash memory is again strongly limited by the high compression ratio of bloom filters. As section 5 will show, the lookup cost is kept low, though roughly multiplied by a factor 2 with high update/delete rate.

5. PERFORMANCE EVALUATION

The first objective of this section is to study how traditional B+Tree, batch methods and PBFilter perform in the embedded context. To allow a fair comparison between the approaches and isolate the FTL cost indirectly paid by batch methods and B+Tree, we introduce a precise analytical cost model. The results are more easily interpretable than real measurements performed on an opaque firmware. These results show that, while B+Tree and batch methods can slightly outperform PBFilter in some situations, PBFilter is the sole method to meet all requirements of an embedded context. Then, this section discusses how PBFilter can be tuned in a co-design perspective. Finally, preliminary performance measurements conducted on a specific hardware platform are given for illustrative purpose.

5.1 Analytical Performance Comparison

5.1.1 Indexing Methods under Test

As stated above, the objective is not to perform an exhaustive comparison of all Flash-based indexing methods, considering that only PBFilter has been specifically designed to cope with embedded constraints. The comparison will then concentrate on opposite approaches (traditional, batch, Summarization & Partitioning), rather than focusing on variations. Regular B+Tree running on top of FTL, denoted by BTree hereafter, is considered as a good representative of traditional disk-based indexing methods running on Flash memory with no adaptation. BFTL [18] is selected as a good, and probably best known, representative of batch methods. To better understand the impact of (not) bounding the log size in batch methods, we consider two variations of BFTL: BFTL1 with no compaction of the node translation table and BFTL2 with the periodic compaction of the node translation table suggested in [18]. The Bloom filter instantiation of PBFilter, denoted by PBF hereafter, is so far the unique representative of Summarization & Partitioning methods.

The performance metrics used to compare these methods are those introduced in Section 2.3, namely: RAM (RAM consumption in KB), R (average number of page reads to lookup a key), IR (total number of page reads to insert N records), W (total number of page writes to insert N records), VP (total number of valid Flash pages) and OP (total number of obsolete Flash pages).

5.1.2 Parameters and Formulas

The parameters and constants used in the analytical model are listed in Table 2 and Table 3, respectively.

Table 4 contains basic formulas used in the cost model and the cost model itself is presented in Table 5. To make the formulas as precise as possible, we use Yao's Formula [20] when necessary.

- ***Yao's Formula**: Given n records grouped into m blocks (1<m≤n), each contains n/m records. If k records (k≤n-n/m) are randomly selected, the expected number of blocks hit is:*

$$Yao(n,m,k) = m \times \left[1 - \prod_{i=1}^{k} \frac{nd - i + 1}{n - i + 1}\right], where, d = 1 - 1/m$$

Table 2. Parameters for the analytical model

Param	Signification
N	Total number of inserted records
Sk	Size of the primary key (in bytes)
B	Number of buffer pages in RAM
C	Maximum size of a node translation table list in BFTL2
d	Value of m/n in Bloom filter (see Table 1 for examples)
k	Number of hash functions used by Bloom filter

Table 3. Constants for the analytical model

Constants	Signification
Sr=4 (bytes)	Size of a physical pointer
fb=0.69	Average fill factor of B+Tree [19]
β =2	Expansion factor of flash storage caused by the buffering policy in BFTL [18]
Sp=2048(bytes)	Size of a Flash page

5.1.3 Performance Comparison

We first compare the four methods under test on each metric with the following parameter setting: N=1 million records, Sk=12, C=5 for BFTL2 (a medium value wrt [18]), and B=7, d=16, k=7 for PBF (which correspond also to medium values). The results are shown in Figures 4(a) to 4(e).

BTree exhibits an excellent lookup performance and consumes little RAM but the price to pay is an extremely high write cost and consequently a very high number of obsolete pages produced (OP). Hence, either the Flash memory usage will be very poor or the garbage collection cost very high. Considering that writes are more time and energy consuming than reads, BTree adapt poorly Flash storage whatever the environment (embedded or not)[3].

BFTL has been primarily designed to decrease the write cost incurred by BTree and Figures 4(c) and 4(e) show the benefit. BFTL1 exhibits the highest benefit in terms of writes and Flash memory usage. However, it incurs an unacceptable lookup cost and RAM consumption given that the node translation lists are not bounded. The IR cost is also very high since each insertion incurs a traversal of the tree. By bounding the size of the node translation lists, BFTL2 exhibits a much better behaviour for metrics R, IR and RAM (though RAM remains high wrt embedded constraints) at the expense of a higher number of writes (to refresh the index nodes) and a higher Flash memory consumption (BFTL mixing valid and obsolete data in the same Flash pages). To better capture the influence of the log size in batch methods, we vary parameter C in Figure 4(f), keeping the preceding values for the other parameters, and study the influence on metrics W, VP and OP. As expected, W and OP (which equals to W) decrease as C increases since the tree reorganizations become less frequent (VP stays equal to 0), up to reach the same values as BFTL1 (equivalent to an infinite C). Conversely, R and RAM grows linearly with C (e.g., R=105 and RAM=2728 when C=30, as shown by formula in Table 5). Trading R and RAM for W and OP is common to all batch methods but there is no trade-off which exhibits acceptable values for RAM, W and OP altogether to meet embedded constraints (Low_RAM, Low_Energy, Low_Storage). Even FlashDB [16] which dynamically takes the best of BTree and BFTL according to the query workload cannot solve the equation.

Though slightly less efficient for lookups than BTree and even BFTL2 when the update/delete rate is high (figure 4(a) shows that metric R for PBF ranges from 10 without update up to 22 with 100% updates)[4], PBF is proved to be the sole indexing method to meet all embedded constraints at once. In this setting, PBF exhibits excellent behaviour in terms of IR, W, VP and OP while the RAM consumption is kept very low. Note that if the RAM constraint is extremely high, the granularity of the buffer could be one sector, as explained in Section 3 and 4.2, leading to a total RAM consumption of 3.5KB[5].

The point is to see whether the same conclusion can be drawn in other settings, and primarily for larger files where sequential methods like PBF are likely to face new difficulties. Figures 4(g) to 4(i) analyse the scalability of BFTL and PBF on R, W and RAM varying N from 1 million up to 7 million records, keeping the initial values for the other parameters (Figures 4(g) and 4(i) use a logarithmic scale for readability). BTree is not further considered considering its dramatically bad behaviour in terms of W and OP.

BFTL2 scales better than PBF in terms of R and even outperforms PBF for N greater than 2.5 million records (though R performance of PBF remains acceptable). However, BFTL2 scales very badly in terms of W. BFTL1 scales much better in terms of W but exhibits unacceptable performance for R and RAM. Unfortunately, PBF scales also badly in terms of W. Beyond this comparison which shows that efficient Flash-based method for indexing very large files still need to be invented, let us see if the scalability of PBF can be improved to cover the requirements of most embedded applications. Actually, the cost of repartitioning becomes dominant for large N and repartitioning occurs at every Flash buffer overflow. A solution for large files is then to increase the size of the Flash buffer hosting the L1 partitions under construction. The comparison between PBF1 and PBF2 on Figure 4(i) shows the benefit of increasing the Flash buffer from 16 pages for PBF1 (that is 4 L1 partitions of 4 pages each) to 64 pages for PBF2 (16 L1 partitions of 4 pages each). Such increase does not impact metric R since the number of reads in L1 partitions does not depend on the number of partitions but of their size (which we keep constant). The RAM impact sums up to 12 more buffers for SKA, but this number can be reduced to only 3 pages by organizing the buffers by sectors. Hence, PBF can accommodate gracefully rather large embedded files (a few millions tuples) assuming the RAM constraint is slightly relaxed (a co-design choice).

[3] The same conclusion can be drawn for other traditional indexing techniques applied to Flash with no adaptation. E.g., for hashing, either the number of buckets is kept very small so that a RAM buffer can be allocated to each and R is very bad (because of the bucket size) or the number of buckets is very high and RAM is very high too.

[4] Note that the R cost for BFTL and BTree neglects the FTL address translation cost which may be high (usually a factor 2 to 3).

[5] For the sake of simplicity, the formulas of the cost model consider the granularity of buffers to be one page.

Table4. Basic formulas of the analytical model

Vars	Annotations	Expressions
Common formulas		
M	Number of index units (IUs) in each page [Note1]	$\lfloor \mathrm{Sp}/(\mathrm{Sk} + 5 * \mathrm{Sr}) \rfloor$ for BFTL1&2, $\lfloor \mathrm{Sp}/(\mathrm{Sk} + \mathrm{Sr}) \rfloor$ for others
Formulas specific to Tree-based methods (Btree, BFTL1, BFTL2)		
ht	Height of B+Tree	$\left| \log_{fb*M+1} N \right|$
Nn	Total number of B+Tree nodes after N insertions	$\sum_{i=1}^{ht} \lceil \mathrm{N}/((fb*M)(fb*M+1)^{i-1}) \rceil$
Ns	Number of splits after N insertions	Nn-ht
L	Average number of buffer chunks that the IUs from the same B+Tree node are distributed to [Note2]	Yao(N, N/(fb*M*B), fb*M) for BTree, Yao(N, β*N/(M*B), fb*M) for BFTL1&2,
α	Number of index units of a logical node stored in a same physical page [Note3]	fb*M/L
Nc	Number of compactions for each node in BFTL2	$\lfloor (\mathrm{L}-1)/(\mathrm{C}-1) \rfloor$
Formulas specific to PBF		
N_{KA}	Total number of pages in KA	$\lceil N/M \rceil$
Sb	Size of a bloom filter (bits)	$2^{\lceil \log_2 (M*d) \rceil}$
Mb	Number of bloom filters in a page	$\lfloor Sp*8/Sb \rfloor$
M1	Number of <key, pinter> pairs contained by one bloom filter	$\lfloor Sb/d \rfloor$
Nr	Total number of partition reorganizations [Note4]	$\lfloor \lfloor N_{KA}/Mb \rfloor/(L1*s) \rfloor$
Pf	Number of last final partitions	$L1*s*2^{\lceil \log_2 (Nr\%(Sb/L1/s)) \rceil}$, if Nr>0, else Pf =0
N_{FB}	Number of pages occupied by the final valid blooms	$\lfloor N/(Sp*8*M1) \rfloor * Sb + Pf + \lfloor (\lfloor N_{KA}/Mb \rfloor \mathrm{mod}(L1*s))/s \rfloor * s$
N_E	Total number of pages which can be erased	$\lfloor N/(Sp*8*M1) \rfloor * \left\lfloor \sum_{i=2}^{Sb/(L1*s)} (2^{\lceil \log_2 (i-1) \rceil} * L1*s) \right\rfloor + \left\lfloor \sum_{i=2}^{Nr\%(Sb/L1/s)} (2^{\lceil \log_2 (i-1) \rceil} * L1*s) \right\rfloor + Nr*L1*s$

[Note1] In BFTL, there are five pointers in each Index Unit (data_ptr, parent_node, identifier, left_ptr, right_ptr), explaining factor 5.

[Note2] Yao's formula is used here to compute how many buffer chunks (1 buffer chunk containing B pages) that fb*M records are distributed to, which is the average length of the lists in node translation table for BFTL1.

[Note3] The IUs from the same logical node are stored in different physical pages, so we divide the total number of IUs (fb*M) by the total number of physical pages to get the average number of IUs stored in the same physical page.

[Note4] L1 denotes the number of pages in each initial L1 partition and L1*s is the size of the Flash buffer used to manage them.

Table 5. Final formulas of the analytical model

Metrics\Methods	BTree	BFTL1	BFTL2	PBF
R [Note1]	ht	(ht-1)*L+L/2	(ht-1)*C+C/2	R1+R2+$\lceil f*N_{KA}*\lceil M1/M \rceil/2 \rceil$+ R3
W [Note2]	N/α+2Ns	β *N/M + β *Ns/2	W1	$N_{KA} + N_{FB} + N_E$
IR [Note3]	IR1	IR2	IR3	N_E
RAM [Note4]	B*Sp/1024	(Nn*L*Sr+B*Sp) /1024	(Nn*C* Sr+B*Sp) /1024	B*Sp/1024
VP [Note5]	Nn	W	W	$N_{KA} + N_{FB}$
OP [Note5]	W-Nn	0	0	N_E

Where, IR1 = Ns*(fb*M/2) + $\sum_{i=1}^{ht-2} i*((fb*M)((fb*M+1)^i - (fb*M+1)^{i-1})+1)+(ht-1)*(N-(fb*M)(fb*M+1)^{ht-2})$

IR2 = Ns*(fb*M/2) + $\sum_{i=1}^{ht-2} i*L*((fb*M)((fb*M+1)^i - (fb*M+1)^{i-1})+1)+(ht-1)*L*(N-(fb*M)(fb*M+1)^{ht-2})$

IR3 = Ns*(fb*M/2) + $\sum_{i=1}^{ht-2} i*C*((fb*M)((fb*M+1)^i - (fb*M+1)^{i-1})+1)+(ht-1)*C*(N-(fb*M)(fb*M+1)^{ht-2})$

W1= β *N/M + β *Ns/2 + β *Nn* $\sum_{i=0}^{Nc-1} (\alpha C + i*(\alpha C - 1))$ /M

R1 = $\lfloor (N_{FB} - Pf)/N_{FB} \rfloor * \lceil ((N_{FB}/L1) \bmod s)/2 \rceil$, R2= $\lceil Yao(Sb/s, Pf/s, \mathrm{k}) \rceil$,R3= $\lceil \mathrm{N}/(\mathrm{Sp}*8*\mathrm{M2})/2 \rceil * Yao(Sb/s, Sb/s, \mathrm{k})$

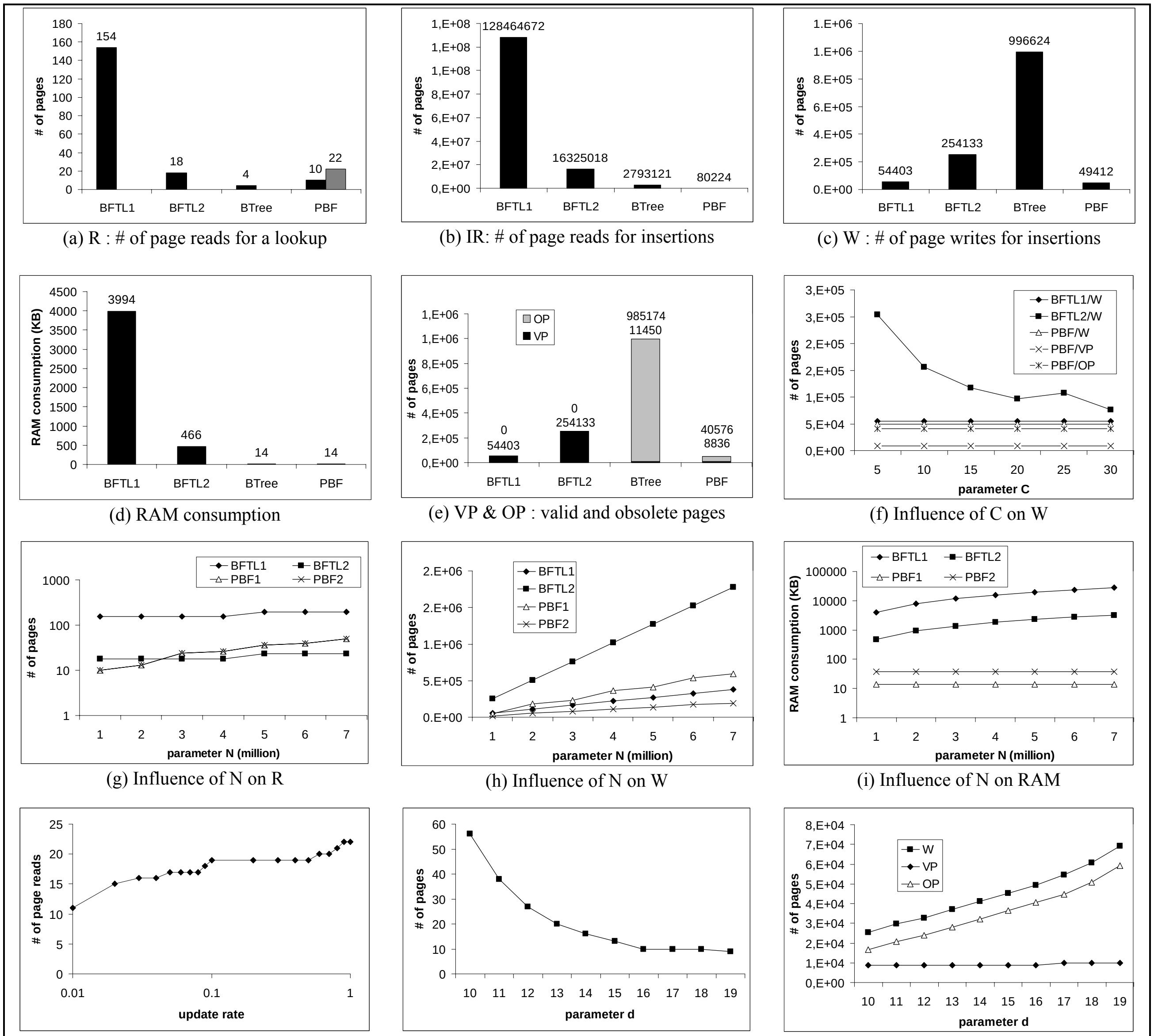
[Note1] For BTree, we did not consider the additional I/Os of going through the FTL indirection table. For BFTL1&2, loading a node requires traversing, in the node translation table, the whole list of IUs belonging to this node and accessing each in Flash. In PBF, the read cost comprises: the lookup in the final and initial partitions and the cost to access (KA), including the overhead caused by false positives.

[Note2] For BTree, the write cost integrates the copy of the whole page for every key insertion (2 times more for splits). BFTL methods also need data copy when doing splits and the write cost of BFTL2 integrates the cost of periodic reorganizations. The write cost for PBF is self-explanatory.

[Note3] For Tree-based methods, the IR cost integrates the cost to traverse the tree up to the target leaf and the cost to read the nodes to be split. For PBF, it integrates the cost to read the partitions to be merged at each iteration.

[Note4] RAM comprises the size of the data structures maintained in RAM plus the size of the buffers required to read/write the data in Flash.

[Note5] VP+OP is the total number of pages occupied by both valid and stale index units. In BFTL1&2, OP=0 simply because stale data are mixed with valid data. By contrast, stale data remain grouped in BTree and PBF. In BTree, this good property comes at a high cost in terms of OP.



(a) R : # of page reads for a lookup

(b) IR: # of page reads for insertions

(c) W : # of page writes for insertions

(d) RAM consumption

(e) VP & OP : valid and obsolete pages

(f) Influence of C on W

(g) Influence of N on R

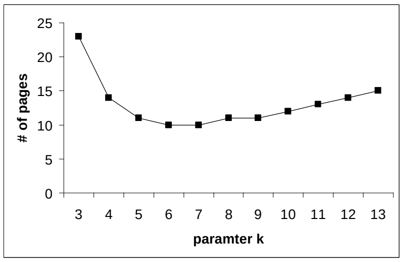
(h) Influence of N on W

(i) Influence of N on RAM

(j) Influence of update rate on R

(k) Influence of d on R

(l) Influence of d on W, VP and OP



(m) Influence of k on R

Table 6. Average R in the experiments

Hash Distrib.	Bob Jenkins	Super-Fast	Arash Partow
Random	12.33	12.28	68.37
Ordinal	12.34	12.32	41.06
Normal	12.28	12.36	76.95

Figure 4. Evaluation results

5.1.4 About Frequent Deletions

As shown is Figure 4(a), large number of random deletions or updates degrades the lookup performance of PBFilter because of the search in DA. Figure 4(j) shows more precisely the impact of random updates/deletions on metric R when there are 1 million valid tuples. It grows with the update rate (number of updates/number of valid tuples) slowly thanks to DA indexing (e.g., for an update rate of 100%, R = 22). This confirms the benefit to build a single DA area for the complete database if RAM buffers needs to be saved.

5.2 PBFilter Adaptability and Predictability

Tuning parameters d and k used to build the Bloom filters in PBF determines the quality of the summarization (false positive rate), the size of the summary and then the partitioning cost with a direct consequence on metrics R, W, VP and OP. This makes PBF adaptable to various situations and brings high opportunities in terms of co-design, assuming the consequences of tuning actions can be easily predicted and quantified.

Figure 4(k) shows the influence of *d* on R with the other parameters set to the previous values: N=1million, Sk=12, B=7, k=7. As expected, the bigger *d*, the smaller the false positive rate and then the better R. At the same time, larger Bloom filters increase the frequency of repartitioning and then increase W and OP in the proportion shown in Figure 4(l). The impact on VP is however very limited because of the small size of SKA compared to KA (e.g., for d=16 and k=7, |SKA|/|KA| = 0.1). Figure 4(m) shows the influence of *k* on R with d=16. The bigger k, the better R, up to a given threshold. This threshold is explained by the Bloom filter principle itself (see formula in Section 4.1 showing that there is an optimal value for k beyond which the false positive rate increases again) and by the fact that a bigger k means scanning more partitions in SKA, a benefit which must be compensated by lower accesses in KA.

As a conclusion, *d* introduces a very precise trade-off between R and W, VP, OP, allowing adapting the balance between these metrics to the targeted application/platform tandem. The choice of *k* under a given *d* should minimize i*k+f*|KA|/2, where i is the number of pages in each final partition.

To illustrate this tuning capability, let us come back to the management of large files. Section 5.1.3 presented a solution to increase PBF scalability in terms of W. The scalability in terms of R can be also a concern for some applications. The decrease of R performance for large files is due to the increase of the number of pages in each final partition and of the average accesses to KA which is f*|KA|/2. The growth of the size of each final partition can be compensated by a reduction of k and a smaller f can still be obtained by increasing d. For instance, the values d=24 and k=4 produce even better lookup performance for N=5 million records (R=9) than the one obtained with d=16 and k=7 for N=1 million records (R=10). The price to pay in terms of Flash memory usage can be precisely estimated thanks to our cost model.

5.3 Experimental Results on Real Hardware

5.3.1 Platform Description

PBFilter has been implemented and integrated in the storage manager of an embedded DBMS dedicated to the management of secure portable folders [1]. The prototype runs on a secure USB Flash platform provided by Gemalto, our industrial partner. This platform is equipped with a smartcard-like secure microcontroller connected by a bus to a large (Gigabyte-sized soon) NAND Flash memory (today the 128MB Samsung K9F1G08X0A module), as shown in Figure 5.

The microcontroller itself is powered by a 32 bit RISC CPU (clocked at 50 MHz) and holds 64KB of RAM (half of it preempted by the operating system) and 1MB of NOR Flash memory (hosting the on-board applications' code and used as write persistent buffers for the external NAND Flash).

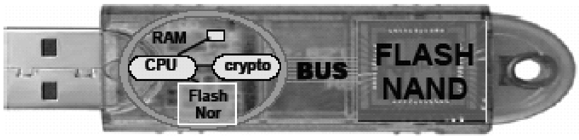


Figure 5. Secure USB Flash device

There are three nested API levels to access the NAND Flash module: FIL (Flash Interface Layer) providing only basic controls such as ECC, VFL (Virtual Flash Layer) managing the bad blocks and FTL (Flash Translation Layer) implementing the address translation mechanism, the garbage collector and the wear-leveling policies. We measured the cost of reading/writing one sector/page through each API level using sequential (seq.) and random (rnd.) access patterns. The numbers are listed in Table 7 and integrate the cost to upload/download the sector/page to the Flash module register and the transfer cost from/to the RAM of the microcontroller (masking part of the difference in the hardware cost). FIL and VFL behave similarly for sequential and random access patterns while the variation is significant with FTL. Random writes exhibit dramatic low performance with FTL (a behavior we actually observed in many Flash devices).

Table 7. I/O Performance through different API levels

API Levels	R(μs) sector/page	W(μs) sector/page
FIL(seq. & rnd.)	100/334	237/410
VFL(seq. & rnd.)	109/367	276/447
FTL(seq.)	122/422	300/470
FTL(rnd)	380/680	≈ 12000

5.3.2 Experimental Results

We ran our prototype under all the parameter settings used in 5.1 and 5.2. We measured the I/Os and compares the results with those produced by the cost model.

Unsurprisingly, the tests produced exactly the same numbers as those computed by the cost model for all metrics but R. Indeed, the sequential organization and the fixed size of all data structures make the insertion process and the number of repartition steps fully predictable for a given parameter setting, avoiding any uncertainty for IR, W, VP and OP metrics (In the prototype, transaction atomicity is guaranteed thanks to internal NOR Flash buffers and do not interfere with the NAND Flash management).

The discrepancy related to the R metric deserves a deeper discussion. The cost model computes the false positive rate using the formula given in 4.1, assuming the k hash functions are totally independent, a condition difficult to meet in practice. Much work [7, 12] has been done to build efficient and accurate bloom filter hash functions. In our experiment, we compared Bob Jenkins' lookup2, Paul Hsieh's SuperFastHash, and Arash Partow hash over datasets of different distributions (random, ordinal and normal) produced by Jim Gray's DBGen generator. The results show that the degradation of the false positive rate is quite acceptable for the former two hash functions but not for the latter. Table 6 shows the R metric measured for each hash function and data distribution under the setting: N=1 million, Sk=12, d=16, k=7 (the cost model gives R=10 for this setting). About the efficiency of hash functions, Bob Jenkins and SuperFastHash are quite fast (6n+35 and 5n+17 cycles respectively, where n is the key size in bytes), and k independent hash values can be obtained by calling only three times the hash function [7].

We have done preliminary performance measurements in terms of response time for insertions and lookups on top of different API levels. Today, we are not granted permission by our industrial partner to publish absolute performance numbers, other than those given in Table 7, due to a pending patent. However, the preliminary observations show that (1) the average insertion cost of PBFilter is low in every situations (even on top of FTL) due to its sequential write feature, (2) the lookup cost is very satisfactory on top of FIL and VFL with an increase of nearly 70% on top of FTL and (3) the CPU cost remains low (less than 15% of the total) despite the low frequency of the microcontroller. Further experiments are required to fully capture the behaviour of PBFilter on this hardware platform considering different NAND Flash APIs and variant datasets. We expect that real numbers would be made public soon.

6. CONCLUSION

NAND Flash has become the most popular stable storage medium for embedded systems and efficient indexing methods are highly required to tackle the fast increase of on-board storage capacity. Designing these methods is complex due to a combination of NAND Flash and embedded system constraints. To the best of our knowledge, PBFilter is the first indexing method addressing specifically this combination of constraints.

The paper introduces a comprehensive set of metrics to capture the requirements of the targeted context. Then, it shows that batch methods are inadequate to answer these requirements and proposes a very different way to index Flash-resident data. PBFilter, organizes the index structure in a pure sequential way and speeds up lookups thanks to Summarization and Partitioning. A Bloom filter based instantiation of PBFilter has been implemented and a comprehensive performance study shows its effectiveness.

PBFilter is today integrated in the storage manager of an embedded DBMS dedicated to the management of secure portable folders. Thanks to its tuning capabilities, PBFilter seems adaptable to various Flash-based environments and application requirements. Typically, PBFilter seems well adapted to any RAM constrained environment, embedded or not. Our future work is to complete performance measurements on real hardware, to propose an accurate management of secondary keys and to investigate new summarization and partitioning strategies to ever enlarge PBFilter application domain.

7. ACKNOWLEDGMENTS

The authors wish to thank Luc Bouganim, Dennis Shasha and Björn Þór Jónsson for fruitful discussions on this paper and Jean-Jacques Vandewalle and Laurent Castillo from Gemalto for their technical support. This research is partially supported by the French National Agency for Research (ANR) under RNTL grant PlugDB and by the Natural Science Foundation of China under grants 60833005, 60573091.

8. REFERENCES

[1] Anciaux, N., Benzine, M., Bouganim, L., Jacquemin, K., Pucheral, P., and Yin, S. Restoring the Patient Control over her Medical History. *21th IEEE Int. Symposium on Computer-Based Medical Systems (CBMS)*, 2008.

[2] Anciaux, N., Bouganim, L., Pucheral, P., Valduriez, P. DiSC: Benchmarking Secure Chip DBMS. *IEEE Transactions on Knowledge and Data Engineering (IEEE TKDE), vol. 20, n°10, 2008*.

[3] Birrel, A., Isard, M., Thacker, C., and Wobber, T. A Design for High-Performance Flash Disks. *Operating Systems Review* 41(2), 2007.

[4] Bityutskiy, A-B., JFFS3 Design Issues. *Tech. report*, Nov. 2005.

[5] Bloom, B. Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM, 13(7)*, 1970.

[6] Dekart SRL.: Dekart Smart Container, 2007. http://www.dekart.com/products/integrated/smart container

[7] Dillinger, P. C., and Manolios, P. Fast and Accurate Bitstate Verification for SPIN. *11th Int. Spin Workshop on Model Checking Software,* LNCS 2989, 2004.

[8] Gonnet, G. and Baeza-Yates, R. *Handbook of Algorithms and Data Structures,* Addison-Wesley, Boston, MA, USA, 1991.

[9] Hamid L. New directions for removable USB mass storage, Press release, 2006. http://www.itwales.com/997893.htm

[10] Intel Corporation, Understanding the Flash Translation Layer (FTL) specification. 1998.

[11] Kim, G., Baek, S., Lee, H., Lee, H., and Joe, M. LGeDBMS: A Small DBMS for Embedded System with Flash Memory. *Int. Conf. on Very Large Data Bases (VLDB),* 2006.

[12] Kirsch, A., and Mitzenmacher, M. Less Hashing, Same Performance: Building a Better Bloom Filter. *Algorithms – ESA 2006, 14th European Symposium,* LNCS 4168, 2006.

[13] Lee, S-W., and Moon, B. Design of Flash-Based DBMS: An In-Page Logging Approach. *Int. Conf. on Management of Data (SIGMOD)*, 2007.

[14] Mani, A., Rajashekhar, M. B., and Levis, P. TINX - A Tiny Index Design for Flash Memory on Wireless Sensor Devices. *ACM Conf. on Embedded Networked Sensor Systems (SenSys)* 2006, Poster Session.

[15] Mitzenmacher, M. Compressed Bloom Filters. *Proceedings of ACM PODC*, 2001.

[16] Nath, S., and Kansal, A. FlashDB: Dynamic Self-tuning Database for NAND Flash. *Int. Conf. on Information Processing in Sensor Networks (IPSN)*, 2007.

[17] Rosenblum, M., and Ousterhout, J. K. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems (TOCS)* 10(1), 1992.

[18] Wu, C., Chang, L., and Kuo, T. An Efficient B-Tree Layer for Flash-Memory Storage Systems. Int. Conf. on *Real-Time and Embedded Computing Systems and Applications (RTCSA)*, 2003.

[19] Yao, A. On Random 2-3 Trees. *Acta Informatica*, 9 (1978).

[20] Yao, S. Approximating the Number of Accesses in Database Organizations. *Communication of the ACM 20(4),*1977.

[21] Yin, S., Pucheral, P., Meng X. PBFilter: Indexing Flash-Resident Data through Partitioned Summaries. Tech. Rep. RR-6548. INRIA. 2008.

[22] Zeinalipour-Yazti, D., Lin, S. V., Kalogeraki, Gunopulos, D., and Najjar, W. MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices. *USENIX Conf. on File and Storage Technologies (FAST)*, 2005.