PBFilter: Indexing Flash-Resident Data through Partitioned Summaries

Shaoyi Yin, Philippe Pucheral, Xiaofeng Meng

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

PBFilter: Indexing Flash-Resident Data through Partitioned Summaries

Shaoyi Yin, Philippe Pucheral, Xiaofeng Meng

*R apport
de recherche*

PBFilter: Indexing Flash-Resident Data through Partitioned Summaries

Shaoyi Yin[1], Philippe Pucheral[2], Xiaofeng Meng[3]

Abstract: NAND Flash has become the most popular persistent data storage medium for mobile and embedded devices and is even being considered as a credible competitor for traditional disks. The hardware characteristics of NAND Flash (e.g. page granularity for read/write with a block-erase-before-rewrite constraint, limited number of erase cycles) preclude in-place updates. Previous works adapted traditional indexing methods to cope with these constraints mainly by deferring index updates thanks to a log and batching them to decrease the number of rewrite operations in Flash. These methods introduce a complex trade-off between read and write performance and neglect negative side-effects on the RAM consumption, Flash consumption and garbage collection cost. In this paper, we propose a new alternative for indexing Flash-resident data, designed from the outset to exploit the peculiarities of NAND Flash. This approach, called PBFilter, organizes the index structure in a pure sequential way to avoid the side-effects mentioned above. Key lookups are sped up thanks to two principles called Summarization and Partitioning. We instantiate these principles by data structures and algorithms based on Bloom Filters and show the effectiveness of this approach through a comprehensive performance analysis. PBFilter can be instantiated with different Summarization and Partitioning algorithms opening up a new way to think about indexing Flash-resident data.

Keywords: Indexing method, NAND Flash, Bloom Filter, cost model.

[1] Author1 affiliation – shaoyi.yin@inria.fr
[2] Author2 affiliation – philippe.pucheral@inria.fr
[3] Author3 affiliation – xfmeng@ruc.edu.cn

PBFilter: Indexing Flash-Resident Data through Partitioned Summaries

Résumé: La mémoire de type Flash NAND s'est imposée comme le plus populaire medium de stockage pour les équipements mobiles et embarqués et est même considérée comme un concurrent crédible pour les disques traditionnels. Les caractéristiques matérielles de la Flash NAND (granularité page pour la lecture/écriture avec une contrainte de type block-erase-before-rewrite, nombre limité de cycles d'effacement, etc) empêchent les mises à jour en place. Les travaux précédents relatifs à l'indexation de données en Flash ont adapté les méthodes d'indexation traditionnelles pour faire face à ces contraintes, principalement en différant les mises à jour dans l'index grâce à un journal et en les groupant pour réduire le nombre de réécritures. Ces méthodes introduisent un compromis complexe entre la performance des lectures et des écritures et négligent les effets de bord sur la consommation de RAM, la consommation de Flash et le coût de collecte des miettes. Dans cet article, nous proposons une nouvelle alternative pour indexer les données résidant en Flash, conçue dès le départ pour exploiter au mieux les particularités de la Flash NAND. Cette approche, appelée PBFilter, organise la structure de l'index de façon purement séquentielle pour éviter les effets de bord mentionnés ci-dessus. Les recherches sur clé sont accélérées grâce à deux principes appelés Résumé (Summarization) et Partitionnement (Partitioning). Nous instancions ces principes par des structures de données et algorithmes basés sur les Bloom Filters et montrons l'efficacité de cette approche par une analyse de performances. PBFilter peut être instancié avec différents algorithmes de Résumé et de Partitionnement apportant une nouvelle façon de concevoir l'indexation de données résidant en Flash.

Mots clés: méthode d'indexation, Flash NAND, Bloom Filter, modèle de coût.

1 Introduction

NAND Flash memory stands out as the best adapted storage medium for an ever wider spectrum of mobile and embedded devices (PDAs, cell phones, cameras, sensors, smart cards, USB keys, mp3 readers etc). The reasons for this success include shock resistance, size, energy consumption, performance and ease of on-chip integration. As capacity increases and price decreases, Flash memory is being considered as a mid-term serious competitor even for traditional disks [2].

However, NAND Flash exhibits specific hardware characteristics making database management very challenging. Reads and writes are done at a page granularity, as in traditional disks, but writes are more time and energy consuming than reads. In addition, a page cannot be rewritten before erasing the complete block containing it, a costly operation. Finally, a block wears out after 10^5 to 10^6 repeated write/erase cycles. Due to this, updates are usually performed "out-of-place", meaning that a page is copied in another location before being modified, entailing address translation and garbage collection overheads.

Database storage models dedicated to NAND Flash have been proposed in [10, 12]. The strong point of IPL [12] is to offer a storage and buffering approach hiding the Flash peculiarities to the upper layers of the DBMS. The idea is delaying the updates in Flash thanks to a log stored in each physical block and rebuilding the accurate version of each page at load time. The updates are physically applied to a page when the corresponding log region becomes full. While elegant, this general method is not well suited to manage hot spot data in terms of updates, like index pages, because of frequent log overflows. Recent works addressed this issue by adapting B+Tree-like structures to NAND Flash [3, 15, 19]. While different in realization, these methods rely on a similar approach: to delay the index updates thanks to a log dedicated to the index and batch them with a given frequency with the objective to group updates related to a same index node. We call these methods *batch methods* hereafter. Batch methods entail a new trade-off between the number of reads and writes but good performance for both is difficult to achieve at once due to the Flash characteristics. In addition, read/write performance is no longer the sole metrics of interest in the Flash context. First, RAM consumption is a primary concern in many mobile and embedded devices, today the main target of Flash chip manufacturers. Second, the data organization in Flash impacts the costs of address translation and stale data reclamation. These indirect costs have been shown high and not easily predictable [15]. Hence, we argue that a larger set of metrics (including RAM consumption and Flash usage indirect costs) has to be considered when designing an indexing technique for Flash-resident data. We also believe that a thorough rethinking of the indexing methods is mandatory to tackle accurately the Flash constraints rather than adapting traditional index structures initially designed for supporting random I/O.

In this paper, we propose a new indexing approach designed to exploit natively the peculiarities of NAND Flash. NAND Flash is best suited to support sequential writes and page-level reads. Hence, the indexing approach proposed in this paper, called PBFilter, relies on producing only sequential writes and page-level reads. Let us imagine an index organized as a pure sequential append-only list of (key, pointer) pairs. This would avoid "out-of-place" updates, and then save address translation and stale data reclamation costs, provided a simple buffering strategy is followed. The question becomes how to look up a given key in a sequential list with acceptable performance? The PBFilter approach relies on two principles. *Summarization* consists in building a sequential summary of the (key, pointer) list and in doing the lookup first in this summary to determine the region of interest of the list. Different summarization algorithms can be envisioned introducing an interesting trade-off between the compression ratio of the summary and its accuracy. *Partitioning* consists in splitting a sequential structure (e.g., the key-pointer list or its summary) vertically in such a way that only a subset of partitions need to be scanned at lookup time. Again, different partitioning strategies can be devised introducing a trade-off between the number of partitions and the RAM consumption. The key idea behind Summarization

and Partitioning is that both help speeding up the lookup without hurting sequential writes and page-level reads.

The contribution of this paper is threefold:

- To propose a new approach based on Summarization and Partitioning for indexing Flash-resident data, which: (1) avoids the penalty of address translation and garbage collection, (2) minimizes RAM consumption and (3) makes read/write performance and resource consumption (RAM and Flash) fully predictable.

- To instantiate this generic approach with concrete data structures and algorithms based on Bloom Filters [4] and demonstrate that the preceding objectives can be reached while maintaining very good lookup performance.

- To propose a comprehensive set of metrics capturing the behavior of Flash-based indexing methods and compare them through a precise analytical performance model.

In addition, as PBFilter can be instantiated with different Summarization and Partitioning algorithms, we expect that this work can open up interesting research directions related to the indexing of Flash-resident data.

The paper is organized as follows. Section 2 introduces the main characteristics of NAND Flash, studies the related work and states the problem addressed in this paper. Section 3 details the PBFilter indexing scheme. Section 4 presents a PBFilter instantiation through partitioned Bloom Filter summaries. Section 5 describes our analytical performance model, gives comparisons with batch methods, and shows preliminary experimental results. Section 6 sketches interesting open issues and Section 7 concludes.

2   Problem statement

2.1   NAND Flash characteristics

Flash devices come today in various form factors such as compact flash cards, secure digital cards, smart cards, USB token, and even as composite chip modules providing a disk like bus interface. Whatever the form factor and the manufacturer, Flash chips share common characteristics. A typical NAND Flash array is divided into blocks (the number of which depends on the Flash module capacity), in turn divided into pages (e.g., 32-64 pages per block) themselves divided into sectors (e.g., 4-8 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 512-2048 bytes. A page can only be rewritten after erasing the entire block containing it (usually named *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 10^5 to 10^6 repeated write/erase cycles, requiring write load to be spread out evenly across the memory.

These hardware constraints make the management of updates complex. This complexity is slightly mitigated by the decomposition of a page into sectors. Sectors can be written independently in the same page, allowing one write per sector before requiring erasing the block. To avoid erasing blocks too frequently, updates are usually performed "out-of-place", meaning that a page to be updated is copied in another block and its initial image becomes garbage. This introduces the need for: (1) a translation mechanism relocating the pages and making their address invariant through indirection tables and (2) a garbage collection mechanism erasing blocks, either lazily (waiting that all pages of the block become stale) or eagerly (moving the active pages still present in the block before erasing it). Both mechanisms are combined in a Flash Translation Layer (FTL), usually integrated in the firmware of the Flash device [9].

For illustration purpose, Table 1[4] presents the execution time and energy consumption of various Flash devices for read and write operations at page granularity. The last row reports raw numbers, showing the performance of the hardware only. We can note very high discrepancies between the platforms, both in absolute times and in terms of the read/write ratio, with the commonality that writes are always more expensive than reads. As precisely studied in [15] these discrepancies are explained partly by the raw chip characteristics and partly by the firmware managing the FTL. Raw chip characteristics are public but FTL are usually proprietary and constitute the primary source of performance unpredictability.

Table 1. Perf. and energy consumption of Flash devices

Device	R(μJ)	W(μJ)	R(μs)	W(μs)
Compact Flash 512M	2970	6220	18000	29000
Mini SD 512M	109	22292	1100	193000
Samsung 128MB (raw)	0.74	9.9	15	200

2.2 Related work

Several storage and indexing models have been specifically designed for EEPROM and NOR Flash memories [5, 18]. Those models consider memories with a byte or word granularity for read and write operations, making them irrelevant in a NAND Flash context. Some other work [10, 12] 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, [12] proposed a storage and buffering model enabling existing DBMS to run on Flash with a minimal adaptation of the engine, but this solution is sub-optimal for hot-spot data structures like indexes, as explained in Section 1.

With the broadening of NAND Flash usage, recent work has considered more specifically the indexing problem in NAND Flash. As for traditional indexes, we can distinguish the hash-based and tree-based families of indexing methods. Little attention has been paid so far to hash-based methods in Flash. The reason is most probably that hashing performs well only when the number of buckets can be made high and when the RAM can accommodate one buffer per bucket, but RAM is often a scarce resource in Flash-based devices. One exception is the Microhash indexing structure designed to speed up lookups in sensor devices [22]. In this particular context, sensed data vary on a small domain (e.g., temperature values) so that the number of buckets can remain low. However, this method is not general and not adapted to large domain indexes like indexes on primary keys.

In the tree-based family, one work considers also the indexing of sensed data [13]. This work proposes a tiny index called TINX based on a specific unbalanced binary tree. The performances revealed by the authors (e.g., 2500 page reads to retrieve one record among 0.6 million records) disqualify this method for large databases. 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 Flash characteristics [19]. Indeed, each time a new record is inserted into a file, its key must be added to a B+Tree leaf node, incurring an out-of-place update of the corresponding Flash page. To avoid such updates, BFTL [19] 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

---

[4] This table is taken from [15]. The numbers of the last row are close to the Samsung NAND Flash module K9F1G08X0A we used for our own experiments.

index units belonging to this node. In order to bound the size of these lists and therefore the RAM consumption as well as the lookup cost, each list is compacted when a threshold (e.g., 5 log pages in the list) is reached. At this time, the logged updates are batched to refresh the physical image of the corresponding B+Tree node.

FlashDB [15] mixes Regular B+Tree and BFTL to take the best of each, using a self-tuning principle linked to the query workload. JFFS3 proposes a slightly different way to optimize B+Tree usage [3]. 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 in the journal index and then in the B+Tree.

In short, all the B+Tree based methods rely on the same principle: (1) to delay the index updates thanks to a log and batch them with the objective to group updates related to the same index node; (2) to build a RAM index at boot time to speed up the lookup of a key in the log; (3) to commit the log updates with a given commit frequency (CF) in order to bound the log size. This introduces the following trade-off:

– The larger CF, the smaller the number of writes but the higher the number of reads (both at lookup and insertion time).

Unfortunately, this trade-off is more difficult to capture and to exploit than in a traditional (disk-based) context. First, writes generate out-of-place updates entailing an indirect garbage collection cost. This cost depends on the strategy implemented in the underlying FTL, with large discrepancies among the platforms [15]. Second, the way the index structure is organized in Flash also impacts the garbage collection cost and the Flash occupancy, depending on whether stale data is grouped in a small set of pages or is spread over all pages occupied by the index. Third, the larger CF, the larger the RAM consumption. This consequence is often ignored while it may definitely proscribe the use of an indexing method for a particular platform.

2.3 Problem statement

In the light of the preceding discussion, more complete and accurate metrics seem mandatory to help assessing the adequacy of an indexing method to a target platform. We propose evaluating a Flash-based indexing method according to the following metrics:

– **M1:** Read cost at lookup time.

– **M2:** Read and write cost at insertion time.

– **M3:** Flash usage in terms of space occupancy and effort to reclaim stale data.

– **M4:** RAM consumption.

– **M5:** Performance and resource consumption predictability.

M1 and M2 are the usual performance metrics for indexing methods. The trade-off between M1 and M2 is tied to the application (read or write intensive). M2 prioritizes insertions, usually more frequent than deletes or updates in database applications. Depending on the targeted platform, the metrics of interest can be execution time or energy consumption. To answer both requirements, M1 and M2 are expressed in terms of number of read and write operations.

The objective of metric M3 is to capture (1) total amount of Flash occupied by the index and (2) indirect cost incurred by garbage collection, used to decrease this amount by reclaiming stale data. Indirect costs are by nature difficult to estimate and have been shown directly tight to the underlying FTL [15]. To tackle this issue, we suggest expressing metric M3 by two values, namely the total number of active pages occupied by the index (i.e., pages containing at least one valid item) and the total number of pages containing only stale data (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 about the quality of the Flash occupancy and about the effort to reclaim stale space, independently of the FTL implementation.

Metric M4 is of utmost importance for most embedded and mobile devices (smart cards, sensors, USB dongles, etc) which are today the primary targets of Flash storage. To illustrate this, secure chips and sensors are equipped with Kilo-bytes sized RAM. Even if not all platforms exhibit so severe RAM constraints, we attach a particular importance to this metric because it may definitely disqualify certain techniques. Metric M4 is expressed in KB and comprises the buffers to read from and write to the Flash and the main memory data structures required by the indexing method.

Finally, metric M5 captures the performance and resource consumption predictability. Predictability is an important concern in database systems in general and in Flash-based DBMS in particular considering the discrepancy among FTL implementations. To avoid making this metric fuzzy by reducing it to a single number, M5 is qualitative instead of quantitative and is spit into three dimensions. First, we attach four ranking to the dependence of the indexing method wrt the underlying FTL, considered as the primary source of performance uncertainty: totally dependent of the FTL, independent of the FTL's garbage collection mechanism (i.e., no stale data is ever produced or stale data is reclaimed by the indexing method itself), independent of the FTL's address translation mechanism (i.e., valid data never moves or moves are managed by the indexing method itself), totally independent of the FTL. The second dimension of performance predictability captures whether the values measured for M1 and M2 can be bounded whatever their absolute value (Boolean metric). Finally, resource consumption predictability is also a Boolean metric expressing whether the RAM requirement can be bounded whatever its absolute value.

Finding the best trade-off between these five metrics is application and platform dependent. However, it is important to study all the metrics, because (1) a dramatically bad behavior on a metric may disqualify an indexing method and (2) this assures that all metrics of interest have been considered for the targeted context.

The more metrics are positively met, the wider the application domain is. Typically, in the embedded device context where resources are highly constrained, the five metrics are significant. To the best of our knowledge, no existing Flash-based indexing method performs well on all the metrics.

3 PBFilter indexing scheme

PBFilter is an alternative to the "batch" indexing methods. The general idea is to perform index updates eagerly and make this acceptable by organizing the complete database as a set of sequential data structures, as pictured in Figure 1. The objective is to transform database updates into append operations so that writes are produced sequentially, an optimal scenario for NAND Flash.

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[5] and pt is the record physical address. If a record 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 in 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 database updating process never incurs rewriting a same page in Flash.

The benefits provided by this simple database organization are obvious with respect to the met-

---

[5] Like all state of the art methods mentioned in Section 2, we concentrate the study on primary keys. The management of secondary keys is briefly addressed in Section 6.

rics introduced in Section 2. *Metric M2:* a lower bound is reached in terms of reads/writes at insertion time since: (1) the minimum of information is actually written in Flash (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 and (3) pages do not move, thus physical pointers can be used and FTL is not required, saving the overhead incurred by the FTL address translation mechanism[6]. *Metric M3:* neglecting DA, a lower bound is reached in terms of Flash space occupancy, again because the information written is minimal and never updated. Hence, the number of active pages containing the index is equivalent to the raw size of this index. The size of DA measures the space overhead incurred by this method (expected to be low considering that deletes are not frequent in most databases). In addition, since a Flash page is never updated nor moved, no stale data is ever produced and the garbage collection cost is saved. *Metric M4:* a single RAM buffer of one page is required per sequential structure (RA, KA and DA). The buffer size can even be reduced to the size of a Flash sector in highly constrained environments (the benefit of a single I/O per page is lost but a page is still never moved making the garbage collection needless). *Metric M5:* PBFilter can bypass the FTL address translation layer (see footnote) so that performance uncertainties linked to the FTL are removed. In terms of resource consumption, the RAM usage is bounded as discussed above.

The drawbacks of this organization are obvious as well. The performance reached for metric M1 is far from being satisfactory since the read cost is ($|KA|/2 + |DA| + 1$) on the average, where $|\ |$ denotes the page cardinality of a structure. Consequently, despite the removal of the FTL costs, performance predictability (metric M5) is not achieved since the uncertainty of the read cost at lookup time (for an existing record) is up to ($|KA| - 1$).
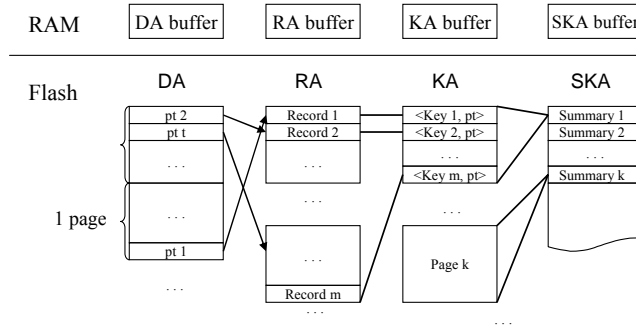

Figure 1. Database organization

Suggesting an indexing method entailing such bad and unpredictable lookup cost is senseless at first glance. The objective is to circumvent these two drawbacks with a minimal degradation of the benefits listed above. To this end, we introduce two additional principles.

The first one is *Summarization*. Summarization refers to any method which can condense the information present in KA and can produce the condensed information sequentially. 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., percentage 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

---

[6] Several FTL provide different API levels (e.g., FTL, VFL, FIL levels in Samsung S-SIM FTL) allowing bypassing some functionalities (e.g., bad block management, ECC management, address translation, etc).

net effect of summarization is replacing the lookup cost in KA by ($|KA|/2c + f|\,|KA|\,|/2$) on the average, where $|\,|\,|$ denotes the element cardinality of a structure. The positive impact on metrics M1 and M5 can be very high for favorable values of c and f. The negative impact on M4 is limited to a single new buffer in RAM. The negative impact on M2 and M3 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, the only requirement being to respect the following property: *summaries must allow membership tests with no false negatives*.

The second principle is *Partitioning*. The idea behind partitioning is to vertically split a sequential structure into p sequential partitions so that only $p'<p$ partitions have 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 metrics M1 and M5. 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*.

The question is whether condenser algorithms and partitioning strategies satisfying the requirements above actually exist and make the general approach viable. In the next section, we present one solution for each problem and show that, taken together, they lead to an approach satisfying the problem stated in Section 2, that is exhibiting at least a satisfactory behavior for all metrics at once. We do not argue that no other or better solutions exist. On the contrary we expect that this study is the first step towards the definition of a family of solutions.

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 [4]. Hence, it presents all the characteristics required for a condenser.

A Bloom filter to represent a set $A=\{a_1, a_2, \ldots a_n\}$ of n elements is described by a vector v of m bits, initially all set to 0. The Bloom filter uses k independent hash functions, $h_1, h_2, \ldots h_k$, each producing an integer in the range [1,m]. For each element $a_i \in A$, the bits at positions $h_1(a_i)$, $h_2(a_i), \ldots, h_k(a_i)$ in v are set to 1 (a particular bit might be set to 1 multiple times). Given a query for element a_j, all bits at positions $h_1(a_j), h_2(a_j), \ldots, h_k(a_j)$ are checked. If any of them is 0, then a_j cannot be in A (Bloom filter do not generate false negatives). 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].

This probability, called the *false positive rate* and denoted by f in the sequel, can be calculated easily assuming the k hash functions are perfectly 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 $\left(1-1/m\right)^{kn} \approx e^{-kn/m}$. The probability of a false positive is then $\left(1-\left(1-1/m\right)^{kn}\right)^k \approx \left(1-e^{-kn/m}\right)^k = \left(1-p\right)^k$ 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 2 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].

Table 2. 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

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. The computed Bloom filter summary is stored in the RAM buffer allocated to SKA. In turn, the SKA buffer is flushed to the Flash when full. At lookup time, the searched key a_i is hashed with the *k* hash functions (step 1). 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 (step 2). The corresponding page of KA is directly accessed (step 3) and the probability that it contains the expected index entry (a_i,pt) is (1-f). Otherwise, the scan continues in SKA. A fourth step is required to check that pt ∉DA before accessing the record in RA. We make the assumption that inserts vastly outnumber deletes so |DA| is small enough to have little effect on the total read cost of the lookup. We believe that this assumption is usually met in the database context but Section 6 discusses an optimization if it turns out to be wrong.

4.2 Partitioned Summaries

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 then 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. As already stated, the larger *p*, the higher the partitioning benefit but also the greater the RAM consumption (*p* buffers) or the greater the number of writes and the need for garbage collection (if the number of buffers is less than p, buffers must be flushed while the corresponding partition pages are not full, entailing several flushes in the same page, then page rewrites).

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, stale data are naturally grouped in the same blocks which can be erased as a whole at low cost). This 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 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). 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 Flash page 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≤i≤s) of this buffer is flushed in the first sector of the first page of the ith L1 partition.

The second flush of the SKA buffer will fill in the second sector of the first page of each L1 partition 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 $[((i-1)*m/s^2 + 1) .. (i*m/s^2)])$.
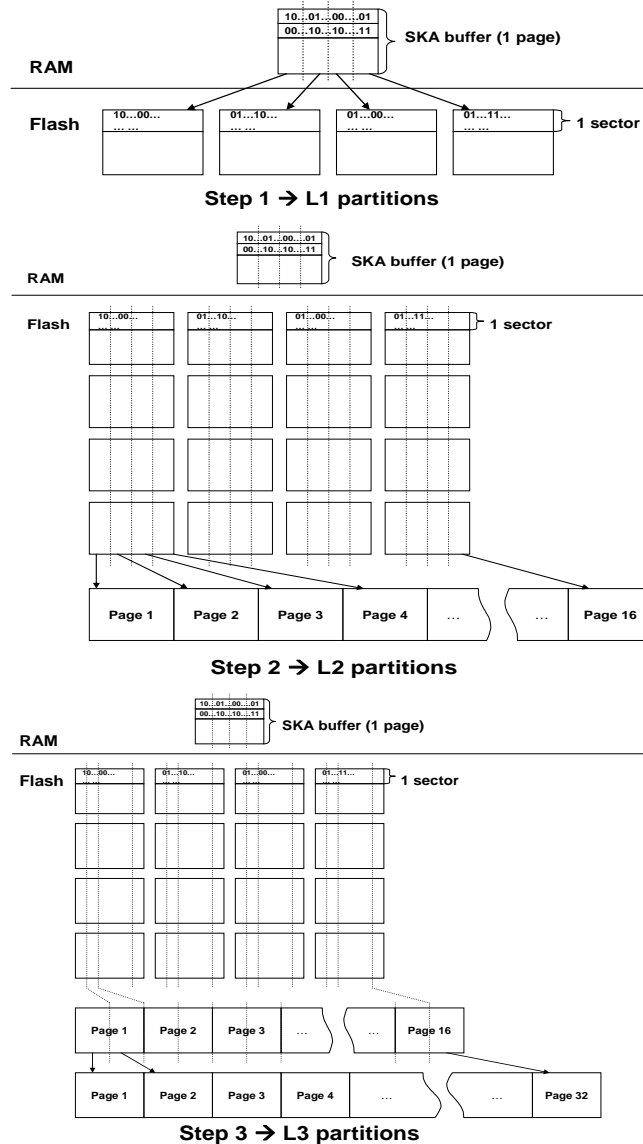
Figure 2. Dynamic partitioning

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).

The benefit of partitioning dynamically SKA is significant with respect to metrics M1 and M5 since the lookup cost can be made low and almost constant whatever the size of the indexed file. Indeed, 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 $(k + \min(k, s) * s/2)$, usually with $s=4$. The RAM consumption (metric M4) 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 metric M2 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/m records, where B_p is the size of a Flash page in bits (B_p/m is likely to be greater than 1000). Section 5 will show that this extra cost is low compared to the insertion cost of existing indexing techniques. Section 5 will also show the low impact of partitioning on metric M3 for the same reason, that is the high compression ratio obtained by Bloom filters making SKA small with respect to KA.

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 stale 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 IUs 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 stale.

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.

5 Performance evaluation

5.1 Performance Analytical Model

5.1.1 *Indexing methods under test*

This section compares quantitatively the Bloom filter instantiation of PBFilter, denoted by PBF hereafter, with representative related works: (1) regular B+Tree running on top of FTL, denoted by BTree; (2) BFTL with no compaction of the node translation table, denoted by BFTL1 and (3) BFTL with periodic compaction of the node translation table, denoted by BFTL2. BTree is the best representative of traditional disk-based indexing methods and will serve as a point of reference to assess the benefit of designing specific indexing methods for Flash. BFTL2 [19] is the best known representative of batch methods. Finally, BFTL1 is added in the comparison to understand the impact of (not) bounding the size of the log in batch methods. We do not integrate FlashDB [15] in the comparison because it corresponds to a dynamic mix of BTree and BFTL2. Hence its behaviour is difficult to translate accurately in a cost model and its main performance metrics would be always in-between of BTree and BFTL2. Other variations of Tree-based batch methods could also have been considered but the main objective is comparing opposite approaches (traditional, batch, Summarization & Partitioning), rather than focusing on variations.

This objective also led us to build a precise analytical cost model rather than measuring different implementations on the same hardware platform. Indeed, traditional and batch methods rely on a FTL while PBFilter is designed to bypass it. Comparing numbers obtained on top of an opaque and unpredictable FTL [15] does not make sense and would be considered unfair. By contrast, the cost model parameters can be easily set to significant numbers representing specific hardware features.

Section 5.4 reports on real experiments made with PBFilter and shows the accuracy of the cost model.

5.1.2 *Performance Metrics*

According to the problem statement introduced in Section 2.3, our performance analytical model distinguishes the following metrics:

– R: average number of page reads to look up a key.

– W: total number of page writes to insert N records.

– IR: total number of page reads to insert N records.

– RC: RAM consumption in KB.

– FO: total number of Flash pages occupied by the index structure, i.e. containing at least one valid index information.

– FE: total number of Flash pages which contain only stale data and can be erased without any data move.

Let us come back to the distinction between FO and FE, introduced in Section 2.3 to capture the garbage collector cost without making assumption on the FTL implementation. Let us denote by VD (resp. SD), the set of index data currently valid (resp. stale). If an index method never produces stale data, FO=VD and FE=0. If stale data are produced (the usual case) but are well clustered into Flash pages, FO=VD and FE=SD. If however, valid and stale data are mixed into the same Flash pages, FO can be up to VD+SD (FE being the complement).

5.1.3 *Parameters and Formulas*

The parameters and constants used in the analytical model are listed in Table 3 and Table 4, respectively.

Table 5 contains basic formulas used in the cost model and the cost model itself is presented in Table 6. To make the formulas as precise as possible, we use Yao's Formula [21] 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:*

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

Table 3. Parameters for the analytical model

Parameters	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 2 for examples)
k	Number of hash functions used by Bloom filter

Table 4. Constants for the analytical model

Constants	Signification
Sr=4 (bytes)	Size of a physical pointer
fb=0.69	Average fill factor of B+Tree [20]
β =2	Expansion factor of flash storage caused by the buffering policy in BFTL [19]
Sp=2048(bytes)	Size of a Flash page

5.2 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 [19]), 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). The first conclusion which can be drawn from these figures is that each method, except PBF, exhibits a very bad behaviour on at least one (and often more) metric. BTree exhibits a very good lookup performance but the price to pay is an extremely high write cost and consequently a high number of stale pages produced (FE). This proves that regular indexing techniques cannot accommodate Flash storage with no adaptation[7]. BFTL has been primarily designed to decrease the write cost incurred by BTree and Figures 4(c) and 4(e) shows the benefit. However, BFTL1 incurs a high lookup cost and a high RAM consumption given that the node translation lists are not bounded. The IR cost is also very bad 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 the metric R, IR and RC (though RC remains very high) at the expense of a higher number of writes (to refresh the index nodes) and a higher Flash consumption (BFTL

---

[7] We could have drawn the same conclusion if we had measured the behavior of traditional hashing in Flash. Indeed, either the number of buckets is 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 RC is very high too.

mixing valid and stale data in the same Flash pages). Hence, these figures show that PBF is the sole indexing method providing acceptable performance on all metrics at once. In this setting, PBF performance is not simply satisfactory, it outperforms BFTL1 on all metrics, BTree on metrics IR, W and FO and BFTL2 on metrics IR, W, RC and FO while obtaining as good performance as its competitors on metrics R. In addition, PBF exhibits an excellent behaviour in terms of RC, consuming only 7 buffers of 2KB (1 buffer to read from Flash, 4 buffers to write the structures SKA and another 2 for KA and DA). Note that the RAM requirement for SKA can even be reduced to a single buffer, as explained in Section 4.2, leading to a total RAM consumption of 8KB[8]. This makes PBF particularly adapted to mobile and embedded devices.

BTree being disqualified by its dramatically bad behaviour in terms of W and FO, the rest of the study concentrates on BFTL2 and PBF (BFTL1 is considered only when this allows drawing interesting conclusions).

For further analysis, we vary parameter C, keeping the preceding values for the other parameters, to determine whether BFTL2 could outperform PBF in another setting. The influence of C on metrics W, FO and FE is pictured in Figure 4(f). As expected, W and FO decrease as C increases since the tree reorganizations become less frequent (FE stays equal to 0). Actually, W and FO decrease asymptotically up to reach the same values as BFTL1 which is nothing but BFTL2 with an infinite C. Whatever C, the performance of BFTL2 on W and FO is still worse than PBF. In addition, R grows linearly with C (see formula in Table 6) so that BFTL2 needs reading 105 pages to do a lookup for C=30. The impact of increasing C is also very bad on RC (Figure 4(d) makes this clear comparing BFTL1 and BFTL2). Conversely, favourable values of C for R (e.g., R=18 for C=5) have a dramatic impact over W and FO (e.g., W=FO=241466 for C=5). As a conclusion, BFTL2 performs better than PBF on no metric whatever C.

Figures 4(g) to 4(i) analyse the scalability of BFTL and PBF methods on R, W and RC 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). 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). Unfortunately, BFTL2 seems not the ideal alternative for high N because it scales very badly in terms of W. BFTL1 scales much better in terms of W but exhibits unacceptable performance for R and RC. Let us come back to PBF. PBF scales also badly in terms of W. The reason is that 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(h) 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 impact on metric RC is negligible, especially in environments dealing with very large files (12 more buffers for SKA which could be only 3 pages when organizing the buffer by sectors). This optimization seems very effective for the scalability on W. The scalability on R requires a further optimization. Actually, the bad R performance of PBF 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 while obtaining a smaller f by increasing d to trade-off with the growth of $|KA|$. For instance, the values d=24 and k=4 produce even better lookup performance for N=5 million records than the one obtained with d=16 and k=7 for N=1 million records (9 vs. 10). Anyway, future work is required to optimize PBF further for very large files.

To make the comparison as complete as possible, Figures 4(m) and 4(n) analyze the influence of Sk on all methods, keeping the initial values of the other parameters. Regarding metric R,

---

[8] For the sake of simplicity, the formulas of the cost model consider 4 buffers for SKA.

Tree-based methods accommodate large keys better than PBF. The reason is that large keys incur more KA pages and then magnifies the influence of false positives. But again, the performance can be maintained by tuning parameters d and k so as to reduce the false positive rate in the required proportion. In Figures 4(m), we can note that the performance with some special number of Sk is even better than that with bigger Sk, simply because only 1 KA page needs to be accessed for the Bloom filter qualified by the lookup for such situation while 2 accesses for others[9]. The difference between PBF1 (d=16 and k=7) and PBF3 (d=20 and k=7) on both figures illustrates the impact of this tuning[10]. The next subsection discusses more deeply the tuning of PBF.

5.3 Tuning PBFilter

As shown in the previous section, tuning d and k allows adapting gracefully PBF to various situations. It is therefore important to delimit the range of tuning capabilities provided by PBF and understand their impact on all metrics.

The good behavior of PBF highlighted above has been obtained with relatively high values of *d* and *k* (d=16 and k=7). The size of the Bloom filters m has actually a limited impact on the size of the global index structure (in the measured setting, |SKA|/|KA| = 0.10). It mainly impacts the cost of repartitioning, which could also be tuned as suggested above. In addition, the CPU cost incurred by the computation of k hash function is negligible compared to the I/O cost (see next section). This brings high opportunities to tune *d* and *k*.

Figure 4(j) 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 FE in the proportion shown in Figure 4(k). As expected, the impact on FO is negligible. Figure 4(l) 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, FO, FE, 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.
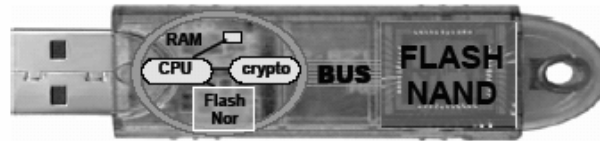
5.4 Experimental Results

Figure 3. Secure USB Flash device

---

[9] To ease the repartition process and maximise space occupancy, the Bloom filter size should be a power of 2. If by modifying d, m becomes greater than 2^x, then m is enlarged to 2^{x+1} and each Bloom filter is computed on a larger number n of keys (from two consecutive pages of KA) required to recover the expected value of d. The consequence is simply reading 2 KA pages instead of 1 for the Bloom filter(s) qualified by the lookup.

[10] Another way to deal with large Sk is hashing the keys thanks to a collision-free hash function (e.g., HMAC producing a 8 bytes signature) before inserting them in KA. This does not impact the lookup process since only exact-matches are considered.

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 an experimental secure USB Flash platform provided by Gemalto, our industrial partner. This platform, pictured in Figure 3, 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). The microcontroller itself is powered by a 32 bit RISC CPU (clocked at 50 MHz) and holds 64KB of RAM. The DBMS is built on top of a native Gemalto FTL, by-passing the address translation and garbage collection mechanisms but a similar solution is provided by the Samsung S-SIM FTL thanks to the VFL API.

Today, we are not granted permission to publish performance numbers about this platform due to pending patents, one being related to PBFilter [16]. Anyway, as already stated, real numbers are of little interest when evaluating Flash indexing methods due to the high discrepancy among the platforms. Hence the results are given in terms of number of operations, similarly to the cost model. Experiments have been conducted thanks to a cycle-accurate hardware simulator of the platform for convenience.

We ran our prototype under all the parameter settings used in 5.2 and 5.3. 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, FO and FE metrics (In the prototype, transaction atomicity is guaranteed thanks to 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 Section 4.1, assuming the k hash functions are totally independent, a condition difficult to meet in practice. Much work [7, 11] 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 7 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, as in Figure 4(a)). 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].

Table 5. Basic formulas of the analytical model

Vars	Annotations	Expressions
Common formulas		
M	Number of index units (IUs) in each page [Note1]	$\lfloor Sp/(Sk + 5*Sr) \rfloor$ for BFTL1&2 $\lfloor Sp/(Sk + Sr) \rfloor$ for others
Formulas specific to Tree-based methods (Btree, BFTL1, BFTL2)		
ht	Height of B+Tree	$\lceil \log_{fb*M+1} N \rceil$
Nn	Total number of B+Tree nodes after N insertions	$\sum_{i=1}^{ht} \lceil \frac{N}{(fb*M)(fb*M+1)^{i-1}} \rceil$
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	$\lceil \frac{L-1}{C-1} \rceil$

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	$\lfloor N/(Sp*8*M1) \rfloor *Sb + Pf + \lfloor (\lfloor N_{KA}/Mb \rfloor \bmod(L1*s))/s \rfloor *s$
N_E	Total number of pages which can be erased	$\lfloor N/(Sp*8*M1) \rfloor * \left\lfloor \sum_{i=2}^{Sb/(L1*s)}(2^{\lceil \log_2(i-1) \rceil}*L1*s) \right\rfloor + \left\lfloor \sum_{i=2}^{Nr\%(Sb/L1/s)}(2^{\lceil \log_2(i-1) \rceil}*L1*s) \right\rfloor + Nr*L1*s$

Notes:
[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 6. Final formulas of the analytical model

Metrics\\Methods	BTree	BFTL1	BFTL2	PBF
R [Note1]	ht*3	(ht-1)*L+L/2	(ht-1)*C+C/2	R1+R2+$\lceil f*N_{KA}*\lceil M/\bar{M} \rceil /2 \rceil$+ 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
RC [Note4]	B*Sp/1024	(Nn*L*Sr+B*Sp) /1024	(Nn*C* Sr+B*Sp) /1024	B*Sp/1024
FO [Note5]	Nn	W	W	$N_{KA} + N_{FB}$
FE [Note5]	W-Nn	0	0	N_E

- *Where, IR1 = Ns*(fb*M/2)* $+ \sum_{i=1}^{ht-2} i*3*((fb*M)((fb*M+1)^i-(fb*M+1)^{i-1})+1)+(ht-1)*3*(N-(fb*M)(fb*M+1)^{ht-2})$

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

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

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

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

[Note1] For BTree, the read cost of a node integrates two additional I/Os to go 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 (and 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] RC 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] FO+FE is the total number of pages occupied by both valid and stale index units. In BFTL1&2, FE=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 FE.
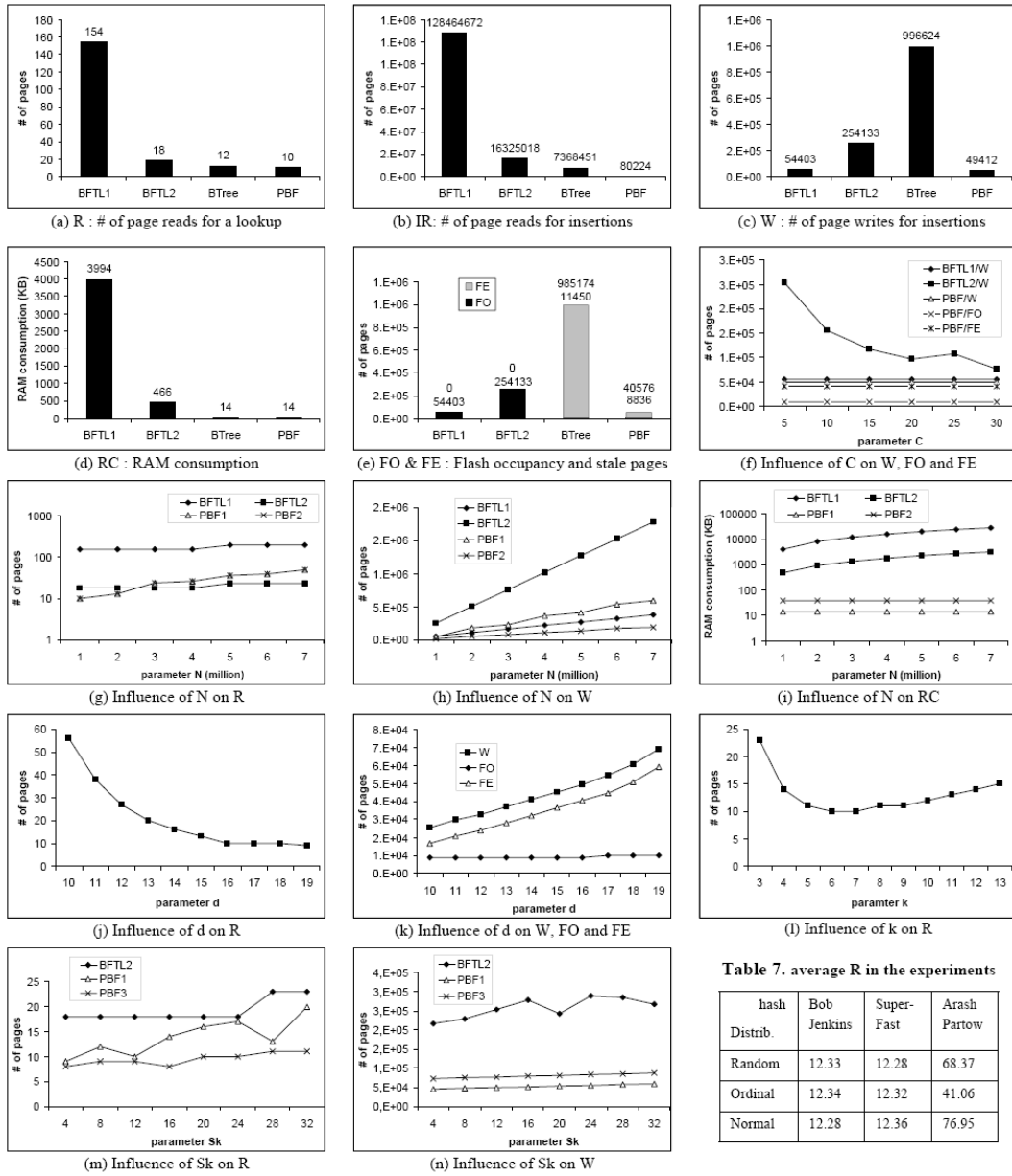
(a) R : # of page reads for a lookup

(b) IR: # of page reads for insertions

(c) W : # of page writes for insertions

(d) RC : RAM consumption

(e) FO & FE : Flash occupancy and stale pages

(f) Influence of C on W, FO and FE

(g) Influence of N on R

(h) Influence of N on W

(i) Influence of N on RC

(j) Influence of d on R

(k) Influence of d on W, FO and FE

(l) Influence of k on R

(m) Influence of Sk on R

(n) Influence of Sk on W

Table 7. 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

6 Open issues

Some questions would deserve more discussions and are part of our future work agenda.

What if deletions or updates are very frequent? The PBFilter approach has been preliminary designed to tackle applications where insertions are more frequent and critical than deletes or updates. We believe that this characteristic is common in practice. Let us assume that this assumption is wrong and then the size of DA is high. The solution would be to apply exactly the same techniques as in KA to speed-up the lookup in DA, that is Bloom filter summarization and partitioning. Assuming deletes are as frequent as inserts, all costs should be multiplied roughly by a factor 2 (read cost, write cost, Flash occupancy, RAM usage). Extrapolating the

results presented in Section 5 shows that PBFilter would remain very competitive even in this scenario and that the objective stated in Section 2, that is exhibiting a satisfactory behavior for all metrics at once, is still met.

Can we envision other/better condenser algorithms? We believe that PBFilter can be instantiated in different ways to serve optimally different purposes or tackle different target platforms. For instance, let us assume a context where the read/write cost ratio is so low that metric M2 strongly dominates M1 and where M4 can be slightly relaxed. In this case, we could use a compression algorithm which reaches a higher compression of KA at the price of a decompression in RAM before exploiting each summary record. Partitioning no longer holds in this context but the number of writes in SKA is reduced (no partition to reorganize, reduction of SKA size). An example of such algorithm

is compressed Bloom filters [14], but any other compression algorithm could be considered as far as membership tests can be performed on the decompressed summaries without false negative. Classifying the algorithms of interest according to the criteria to optimize is part of our future work.

Can we index secondary keys? Summarizing a KA list containing many duplicates does not make sense. Indeed, if the selectivity of the secondary key is lower than ($|KA|/||KA||$), every page of KA would be qualified. A first solution could be to use PBFilter to maintain the set of secondary keys present in the database (without duplicates), doing lookups in this set and attaching a list of pointers to records to each of these keys. How to manage the dynamicity of these pointer lists efficiently in Flash is an open question. Another approach would be to encode KA keys differently. Let us assume first a secondary key with a very low selectivity. The usual way to index such attribute is to build a bitmap index, that is one bitmap per domain value with one bit per record. Bitmaps can be built sequentially and replace key values in KA. The vertical partitioning principle introduced in Section 4.2 can apply to KA with the benefit to read only the requested bitmap at lookup time. If the secondary key has a higher selectivity, Value-List indexes [6] seem very appropriate to encode keys in KA. The intuition is drawing a parallel between bitmap indexing and number representation in different bases. To illustrate this, a number in the range [0..999] can be represented either by one digit in base 1000 (i.e., one bit set to 1 among 1000 bitmaps) or by three digits in base 10 (i.e., 3 bits set to 1 among 3*10 bitmaps), etc. This introduces a trade-off between space occupancy and lookup performance (e.g., in the latter case, 3 bitmaps have to be scanned). [6] also introduces Range-Encoded Bitmap Indexes to support range queries. All these techniques seem well adapted to our context since they can be managed sequentially and can be partitioned. A deeper analysis of secondary keys and range queries is part of our future work.

7 Conclusion

NAND Flash has become the most popular persistent storage medium for mobile and embedded devices and is being considered as a mid-term competitor for traditional disks, urging the database community to design Flash-aware indexing methods. Previous works in this area defer index updates thanks to a log and batch them with the objective to decrease the cost of writes in Flash. This introduces a complex trade-off between read and write performance and entails side-effects on the RAM consumption, Flash consumption and garbage collection cost. While often neglected, these side-effects may disqualify an indexing method for a target platform.
This paper introduced a comprehensive set of metrics to capture the behavior of Flash-based indexing methods, including their side-effects. Then, it proposed an alternative to batch methods, called PBFilter, exploiting natively the peculiarities of NAND Flash memory and shown its effectiveness through a comprehensive analytical performance study and preliminary performance measurements. Thanks to its good behavior on all metrics at once, including predictability (very low and bounded RAM requirement, FTL independence), PBFilter is particularly well suited to mobile and embedded devices. It has been integrated in the kernel of an embed-

ded DBMS and is being validated in this context with the help of our industrial partner. Thanks to its tuning capabilities, PBFilter seems easily adaptable to other Flash-based environments and can meet various application requirements in terms of file size, key size, read/write trade-offs. Investigating other summarization and partitioning strategies could ever enlarge the application domain of PBFilter and is part of our future work, as well as the optimal management of very large files.

8 Acknowledgments

9 Bibliography

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

[2] Birrel, A., Isard, M., Thacker, C., and Wobber, T. A Design for High-Performance Flash Disks. *Operating Systems Review* 41(2), 2007, pp. 88-93.

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

[4] Bloom, B. Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM, 13(7)*, 1970, pp. 422-426.

[5] Bolchini, C., Salice, F., Schreiber, F., and Tanca, L. Logical and Physical Design Issues for Smart Card Databases. *ACM Transactions on Information Systems (TOIS) 21(3)*, 2003, pp. 254-285.

[6] Chan, C-Y., and Ioannidis, Y-E. Bitmap Index Design and Evaluation. *Int. Conf. on Management of Data (SIGMOD),* 1998.

[7] Dillinger, P. C., and Manolios, P. Fast and Accurate Bitstate Verification for SPIN. *Proceedings of the 11th Internation Spin Workshop on Model Checking Software,* Springer-Verlag, Lecture Notes in Computer Science 2989, April 2004, pp. 57-75.

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

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

[10] Kim, G-J., Baek, S-C., Lee, H-S., Lee, H-D., and Joe, M. LGeDBMS: A Small DBMS for Embedded System with Flash Memory. *Int. Conf. on Very Large Data Bases (VLDB),* 2006, pp. 1255-1258.

[11] Kirsch, A., and Mitzenmacher, M. Less Hashing, Same Performance: Building a Better Bloom Filter. *Algorithms – ESA 2006, 14th Annual European Symposium,* Springer-Verlag, Lecture Notes in Computer Science 4168, 2006, pp. 456–467.

[12] Lee, S-W., and Moon, B. Design of Flash-Based DBMS: An In-Page Logging Approach. *Int. Conf. on Management of Data (SIGMOD)*, 2007, pp. 55-66.

[13] 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, pp. 425-426.

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

[15] Nath, S., and Kansal, A. FlashDB: Dynamic Self-tuning Database for NAND Flash. *Int. Conf. on Information Processing in Sensor Networks (IPSN)*, 2007, pp. 410-419.

[16] Pucheral, P., and Yin, S. *System and Method of Managing Indexation of Flash Memory.* Deposited by INRIA and Gemalto as a European patent n° 07290567.2-, 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, pp.26-52.

[18] Sen R., and Ramamritham K., Efficient Data Management on Lightweight Computing Device. *Int. Conf. on Data Engineering (ICDE)*, 2005, pp. 419-420.

[19] 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)*, 2003, pp. 409-430.

[20] Yao, A. On Random 2-3 Trees. *Acta Informatica*, 9 (1978), pp. 159-170.

[21] Yao, S. Approximating the Number of Accesses in Database Organizations. *Communication of the ACM 20(4),*1977, pp. 260-261.

[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, pp. 31-44.