Information Systems ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at SciVerse ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

PBFilter: A flash-based indexing scheme for embedded systems

Shaoyi Yin*, Philippe Pucheral

University of Versailles & INRIA, France

ARTICLE INFO

Article history:
Received 21 December 2011
Accepted 22 February 2012
Recommended by: G. Vossen

Keywords:
NAND Flash
Indexing
Storage
Embedded systems
Bloom filter

ABSTRACT

NAND Flash has become the most widely used electronic stable storage technology 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 (e.g., block-erase-before-page-rewrite constraint and limited number of erase cycles) and embedded system constraints (e.g., 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 properly. In this paper, we propose a different 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 analytical performance study. Extensions of PBFilter on range queries and multi-criteria queries are also discussed. The proposed technique is integrated into a full-fledged embedded DBMS engine. We describe the complete design of the DBMS engine to illustrate the feasibility of adopting PBFilter technique in a real system. Finally, we show some performance measurements of the prototype on top of a real hardware platform, in order to validate the new technique in a practical manner.

1. Introduction

Smart cards were equipped with kilobytes of EEPROM stable storage in the 90's, with megabytes of NAND Flash in the 00' and mass-storage smart objects are emerging, linking a microcontroller to gigabytes of NAND Flash memory [14]. 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

* Corresponding author. Present address: ETIS, Cergy-Pontoise University, St-Martin 2, avenue Adolphe-Chauvin 95302 CERGY-PONTOISE CEDEX, France. Tel.: +33 134 256 650, fax: +33 134 252 829.
E-mail addresses: yinshaoyi@gmail.com (S. Yin), Philippe.Pucheral@inria.fr (P. Pucheral).

0306-4379/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
doi:10.1016/j.is.2012.02.002

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 or lifetime), microcontroller size (to increase tamper-resistance or ease physical integration in various appliances) and storage capacity (to save production costs on billion-scale markets) [3]. 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 [18,21] 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 [1,5,22,25,29]. 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 [7].

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 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 [3]. For illustrative purpose, today's powerful microcontrollers include 32–128 KB of RAM. Even if this capacity is likely to increase in the future, the ratio between RAM and stable storage is likely to continuously decrease (e.g., smart objects combining a microcontroller with an SD card). Hence, the less RAM an indexing method consumes, the wider the range of devices that can be targeted.

Low_Energy is also critical for 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.[1] 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. Range query on secondary keys and multi-criteria query can be supported and optimized gracefully by following the same principles.

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 extends PBFilter technique to

[1] A preliminary design of PBFilter has been published in [33]. The current paper extends this initial design with the support of lookups on secondary keys, range queries and multi-criteria queries. It extends the analytical model accordingly and integrates new state of the art methods in the related work section and in the performance comparison. It also details how PBFilter has been integrated in a complete DBMS engine and finally gives performance numbers obtained by measuring our current prototype running on a real hardware platform.

Please cite this article as: S. Yin, P. Pucheral, PBFilter: A flash-based indexing scheme for embedded systems, Information Systems (2012), doi:10.1016/j.is.2012.02.002

support range query on secondary keys and optimize multi-criteria query. Section 6 presents a comprehensive performance study by using analytical models. Section 7 describes an embedded DBMS engine integrating PBFilter technique and shows real performance numbers produced by running the prototype. Finally, Section 8 concludes.

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), potentially divided again into sectors (usually 4 sectors per page).[2] 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. To avoid erasing blocks too frequently, "out-of-place" updates are usually performed by using a Flash Translation Layer (FTL) [16] 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).

2.2. Related work

Benchmarks [7,9] have shown that there exist large discrepancies between different commodity FTLs. However, they share some common behavior. For example, sequential writes are faster than random writes. More specifically, uFLIP [7] reveals some write efficient patterns, such as "random writes limited to a focused area" and "sequential writes limited to a few partitions" etc. Based on these observations, researchers have proposed some intermediate layers, such as Append&Pack [27], ICL [15] and StableBuffer [23], to transform inefficient write patterns into efficient ones. These FTL-friendly models have optimized the write performance and reduced the unpredictability, but they require either RAM resident mapping structures or complex pattern recognition algorithm.

Instead of adapting to the FTL common characteristics, an alternative is to bypass or abandon the FTL and design native Flash-specific storage layers. Typical work includes the In-Page Logging (IPL) [21], Page-differential logging [18], deferred updates [10], and shadow page based storage [20]. Most of them were inspired by the journaling Flash file systems like JFFS [5] and YAFFS [30], which were further inspired by the log-structured file system [26] design. They do not rely on any FTL layer, but they themselves actually are providing the functionalities of FTL, namely address translation, garbage collection and wear-leveling. Thus in terms of RAM consumption and predictability, they are similar to the FTLs.

In terms of indexing technique, traditional structures have been improved to adapt to the NAND Flash in different ways. To the best of our knowledge, most of the proposed methods are 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 [29]. 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 [29] 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 [25] 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 [5]. 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.

Lazy-Adaptive (LA) Tree [1] attaches Flash-resident buffers to B+ tree nodes at multiple level of the tree. Each buffer contains update operations to be performed on the node and its descendants. At an appropriate time, all elements in the buffer are pushed down in a batch to the buffers at the next level. Buffering introduces an inherent tension between update performance and lookup performance, so the LA-Tree adapts dynamically to arbitrary workloads using an optimal online algorithm to offer efficient support for both updates and lookups. However, to avoid fragmentation overhead of the Flash resident buffers, a big RAM buffer pool is needed for write coalescing.

[2] There are two kinds of NAND Flash techniques: SLC (Single Level Cell) and MLC (Multi-Level Cell). Sector exists only in SLC Flash, while page is the smallest unit in MLC Flash. Sectors can be written independently (albeit sequentially) in the same page, allowing one write per sector before the block must be erased. PBFilter can apply as well to SLC and MLC contexts. However, some optimizations presented in the paper rely on the use of sectors. When necessary, this distinction is mentioned in the text.

Please cite this article as: S. Yin, P. Pucheral, PBFilter: A flash-based indexing scheme for embedded systems, Information Systems (2012), doi:10.1016/j.is.2012.02.002

FD-tree [22] consists of a small B+ tree as the head tree on the top, and a few levels of sorted runs of increasing sizes at the bottom. Updates are only applied to the head tree, and then merged to the lower level sorted runs in batches. As a result, most random writes are transformed into sequential ones through the merge. However, FD-tree is targeted for off-the-shelf large Flash SSDs, and it assumes that the RAM buffer pool is large enough so that the head tree can be held in the buffer pool.

In short, all B+Tree-based methods rely on the same principle (Fig. 1): (1) to delay index updates using a log (or a Flash resident buffer) and batch them with the purpose of grouping updates related to the same index node; (2) to build a RAM index at boot time to speed up lookup of a key in the log; (3) to 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 lower 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 lower CF. Section 6 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 [7]. Flash memory usage is also difficult to predict because it depends on the distribution of obsolete data in the pages occupied by the index.

Some other work has considered the secondary key indexing problem in NAND Flash with limited RAM resource. Hash-based and tree-based index families can be distinguished.

One example of hash-based methods in Flash is Micro-hash designed to speed up lookups in sensor devices [34]. However, this method is not general and only applies to sensed data varying within a small range (e.g., temperature). Another example is FlashStore [11] which is a key-value store on NAND Flash. It organizes key-value pairs in a log-structure on Flash to exploit faster sequential write performance. It uses an in-memory hash table to index them, with hash collision resolved by a variant of cuckoo hashing, thereby consuming a significant RAM space. SkimpyStash [12] is an optimized method to reduce the consumption of RAM, but it still cannot meet the extreme embedded requirements (e.g., at least 1 MB of RAM is required to index 1 million records, while many smart objects have only tens of KB of RAM).

Within the tree-based family, one work has also considered indexing sensed data [24]. 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.

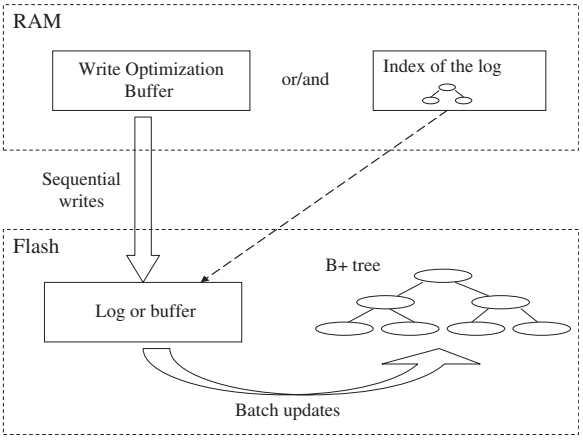

Fig. 1. Batch indexing methods.

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 [3]. 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 the read cost of executing a lookup, denoted by *R*, and the read cost and write cost for inserting keys into the index, respectively denoted by *IR* and *W*. Depending on the objective, the focus can be on 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 then be

Please cite this article as: S. Yin, P. Pucheral, PBFilter: A flash-based indexing scheme for embedded systems, Information Systems (2012), doi:10.1016/j.is.2012.02.002

reclaimed without copying any 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 Fig. 2. 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.

Intuitively, 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. 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 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.

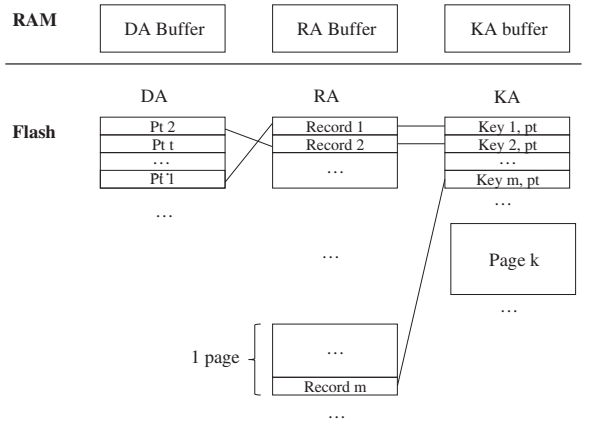

Fig. 2. Sequential database organization.

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. Thus, 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 $(|KA|-1)$.

The objective becomes decreasing 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 summarize the information present in KA into a more compact sequential structure. 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 (see Fig. 3). 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 increasing 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.

Please cite this article as: S. Yin, P. Pucheral, PBFilter: A flash-based indexing scheme for embedded systems, Information Systems (2012), doi:10.1016/j.is.2012.02.002

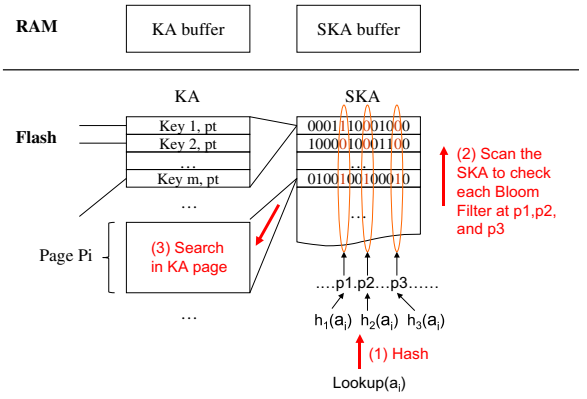

Fig. 3. Key lookup through SKA.

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 data structures can be considered as candidate "summaries", 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 [6]. Hence, it presents all the characteristics required for a summary.

A Bloom filter represents a set $A=\{a_1, a_2, \ldots a_n\}$ of n elements by a vector v of m bits, initially all set to 0. The Bloom filter uses k independent hash functions, $h_1, h_2, \ldots h_k$, 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. 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. Otherwise we conjecture that a_j is in A although there is a certain probability that we are wrong (in this case, it is a false positive). The parameters k and m can be tuned to make the probability of false positives extremely low [6].

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 $(1-1/m)^{kn}=e^{-kn/m}$. The probability of a false positive is then $(1-(1-1/m)^{kn})^k=(1-e^{-kn/m})^k=(1-p)^k$ 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 [13].

Bloom filters can be used as a summarization data structure in PBFilter as shown in Fig. 3. We consider each KA page as a dataset and build a Bloom filter for each. All these Bloom filters are built with the same hash functions, for example h_1 to h_3 in the figure. To lookup a tuple by a given key value a_i, we compute the hash functions of a_i and then scan SKA to check the corresponding bit positions of each Bloom filter. In the figure, only the i^{th} Bloom filter gives a positive answer, so we access the corresponding KA page and the probability that it contains the expected index entry (a_i, pt) is $(1-f)$. If a_i is found in this KA page, we return the result. Otherwise, it is a false positive and we continue scanning SKA. The last step is to check that pt $\notin$ DA before accessing the record in RA.

Using Bloom filter as a summary greatly improves the lookup performance but the scanning cost of SKA remains linear in function of the database size (on the average, half of SKA needs to be scanned). The partitioning principle introduced earlier may help tackling this issue by accessing only the relevant bits of each Bloom filter.

4.2. Partitioned summaries

This section details how the partitioning principle suggested in Section 3 can be applied to Bloom filter based summaries. 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

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	0.0306	0.024	0.0217	0.0216	0.0229	–
12	0.0108	0.0065	0.0046	0.0037	0.0033	0.0031
16	0.005	0.0024	0.0014	0.0009	0.0007	0.0006

Please cite this article as: S. Yin, P. Pucheral, PBFilter: A flash-based indexing scheme for embedded systems, Information Systems (2012), doi:10.1016/j.is.2012.02.002

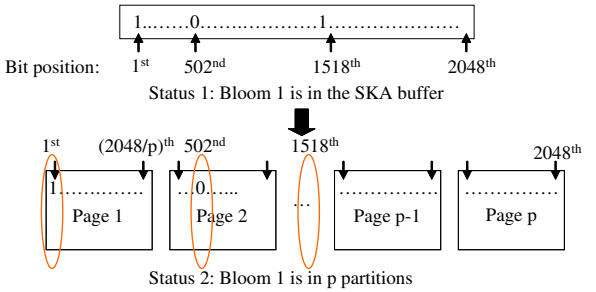

Fig. 4. Vertical partitioning of Bloom filter summaries.

$[((i-1)*m/p+1)..\ (i*m/p)]$ belong to the i^{th} 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 $h_1(a_i)$, $h_2(a_i)$, …, $h_k(a_i)$. Fig. 4 gives an example of the vertical partitioning. Suppose that we use 3 hash functions to build Bloom filters of 2048 bits and the hash values for a given key are 1, 502 and 1518, then at most 3 partitions may contain these bits and the other partitions do not need be accessed.

The benefit is a cost reduction of the lookup at least by a factor p/k, where k is the number of hash functions used for computing the Bloom filters. The larger p, the higher the partitioning benefit for lookups. However, the larger p, the greater the RAM consumption (if p more buffers are allocated) or the greater the number of Flash 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). The next step is then to find a way to build a large number of partitions without violating the Low_RAM and Low_Storage requirements.

4.3. Dynamic partitioning

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.

Figs. 5–7 illustrate 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.

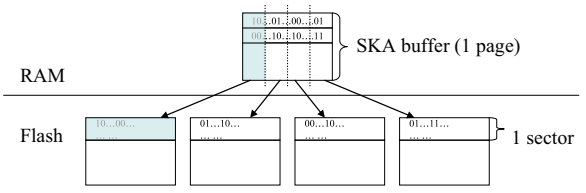

Fig. 5. Initial partitioning of SKA.

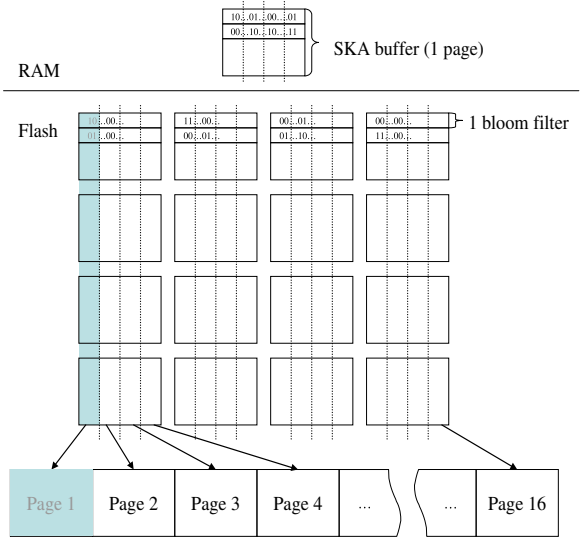

Fig. 6. Repartitioning of SKA.

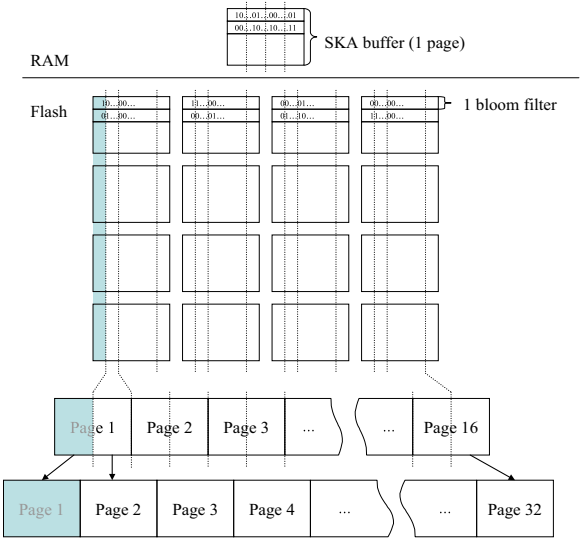

Fig. 7. Second repartitioning of SKA.

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 (Fig. 5), each sector si (with $1 \le i \le s$) 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

Please cite this article as: S. Yin, P. Pucheral, PBFilter: A flash-based indexing scheme for embedded systems, Information Systems (2012), doi:10.1016/j.is.2012.02.002

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 s2 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 $[((i-1)*m/s+1)\ldots\ (i*m/s)]$).

At this time (Fig. 6), the s L1 partitions of s pages each are reorganized (read back and rewritten) to form s2 L2 partitions of one page each. Then, each L2 partition contains 1/s2 part of all Bloom filters. As illustrated in Fig. 7, 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 $[((i-1)*m/s2+1)\ldots\ (i*m/s2)]$).

After another s2 SKA buffer flushes (Fig. 7), s new L1 partitions have been built again and are reorganized with the s2 L2 partitions to form (s2+s2) 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 Figs. 5–7, where $s=4$, the number of partitions grows in an approximately linear way (4, 16, 32…).[3] Assuming for illustration purpose Flash pages of 2 KB, 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 at most 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 $p \leq m$.

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 Bp/d records, where Bp is the size of a Flash page in bits (Bp/d is likely to be greater than 1000). Section 6 will show that this extra cost is actually low compared to existing indexing techniques. Section 6 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.

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 Li-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 for the L1 partitions and of the Li 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.

4.4. 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.

4.5. An illustration of hashed PBFilter

Now let us illustrate the key insertion and lookup processes of hashed PBFilter through an example (Fig. 8). As pointed above, we set $q=s=4$ and $m=2048$, while supposing the size of ⟨key, pt⟩ is 16 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

[3] 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.

Please cite this article as: S. Yin, P. Pucheral, PBFilter: A flash-based indexing scheme for embedded systems, Information Systems (2012), doi:10.1016/j.is.2012.02.002

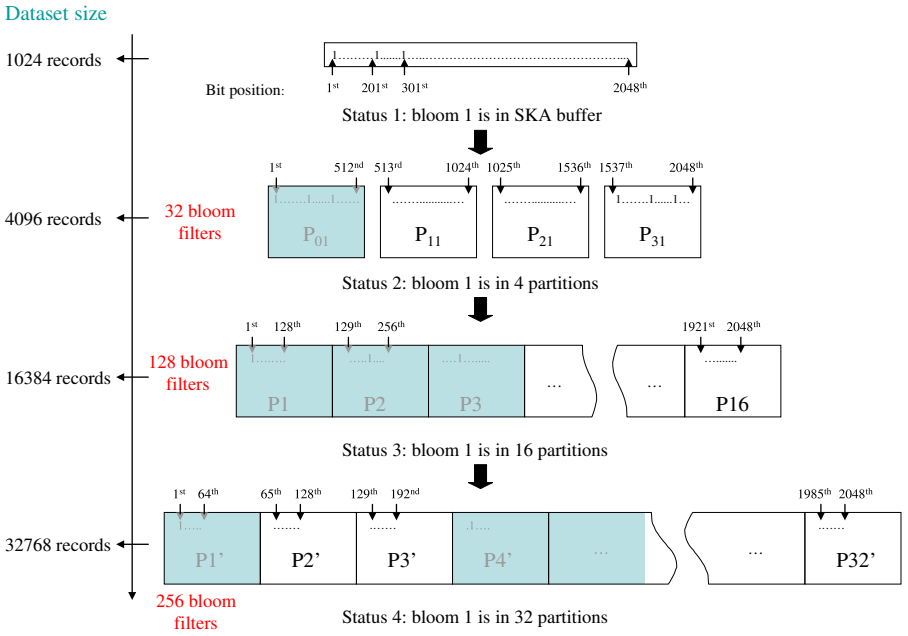

Fig. 8. Illustration of hashed PBFilter.

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 Fig. 8).

After inserting 1024 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 4096 keys, 32 bloom filters have been built and are split into 4 vertical partitions. If we lookup the key key1, we need 1 I/O to check the three bits 1, 201 and 301.

After inserting 16384 keys, the L1 partition pages are full, so the bloom filters are repartitioned into smaller pieces forming the L2 partitions: the first 128 bits of all the bloom filters are written into the first L2 partition P1, the second 128 bits of all the bloom filters are written into the second L2 partition P2, and so forth (Status 3). Now we have 128 bloom filters split into 16 vertical partitions. For the same query, we need 3 I/Os to check the bit positions, because each bit position is located in a separate page.

After inserting 32768 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 the 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 the bloom filters are written into the second L3 partition P2', and so forth (Status 4). At this time, we have 256 Bloom filters split into 32 vertical partitions. 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. 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 there are still only three pages to be checked. This is why we do the vertical partitioning; the objective is to keep the lookup cost as a constant regardless of the database size. In this example, key1 will be found by only checking chosen pages in the L3 partitions. 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 P11, P12, P13 and P14 are scanned if they are not empty.

4.6. 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 in 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 randomly deleting/updating elements. From a legal point of view, the former is allowed and even compulsory, while the latter is often forbidden. From a technical point

Please cite this article as: S. Yin, P. Pucheral, PBFilter: A flash-based indexing scheme for embedded systems, Information Systems (2012), doi:10.1016/j.is.2012.02.002

of view, 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 water mark 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, which 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 6 will show, the lookup cost is kept low, though roughly multiplied by a factor 2 with high update/delete rate.

5. PBFilter extension

5.1. Point and range queries on secondary keys

Using Bloom Filter summaries also makes sense for supporting point queries on secondary keys when the selectivity is relatively high. The lookup process is similar to primary key lookup except that the search does not stop after the first match. However, if the selectivity of the secondary key is worse than ($|KA|/\|KA\|$), every page of KA will be qualified and scanning SKA does no longer provide any advantage. Note that in this case there are many duplicate keys in KA, so there must be a way to make KA more compact in order to speed up the lookup. Bitmap index can be used for this purpose. The design principle of smart bitmap index has been addressed by [8]. It draws a parallel between bitmap indexing and number representation by introducing the attribute value decomposition mechanism, e.g. a number in the range [0.999] can be represented either by one component in base 1000 (i.e., each record is represented by 1000 bits and only 1 bit is set) or by three components in base 10 (i.e., each record is represented by 3×10 bits and 3 bits are set), etc. The partitioning principle introduced in Section 3 can be used for any representation form. Each bitmap is then stored in one separate partition such that only a few partitions need to be checked for a given query. This introduces a trade-off between space occupancy and lookup performance.

In [8], two different encoding schemes are introduced to support different types of select query. They are Equality Encoding and Range Encoding. Range query can be supported efficiently using Range Encoding scheme (only 1–2 bitmap are required from each component to retrieve the qualified tuple identifiers).

Let us consider Range Encoding to index an attribute with 256 distinct values of a table containing 1 million tuples (assuming that the tuple size is 100 bytes, the whole table takes about 50,000 pages of 2KB size). The most direct way is to use a single component, base-256, Range-Encoded bitmap index (Fig. 9(1)), which encodes each value into 255 bits (the highest bit can be omitted as it is always set to 1). In this case about 16000 Flash pages are consumed to store the bitmaps, while on average only 1.3 bitmaps (see formulas given by [8]) need to be scanned (i.e. 83 pages) to retrieve the tuple identifiers qualifying a select predicate. Actually, this is the time-optimal encoding scheme. We can also use space optimal and space-time trade-off optimal schemes. The space optimal scheme (Fig. 9(2)) for this example is to encode the keys into 8 components of base 2 with only 1 bitmap stored in each component. It takes only 512 Flash pages, but on average 7.7 bitmaps (493 pages) have to be scanned to retrieve the qualified tuple identifiers. The time-space optimal scheme (Fig. 9(3)) decomposes the keys into 2 components of base 16 with 15 bitmaps stored in each component. It takes 1920 Flash pages, and about 3.1 bitmaps (198 pages) are required by a select predicate.

In our context, besides the trade-off between lookup performance and Flash space occupancy, the RAM consumption should be also taken into account. However, the size of RAM used as write buffer is actually proportional to the number of bitmaps, because each bitmap is considered as a separate partition which requires a separate write buffer. Thus, the estimation of RAM consumption is straightforward. To reduce the RAM consumption, the same dynamic partitioning technique as before can be adopted.

5.2. Optimization for multi-criteria queries

To evaluate multi-criteria queries, a typical way is evaluating each predicate separately, and join the intermediate results to get the final result. However, the size of the intermediate results is usually very big, thus requires consuming a lot of RAM or materializing them into Flash. An alternative is to evaluate all the predicates in a pipeline way. This is supported natively in our design (meaning that the query can be evaluated without repeated random I/O access) thanks to its sequential organization. For different types of predicates in the query, we have different strategies. They are explained below with some examples based on the following database schema:

Prescription (Doctor_Name, Patient_Name, Drug_Name, Drug_Quantity, Date);

Patient (Name, Sex, Age, City, Height, Weight);

…

1) *All the predicates are equi-select, for example "find all the prescriptions made to Jim Smith by Doctor Lucy Green".*

If the query is predefined, one single SKA can be built for several KAs. For the given example, a SKA based on the concatenation of (Patient_Name, Doctor_Name) is built, summarizing two KA structures on the Prescription table, with ⟨Patient_Name, pointer⟩ and ⟨Doctor_Name, pointer⟩ respectively (Fig. 10). To evaluate the above query, the SKA is checked using (Jim Smith, Lucy Green) as the key. If there is a match, the corresponding two KA pages are accessed to verify whether it is a real match or a false positive.

If the query is ad-hoc, the SKA of the highest selectivity attribute is checked first, and the SKA pages of the other attributes are only checked when there is a match in the

Please cite this article as: S. Yin, P. Pucheral, PBFilter: A flash-based indexing scheme for embedded systems, Information Systems (2012), doi:10.1016/j.is.2012.02.002

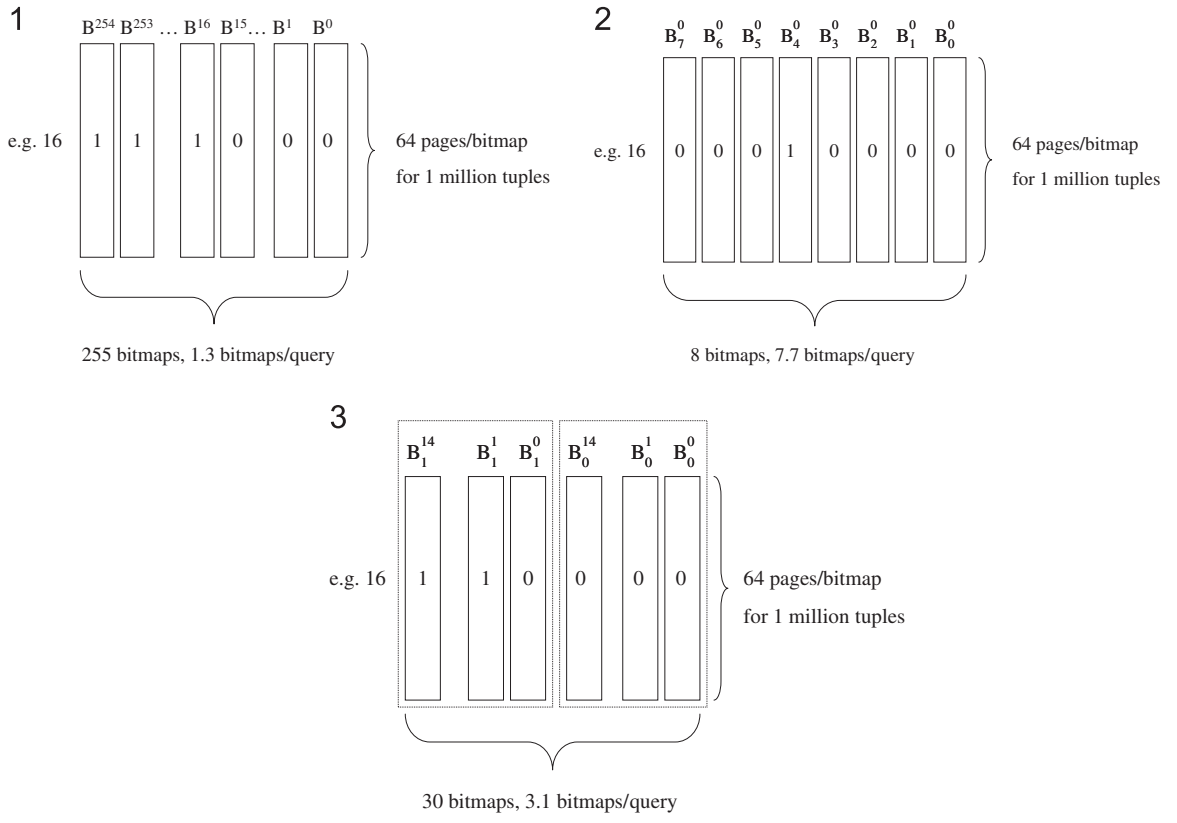

Fig. 9. Partitioned ioannidis bitmap indexing. 1) Time-optimal scheme; 2) space-optimal scheme; 3) time-space optimal scheme.

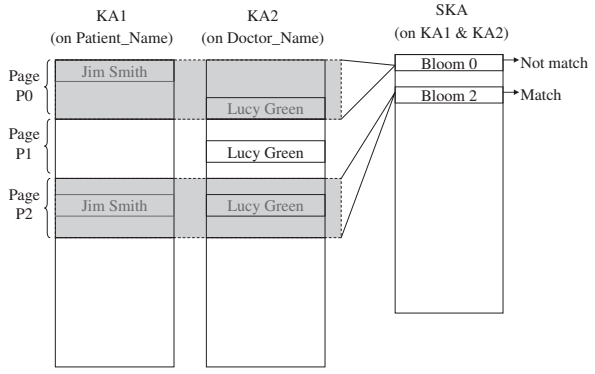

Fig. 10. Concatenated SKA.

first SKA, so that the results are produced in a pipeline way without loading and checking all the SKA pages. This requires however that an homothety can be defined between the two SKAs.[4]

2) *All the predicates are range-select, for example "find all the patients between 15 and 18 year old whose weight is higher than* 80 *kg".*

For this kind of query, the bitmap encoded KAs are scanned in parallel and bitwise operations are used to merge the bitmaps. For the above example, assuming that there are two bitmap KAs on Age and Weight respectively, we load one page of each required bit of both KAs, evaluate the two predicates, and then add an AND operation on the two result bitmaps, so that the final pointer lists can be produced directly.

3) *Equi-select and range-select mixed predicates, for example "find all the patients who live in Paris and older than* 60 *years".*

We scan the SKA on the attribute in the equi-select predicate, and check the bitmap encoded KA page on the attribute in the range-select predicate only when there is a match in the SKA. For the given example, assuming that a SKA is built on KA ⟨City, pointer⟩ and there is a bitmap KA on Age, we scan the SKA on City using "Paris" (Fig. 11). When there is a match, we load the required bits (corresponding to the matched bloom filter) of the KA on Age and evaluate the predicate. If there are matches, we go the KA page of City to check the false positive. If not, we output the result and continue scanning the SKA on City.

6. Analytical performance evaluation

The first objective of this section is to study how traditional B+Tree, batch methods, hash-based methods and PBFilter perform in the embedded context. To allow a fair comparison between the approaches and isolate the

[4] The homothety is trivial to define if the keys present in the summarized KAs have a fixed size. Note that variable size keys can be advantageously replaced by a fixed size hash of the same values.

Please cite this article as: S. Yin, P. Pucheral, PBFilter: A flash-based indexing scheme for embedded systems, Information Systems (2012), doi:10.1016/j.is.2012.02.002

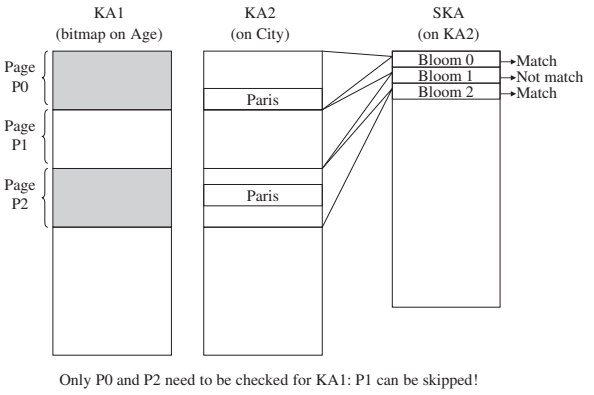

Fig. 11. Equi-selects and range-selects mixed predicates.

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, batch methods and hash-based methods can slightly outperform PBFilter in specific 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.

6.1. Performance comparison

6.1.1. Indexing methods under test

As stated above, the objective is not to perform an exhaustive comparison of all Flash-based indexing methods proposed in the literature, considering that only PBFilter has been specifically designed to cope with embedded constraints. The comparison will then consider one representative of each approach (traditional, batch, hash, Summarization & Partitioning), rather than focusing on slight 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 [29] is selected as a good, and probably one of the 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 (i.e., infinite log) and BFTL2 with the periodic compaction of the node translation table suggested in [29]. SkimpyStash [12], denoted by SStash hereafter, represents the hash-based methods. 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).

6.1.2. Parameters and formulas

The parameters and constants used in the analytical model are listed in Tables 2 and 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 [32] when necessary.

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

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

6.1.3. Performance comparison

We first compare the five 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 [29]), k_ss=8 for SStash (a suggested value by [12]) and $B=7$, $d=16$, $k=7$ for PBF (which correspond also to medium values). The results are shown in Fig. 12(a–e).

These five figures lead to the following conclusions. 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). SStash is almost perfect in terms of performance and Flash usage, but it unfortunately consumes much more RAM than what can accommodate most embedded environments. As a reminder, today's powerful microcontrollers include 32–128 KB of RAM (to be shared by all running software components, including the operating system) and the ratio between RAM and stable storage is

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.
k_ss	Average number of records in each bucket of SStash.

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 [31].
$\beta=2$	Expansion factor of flash storage caused by the buffering policy in BFTL [29].
Sp=2048(bytes)	Size of a Flash page.

Please cite this article as: S. Yin, P. Pucheral, PBFilter: A flash-based indexing scheme for embedded systems, Information Systems (2012), doi:10.1016/j.is.2012.02.002

Table 4
Basic formulas of the analytical model.

Vars	Annotations	Expressions
Common formulas		
M	Number of index units (IUs) in each page [Note1]	$\lfloor Sp/+5*Sr \rfloor$ 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	$\sum_{i=1}^{ht} \left\lceil N/((fb*M)((fb*M+1)^{i-1}) \right\rceil$
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	$\lfloor (L-1)/(C-1) \rfloor$
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	$\lfloor N/(Sp*8*M1) \rfloor *Sb + Pf + \lfloor (\lfloor N_{KA}/Mb \rfloor \bmod (L1*s))/s \rfloor *s$
N_E	Total number of pages which can be erased	$\lfloor N/(Sp*8*M1) \rfloor * \left\lfloor \sum_{i=2}^{Sb/(L1*s)} (2^{\lceil \log_2(i-1) \rceil} * L1*s) \right\rfloor + \left\lfloor \sum_{i=2}^{Nr*(Sb/L1/s)} (2^{\lceil \log_2(i-1) \rceil} * L1*s) \right\rfloor + Nr*L1*s$

[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 partition and L1*s is the size of the Flash buffer used to manage them.

expected to continue decreasing in the foreseeable future (see Section 1). No more than a few KB of RAM can then be allocated to each index.

BFTL has been primarily designed to decrease the write cost incurred by BTree and Fig. 12(c and 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 behavior 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 Fig. 12(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 [25] 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 (Fig. 12(a) shows that metric R for PBF ranges from 10 without update up to 22 with 100% updates),[5] PBF is proved to be the sole indexing method to meet all embedded constraints at once. In this setting, PBF exhibits excellent behavior 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 Sections 3 and 4.2, leading to a total RAM consumption of 3.5 KB.[6]

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. Figs. 12(g) to (i) analyze 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

[5] 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).

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

Please cite this article as: S. Yin, P. Pucheral, PBFilter: A flash-based indexing scheme for embedded systems, Information Systems (2012), doi:10.1016/j.is.2012.02.002

Table 5
Final formulas of the analytical model.

Metrics	Methods				
	BTree	BFTL1	BFTL2	SStash	PBF
R [Note1]	ht	(ht-1)∗L+L/2	(ht-1)∗C+C/2	k_ss/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	$\lceil N/M\rceil$	$N_{KA}+N_{FB}+N_E$
IR [Note3]	IR1	IR2	IR3	0	N_E
RAM [Note4]	B∗Sp/1024	(Nn∗L∗Sr+B∗Sp)/1024	(Nn∗C∗Sr+B∗Sp)/1024	(N+5∗N/k_ss)/1024+2∗2	B∗Sp/1024
VP [Note5]	Nn	W	W	W	$N_{KA}+N_{FB}$
OP [Note5]	W-Nn	0	0	0	N_E

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

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

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

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

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

[Note1]: For BTree, 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.

parameters (Figs. 12(g) and (i) use a logarithmic scale for readability). BTree is not further considered considering its dramatically bad behavior in terms of W and OP. SStash is not taken into account either, because the RAM consumption is linear to N, which is not comparable with the other methods.

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 Fig. 12(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).

6.1.4. About frequent deletions and range queries on secondary keys

As shown is Fig. 12(a), large number of random deletions or updates degrades the lookup performance of PBFilter because of the search in DA. Fig. 12(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.

Considering range queries on secondary keys, Fig. 12(k) shows the lookup cost for the example given in Section 5.1 (i.e. range query on an attribute with 256 distinct values of a table containing 1 million tuples) using (1) table scan, (2) space-time trade-off optimal bitmap and (3) ideal scan which could be produced by an oracle guessing the qualified RA pages and accessing them with no extra cost. Unsurprisingly, all RA pages have to be accessed when the selectivity is worse than 5% in this example, making any index useless. For higher selectivity, the bitmap index is beneficial and produce a cost very close to ideal scan, the bitmap size being very low compared to RA.

Please cite this article as: S. Yin, P. Pucheral, PBFilter: A flash-based indexing scheme for embedded systems, Information Systems (2012), doi:10.1016/j.is.2012.02.002

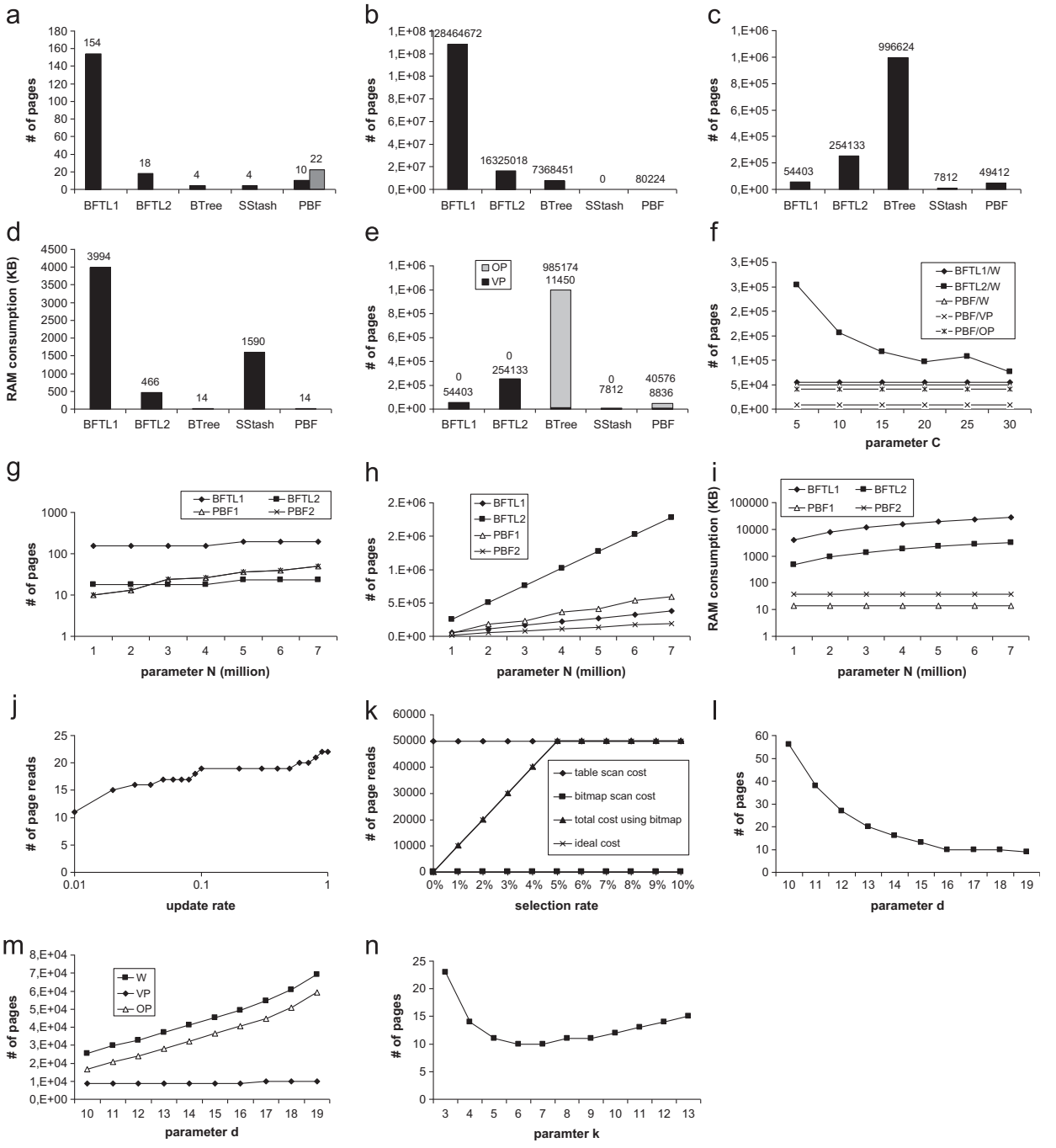

Fig. 12. Evaluation results. (a) R: # of page reads for a lookup, (b) IR: # of page reads for insertions, (c) W: # of page writes for insertions, (d) RAM consumption, (e) VP & OP: valid and obsolete pages, (f) influence of C on W, (g) influence of N on R, (h) influence of N on W, (i) influence of N on RAM, (j) influence of update rate on R, (k) range queries on secondary keys, (l) influence of d on R, (m) influence of d on W, VP and OP and (n) influence of k on R.

6.2. 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.

Fig. 12(l) shows the influence of d on R with the other parameters set to the previous values: $N=1$ million, $Sk=12$, $B=7$, $k=7$. As expected, the bigger d, the smaller the false positive rate and then the better R. At the same time, larger

Please cite this article as: S. Yin, P. Pucheral, PBFilter: A flash-based indexing scheme for embedded systems, Information Systems (2012), doi:10.1016/j.is.2012.02.002

Bloom filters increase the frequency of repartitioning and then increase W and OP in the proportion shown in Fig. 12(m). The impact on VP is however very limited because of the small size of SKA compared to KA (e.g., for $d=16$ and $k=7$, $|SKA|/|KA|=0.1$). Fig. 12(n) 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 6.1.3 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.

7. Experimental results

The above section have shown analytically that the PBFilter indexing scheme performs very well under the embedded constraints. The objective of this section is twofold: (1) to assess whether PBFilter can be integrated in a complete DBMS engine without hurting other major components like buffer management, transaction atomicity and concurrency control and (2) corroborate the analytical results with performance measurements. We have actually developed a full fledged DBMS engine which adopts the PBFilter technique. In this section, we will first describe the hardware platform on which this prototype is running, then explain the design of the embedded DBMS engine and its use cases, and finally show and interpret real performance numbers obtained on the described platform.

7.1. Hardware platform

Our prototype runs on a secure USB Flash device provided by Gemalto, our industrial partner. This platform is equipped with a smartcard-like secure microcontroller (MCU) connected by a bus to a large (Gigabyte-sized soon) NAND Flash memory (today the 128 MB Samsung K9F1G08 × 0A module), as shown in Fig. 13.

The MCU itself is powered by a 32 bit RISC CPU (today clocked at 50 MHz and soon at 120 MHz) associated to a crypto-coprocessor implementing the usual symmetric and asymmetric cryptographic operations (typically, AES and RSA). It holds 64 KB of RAM (half of it pre-empted by the operating system) and 1 MB of NOR Flash memory (hosting the on-board applications' code and used as write persistent buffers for the external NAND Flash). Security modules guarantee the tamper resistance of the MCU.

The current platform has a SIM card form factor and can be plugged into a cell phone or a USB token providing the connectivity with any terminal equipped with an USB port. Whatever their form factor, smart objects share strong hardware commonalities regarding the management of persistent data. They mostly rely on a MCU connected to a NAND Flash store. To this respect, Gemalto's platform is well representative of the family of smart objects dedicated to data driven applications.

7.2. Software platform

The PBFilter scheme is integrated into the PlugDB engine. PlugDB is a relational DBMS engine designed to be embedded in various forms of smart objects. It implements the main features of a DBMS, namely query processing, access control, integrity control and transaction management. PlugDB is used today in a real-life application implementing a secure and portable medico-social folder in the Yvelines district in France [4]. Its goal is to improve the coordination of medical and social cares while giving the control back to the patient over how her data is accessed and shared.[7] In this experiment, PlugDB is embedded in the Gemalto's platform described above. More generally, PlugDB aims at becoming the master piece of the *Personal Data Server* vision [2], developed as an alternative to the systematic centralization of personal data. In this vision, secure tokens are used to securely manage digitized personal data (salary forms, invoices, banking statements, etc), government required electronic records and even sensitive technical, commercial or scientific data.

In all these contexts, the amount of data is potentially huge and requires powerful indexes. Due to the extremely low ratio between the RAM and the capacity of the NAND Flash, PlugDB relies on a massive indexing scheme such that all combinations of selections and joins can be computed in pipeline. In this scheme, all the tables, selection indices, join indices and deleted tuple lists are organized as sequential data structures, following the PBFilter principle.

The acute readers may notice that the write buffer of the sequential data structures are kept in RAM in the PBFilter scheme, so the committed data may be lost in case of failure or disconnection (some secure tokens take their energy from the terminal they are plugged in). This issue is orthogonal to the indexing scheme but it must be tackled in a real prototype where transaction atomicity is

[7] The consortium conducting this field experiment includes the Yvelines district council, INRIA, University of Versailles, Gemalto (the smart cards world leader) and Santeos (an Atos subsidiary managing the French National Electronic Healthcare Record System).

Please cite this article as: S. Yin, P. Pucheral, PBFilter: A flash-based indexing scheme for embedded systems, Information Systems (2012), doi:10.1016/j.is.2012.02.002

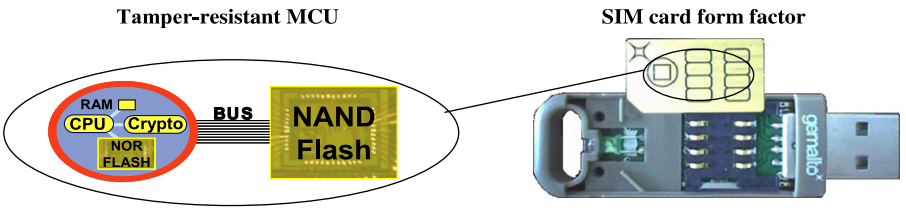

Fig. 13. Functional and physical views of the Gemalto's Secure USB Flash device.

mandatory. To this end, RAM buffers are flushed in NOR, a non-volatile memory with a byte r/w granularity. At commit time, if the committed tuples in NOR represent more than one page of NAND, full pages of NAND are built and flushed in NAND while the rest of the data remains in NOR. For each sequential data structure, we maintain a high water mark HWM which is the logical position of the last committed tuple. If a transaction is rolled back, we simply erase all the tuples/elements inserted after the HWM in NOR.

In the prototype, we have not yet implemented the concurrency control mechanism, meaning that we support one single active transaction at a time. However, the sequential design of PBFilter offers interesting opportunities to simplify the concurrency control. Since data are never overwritten in NAND Flash, multi-version concurrency control policy can be adopted. Thanks to the append-only principle, different versions of the database can be distinguished simply using the different versions of HWMs. If a transaction Ti starting at time ti accesses a data structure, it simply refers to the HWM of this structure created at the last time tj preceding ti, thereby ignoring any new tuples created (committed or not) after time tj. The benefit of using this multi-version control is twofold: (1) no additional log information needs to be maintained, because the old versions of data are naturally reserved in the database; (2) no locks are required for reading transactions.

7.3. Performance measurements

The objective of our performance measurements is to complement the lessons learned from the analytical study in three ways: (1) to measure the actual false positive rate of Bloom filters and compare it with its theoretical evaluation, (2) to measure the actual cost of Flash IOs and capture the discrepancy between reads, sequential writes and random writes on the hardware platform under test and (3) to measure the ratio between the IO cost and the CPU cost and assess whether the latter is negligible or not.

We address the first point by running our code on a simulator so that big data volume (1 million records) can be tested. In order to address the second and third points, we measured the exact execution time of tests running on the real hardware platform with a smaller data volume (100000 records), due to the limited capacity (128 MB) of the NAND Flash embedded in our current experimental platform.

Table 6
Average R in the experiments.

Distribution	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

7.3.1. The real false positive rate

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. Some work [13,19] has introduced how to build efficient and accurate Bloom filter hash functions. In our experiment, we compared Bob Jenkin's 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). In our prototype, we selected the Bob Jenkins implementation of hash functions.

7.3.2. The real cost of flash IOs

The Gemalto's platform used for our experiments provides 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). Our prototype has been developed on top of the VFL layer.

Please cite this article as: S. Yin, P. Pucheral, PBFilter: A flash-based indexing scheme for embedded systems, Information Systems (2012), doi:10.1016/j.is.2012.02.002

7.3.3. Global query cost and relative CPU cost

A basic version of PBFilter (i.e. integrating KA and SKA without partitioning) has been implemented in the PlugDB prototype. To have an idea about the ratio between CPU and IO cost, we let the DBMS engine execute the following query, SELECT * from INFO where IdGlobal = x, where INFO is the biggest table in the medical database used in our field experiment and IdGlobal is its primary key. We built a PBFilter index on the IdGlobal attribute, and the selection is performed by using this index.

In the first experiment, we populated the database with randomly generated keys and executed the above query by setting the IdGlobal to the id of the tuple in the middle of the table after every 4800 insertions. The query execution time is shown in Fig. 14, and the IO/CPU cost is also drawn in the figure. An interesting phenomenon was observed; the CPU cost in the second case (9600 tuples) and fourth case (19200 tuples) is significantly higher than in the first (4800 tuples) and third (14400 tuples) cases respectively. The 4800–9600 discrepancy is due to the appearance of a false positive answer, leading to load and check one extra page of KA. The 14400–19200 discrepancy is due to the fact that the RAM buffer is almost full at query time so that more KA keys have to be checked without the help of SKA. Both observations lead to the conclusion that the key comparison is a CPU consuming operation which cannot be neglected. Despite this, the CPU cost grows linearly with the dataset size, because the bigger the dataset is, the more Bloom filters need to be checked when scanning SKA. However, the IO cost becomes more and more dominant as the dataset size increases.

We then run 20 times the same query by populating the database with different datasets, from 10000 up to 100000 tuples. The average CPU and IO cost in the global execution plan (labeled as *global*) and in the index lookup (labeled as *index*) are plotted in Fig. 15. In this example, the query cost is dominated by the index lookup cost, because the projection operation only retrieves one single tuple that has been selected. Again, we can observe that the IO cost grows linearly with the dataset size, which is not surprising. The result confirms that the CPU cost also grows linearly with the dataset size, due to the increasing number of Bloom filters to be examined and to the increasing number of false positives. The predominance of the IO cost for large datasets confirm the interest of I/O based optimizations like partitioning and pre-hashing in PBFilter.

Table 7
I/O Performance through different API levels.

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

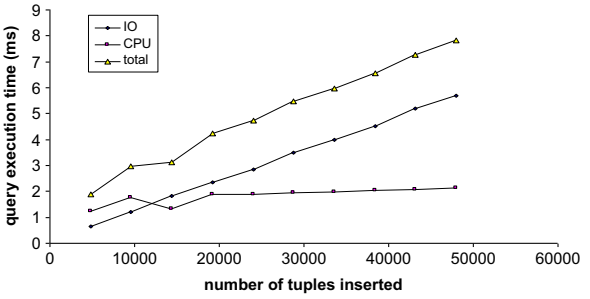

Fig. 14. Execution time of the query (for a specific dataset).

7.3.4. The performance gain provided by PBFilter

In order to show the performance gain of using Bloom filter summaries, as well as the benefit of pre-hashing and partitioning the summaries, we implemented three stand-alone indexing structures: 1) scanning KA (without SKA, without pre-hashing); 2) scanning SKA (with pre-hashing but without partitioning); 3) PBFilter (with pre-hashing and partitioning the SKA). They are not integrated into the PlugDB engine, but have been tested in isolation under the same Gemalto's platform described above. The parameter setting of the test is the same as in Section 6,

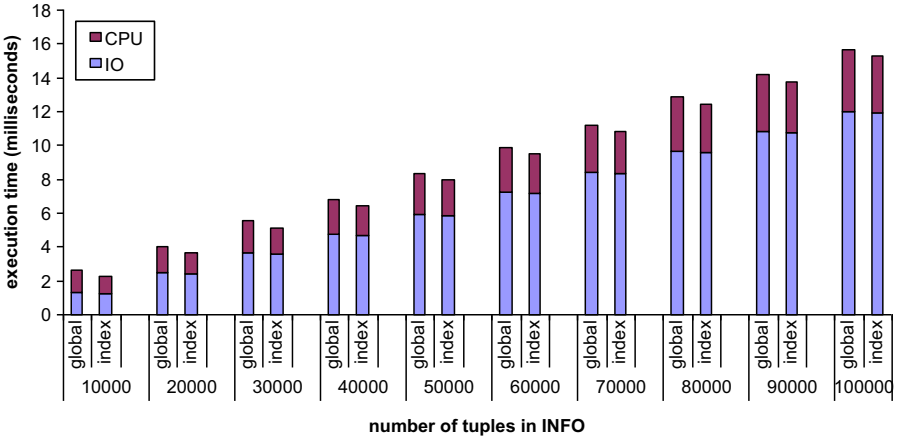

Fig. 15. Execution time of the query (average result).

Please cite this article as: S. Yin, P. Pucheral, PBFilter: A flash-based indexing scheme for embedded systems, Information Systems (2012), doi:10.1016/j.is.2012.02.002

i.e. $Sk=12$, $d=16$ and $k=7$. Database size varies from 10000 to 100000. The test is to lookup the key in the middle of the database which represents the average case. The performance of Scanning KA and Scanning SKA degrades linearly with the size of the database, while the lookup time of PBFilter keeps almost constant, as shown in Fig. 16 (we use a logarithmic scale for the lookup time). For example, when the number of inserted tuples grows from 10000 to 100000, the lookup performance of Scanning SKA becomes about 6.4 times worse, while PBFilter only 1.5 times worse. For big datasets, the performance gain of PBFilter is remarkable. For example, for 100000 tuples, PBFilter performs 4 times better than Scanning SKA, and more than 100 times better than Scanning KA.

Compared to the experiments in Section 7.3.3, the IO cost is reduced a lot due to the partitioning and pre-hashing, so the CPU cost becomes relatively more important (almost half of the total cost). This does not mean however that PBFilter could become CPU bounded but simply that IOs have reached their bare minimum. In addition, when the CPU frequency rises to 120 MHz (corresponding to the next Gemalto platform), the IO cost will dominate again. Fig. 17 illustrates this trend, where CPU1 is clocked at 50 MHz and CPU2 is clocked at 120 MHz. The numbers for 120 MHz CPU are extrapolated thanks to the performance discrepancies measured by Gemalto on both platforms for the basic CPU consuming operations such as key comparison, hash function computation and Bloom filter bit tests. Note that the hardware trend is a regular increase of the CPU frequency

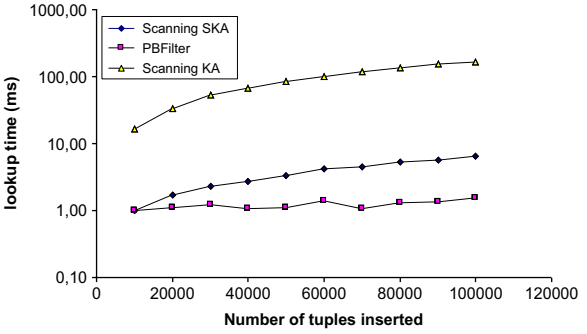

Fig. 16. Execution time of key lookup.

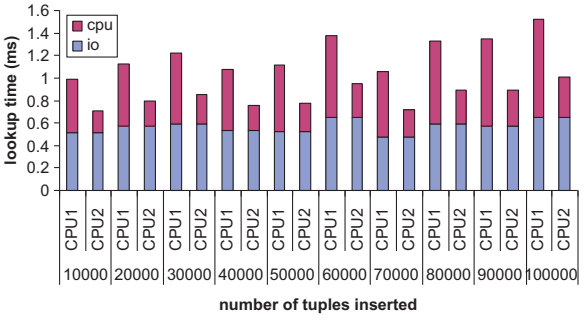

Fig. 17. CPU/IO ratio for PBFilter.

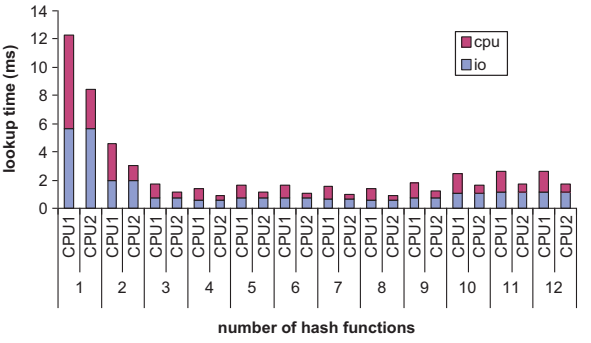

Fig. 18. Influence of k.

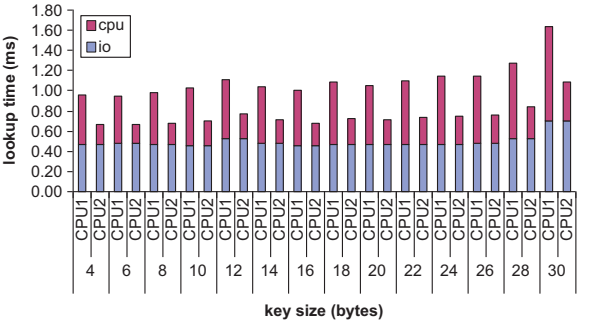

Fig. 19. Influence of key size.

while the cost of NAND Flash IOs is expected to remain rather stable.

7.3.5. *Influence of number of hash functions and key size*

Changing the number of hash functions does not only influence the IO cost, but also the CPU cost. In the real experiment, we measured both with the CPU clocked at 50 MHz and extended the result to the case where the CPU is clocked at 120 MHz. We set $N=100000$, $Sk=12$ and $d=16$. The number of hash functions k varies from 1 to 12. The result coincides with the conclusion given by the cost model, the bigger k, the better R, up to a given threshold, as shown in Fig. 18.

To measure the stability of PBFilter, we varied the key size between 4 bytes and 30 bytes with the following setting: $N=40000$ (due to the limited Flash space), $d=16$ and $k=7$. Both the CPU cost and IO cost increases when the key size grows, because there are less and less keys in each KA pages thus more and more bloom filters are built, but the discrepancy is not large, as we can see in Fig. 19.

8. 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 conflicting NAND Flash and embedded system constraints. To the best of our knowledge, PBFilter is the first indexing method addressing specifically this combination of constraints.

Please cite this article as: S. Yin, P. Pucheral, PBFilter: A flash-based indexing scheme for embedded systems, Information Systems (2012), doi:10.1016/j.is.2012.02.002

The paper introduces a comprehensive set of metrics to capture the requirements of the targeted context. Then, it shows that batch methods and hash-based 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 designed and implemented and a comprehensive performance study shows its effectiveness. The PBFilter has preliminary be designed to support exact match queries on primary keys but it supports as well lookups on secondary keys, range queries and multi-criteria queries.

PBFilter is today integrated in the storage manager of an embedded DBMS used in a field experiment to manage secure personal medical folders. Thanks to its tuning capabilities, PBFilter is expected to gracefully adapt to a wide range of embedded Flash-based platforms and application requirements. As future work, our ambition is to derive from PBFilter the complete design of a so-called low-cost database machine, with the objective to disseminate database features everywhere, up to the lightest smart objects (secure tokens, sensors, smart dongles, chips dedicated to transportation, healthcare, smart building, etc.). Beyond the embedded context, PBFilter seems well adapted to every environments where random writes are detrimental in terms of I/O cost, energy consumption, space occupancy or memory lifetime.

Acknowledgments

The authors wish to thank Nicolas Anciaux, Luc Bouganim and Björn ßór Jónsson for fruitful discussions on this work and Jean-Jacques Vandewalle and Patrick Enjolras for their technical support at Gemalto. This research is partially supported by the French National Agency for Research (ANR) under RNTL grant PlugDB and INS grant KISS and by the Natural Science Foundation of China under grants 60833005, 60573091.

References

[1] D., Agrawal, D., Ganesan, R., Sitaraman, and Y., Diao Lazy-Adaptive tree: An Optimized Index Structure for Flash Devices. In VLDB '09, 2009.
[2] T., Allard, N., Anciaux, L., Bouganim, Y., Guo, L., Le Folgoc, B., Nguyen, P., Pucheral, I., Ray, I., Ray, and S., Yin Secure Personal Data Servers: a Vision Paper. Proceedings of the International Conference on Very Large Data Bases (VLDB), Singapore, September 2010.
[3] N. Anciaux, L. Bouganim, P. Pucheral, P. Valduriez, DiSC: Benchmarking Secure Chip DBMS, IEEE Transactions on Knowledge and Data Engineering (IEEE TKDE) 20 (10) (2008).
[4] N., Anciaux, M., Benzine, L., Bouganim, K., Jacquemin, P., Pucheral, S., Yin Restoring the Patient Control over her Medical History. Proceedings of the International Symposium on Computer-Based Medical Systems (CBMS), Jyvaskyla, Finland, June 2008.
[5] A.-B., Bityutskiy, JFFS3 Design Issues. Tech. report, Nov. 2005.
[6] B. Bloom, Space/time tradeoffs in hash coding with allowable errors, Communications of the ACM 13 (7) (1970).
[7] L., Bouganim, B.ß., Jónsson and P., Bonnet uFLIP: Understanding Flash IO Patterns. CIDR, 2009.
[8] Chan, C., and Ioannidis, Y. Bitmap Index Design and Evaluation. International Conference on Management of Data (SIGMOD), 1998.
[9] F. Chen, D.A. Koufaty, X. Zhang, Understanding Intrinsic characteristics and system implications of flash memory based solid state drives, In SIGMETRICS '09 (2009) 181–192.
[10] B., Debnath, M.F., Mokbel, D.J., Lilja, and D., Du Deferred Updates for Flash-based Storage, msst, pp.1–6, 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies, 2010.
[11] B. Debnath, S. Sengupta, J. Li, FlashStore: high throughput persistent key-value store, Proceedings of VLDB Endow 3 (1–2) (2010) 1414–1425.
[12] B. Debnath, S. Sengupta, J. Li, Skimpystash: ram space skimpy key-value store on flash, Sigmod (2011).
[13] P.C., Dillinger, P., Manolios Fast and Accurate Bitstate Verification for SPIN. 11th International Spin Workshop on Model Checking Software, LNCS 2989, 2004.
[14] Eurosmart, ''Smart USB token'', white paper, 2008.
[15] Roh, H., Lee, D., and Park, S. Yet Another Write-optimized DBMS Layer for Flash-based Solid State Storage. Proceedings of the 19th ACM international conference on Information and knowledge management, October 26–30, 2010, Toronto, ON, Canada.
[16] Intel Corporation, Understanding the Flash Translation Layer (FTL) specification 1998.
[18] Y.R., Kim, K.Y., Whang, and I.Y., Song Page-Differential Logging: An Efficient and DBMS-independent Approach for Storing Data into Flash Memory. In SIGMOD 10, 2010.
[19] A., Kirsch, and M., Mitzenmacher Less Hashing, Same Performance: Building a Better Bloom Filter. Algorithms—ESA 2006, 14th European Symposium, LNCS 4168, 2006.
[20] S., Lee, B., Ko, and H., Yoo Flash-Aware Storage Optimized for Mobile and Embedded DBMS on NAND Flash Memory. United States Patent 7856522, 2010.
[21] S.W., Lee, and B., Moon Design of Flash-Based DBMS: An In-Page Logging Approach. Int. Conf. on Management of Data (SIGMOD), 2007.
[22] Y., Li, B., He, R.J., Yang, Q., Luo, and K., Yi Tree Indexing on Solid State Drives. In VLDB 10, 2010.
[23] Y., Li, J., Xu, B., Choi, and H., Hu StableBuffer: Optimizing Write Performance for DBMS Applications on Flash Devices. CIKM 10, 2010.
[24] A., Mani, M.B., Rajashekhar, and P., Levis TINX—A Tiny Index Design for Flash Memory on Wireless Sensor Devices. ACM Conference on Embedded Networked Sensor Systems (SenSys) 2006, Poster Session.
[25] S., Nath, and A., Kansal FlashDB: Dynamic Self-tuning Database for NAND Flash. International Conference on Information Processing in Sensor Networks (IPSN), 2007.
[26] M. Rosenblum, J.K. Ousterhout, The design and implementation of a log-structured file system, ACM Transactions on Computer Systems (TOCS) 10 (1) (1992).
[27] R., Stoica, M., Athanassoulis, R., Johnson, and A., Ailamaki Evaluating and Repairing Write Performance on Flash Devices. In DaMoN 09, 2009.
[29] C., Wu, L., Chang, and T., Kuo, An Efficient B-Tree Layer for Flash-Memory Storage Systems. International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA), 2003.
[30] YAFFS: Yet Another Flash File System. 〈http://www.yaffs.net〉.
[31] A., Yao, On Random 2–3 Trees. Acta Informatica, 9, 1978.
[32] S. Yao, Approximating the number of accesses in database organizations, Communication of the ACM 20 (4) (1977).
[33] S., Yin, P., Pucheral, X.A., Meng, Sequential Indexing Scheme for Flash-Based Embedded Systems. Proceedings of the International Conference on Extending Database Technology (EDBT), Saint-Petersburg, Russia, March 2009.
[34] D., Zeinalipour-Yazti, S.V., Lin, Kalogeraki, D., Gunopulos,and W., Najjar, MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices. USENIX Conference on File and Storage Technologies (FAST), 2005.

Please cite this article as: S. Yin, P. Pucheral, PBFilter: A flash-based indexing scheme for embedded systems, Information Systems (2012), doi:10.1016/j.is.2012.02.002