A Scalable Search Engine for Mass Storage Smart Objects

Nicolas Anciaux [1,2] Saliha Lallali [1,2] Iulian Sandu Popa [1,2] Philippe Pucheral [1,2]
[1]INRIA Rocquencourt
78153 Le Chesnay Cedex, France
Firstname.Lastname@inria.fr
[2]Université de Versailles Saint-Quentin-en-Yvelines
78035 Versailles Cedex, France
Firstname.Lastname@prism.uvsq.fr

ABSTRACT

This paper presents a new embedded search engine designed for smart objects. Such devices are generally equipped with extremely low RAM and large Flash storage capacity. To tackle these conflicting hardware constraints, conventional search engines privilege either insertion or query scalability but cannot meet both requirements at the same time. Moreover, very few solutions support document deletions and updates in this context. In this paper, we introduce three design principles, namely Write-Once Partitioning, Linear Pipelining and Background Linear Merging, and show how they can be combined to produce an embedded search engine reconciling high insert/delete/update rate and query scalability. We have implemented our search engine on a development board having a hardware configuration representative for smart objects and have conducted extensive experiments using two representative datasets. The experimental results demonstrate the scalability of the approach and its superiority compared to state of the art methods.

1. INTRODUCTION

With the continuous development of sensor networks, smart personal portable devices and Internet of Things, the tight combination of computing power and large storage capacities (Gigabytes) into many kinds of *smart objects* becomes reality. On the one hand, large Flash memories (GBs) are now adjunct to many small computing devices, e.g., SD cards plugged into sensors [19] or raw Flash chips superimposed on SIM cards [4]. On the other hand, microcontrollers are integrated into many memory devices, e.g., Flash USB keys, SD/micro-SD cards and contactless badges. Examples of smart objects combining computing power and large storage capacity (GBs of Flash memory) are pictured in Figure 1.

As smart objects gain the capacity to acquire, store and process data, new services emerge. Camera sensors tag photographs and provide search capabilities to retrieve them [22]. Smart objects maintain the description of the surrounding objects [23] enabling the Internet of Things [1], e.g. shops like bookstores can be queried directly by customers in search of a certain product. Users securely store their personal files (documents, photos, emails) in Personal Data Servers [3, 4, 12, 18]. Smart meters record energy consumption events and embedded GPS devices track locations and moves. A myriad of new applications and services are being built by querying these data. So far, all these data end up on centralized servers where they are analyzed and queried. This centralized model however has two main drawbacks. First, the ever increasing amount of data transferred from their source (the smart objects) to their destination (the servers) has a dramatic negative impact on environmental sustainability (i.e., energy waste) and costs (i.e., data transfer). Hence, significant energy and bandwidth savings can be obtained by storing data locally in smart objects, especially when dealing with large data and seldom used records [8, 19]. Second, centralizing data sensed about individuals introduces an unprecedented threat on data privacy (e.g., at 1Hz granularity, an electricity smart meter can reveal the precise activity of the inhabitants [3]).

Figure 1. Smart objects endowed with MCUs and NAND Flash

This explains the growing interest for transposing traditional data management functionalities directly into the smart objects. Simple query evaluation facilities have been recently proposed for sensor nodes equipped with large Flash memory [8, 9] to enable filtering operations. Relational database operations like selection, projection and join for new generations of SIM cards with large Flash storage capacities have been proposed in [4, 18]. More complex treatments such as facial recognition and the related indexing techniques have been investigated also.

In this paper, we make a step further in this direction by addressing the traditional problem of information retrieval queries over large file collections. A file can be any form of document, picture or data stream associated with a set of terms. A query can be any form of keyword search using a ranking function (e.g., *tf-idf*) identifying the top-k most relevant files. The proposed search engine can be used in sensors to search for relevant objects in their surroundings [17, 23], in cameras to search pictures by using tags [22], in personal smart dongles to secure the querying of documents and files hosted in an untrusted Cloud [4, 12] or in smart meters to perform analytic tasks (i.e., top-k queries) over sets of events (i.e., terms) captured during time windows (i.e., files) [3]. Hence, this engine can be thought of as a generalized Google desktop or Spotlight embedded in smart objects.

Designing such embedded search engine is however challenging due to a combination of severe and conflicting hardware constraints. Smart objects are usually equipped with a tiny RAM and their persistent storage is made of NAND Flash badly adapted to random fine-grain updates. Unfortunately, state-of-the-art indexing techniques either consume a lot of RAM or produce a large quantity of random fine-grain updates. Few pioneer works already considered the problem of embedding a search engine in sensors equipped with Flash storage [17, 20, 22, 23], but they

Proceedings of the VLDB Endowment, Vol. 8, No. 9

target small data collections (i.e., hundreds to thousands of files) with a query execution time which remains proportional to the number of indexed documents. By construction these search engines cannot meet insertion performance and query scalability at the same time. Moreover none of them support file deletions and updates, however useful in a number of scenarios.

In this paper, we make the following contributions:

- We introduce three design principles, namely *Write-Once Partitioning*, *Linear Pipelining* and *Background Linear Merging*, to devise an inverted index capable of tackling the conflicting hardware constraints of smart objects.
- Based on these principles, we propose a novel inverted index structure and related algorithms to support all the basic index operations, i.e., search, insertion, deletion and update.
- We validate our design through an extensive experimentation on a real smart object platform and with two representative datasets and show that query scalability and insertion/deletion/update performance can be met together in various contexts.

The rest of the paper is organized as follows. Section 2 details the smart objects' hardware constraints, analyses the state-of-the-art solutions and derives from this analysis a precise problem statement. Section 3 introduces our three design principles, while Sections 4 and 5 detail the proposed inverted index structure and algorithms derived from these principles. Section 6 is devoted to the tricky case of file deletions and updates. We present the experimental results in Section 7 and conclude in Section 8.

2. PROBLEM STATEMENT

2.1 Smart Objects' Hardware Constraints

Whatever their form factor and usage, smart objects share strong commonalities in terms of data management architecture. Indeed, a large NAND Flash storage is used to persistently store data and indexes, and a microcontroller (MCU) executes the embedded code, both being connected by a bus. Hence, the architecture inherits hardware constraints from both the MCU and the Flash.

The MCUs embedded in smart objects usually have a low power CPU, a tiny RAM (few KB) and a few MB of persistent memory (ROM, NOR or EEPROM) used to store the embedded code. The NAND Flash component (either raw NAND Flash chip or SD/micro-SD card) also exhibits strong limitations. In NAND Flash, the minimum unit for a read and a write operation is the page (usually 512 Bytes, also called a sector). Pages must be erased before being rewritten but the erase operation must be performed at a block granularity (e.g., 256 pages). Erases are then costly and a block wears out after about 10^4 repeated write/erase cycles. In addition, the pages have to be written sequentially in a block. Therefore, NAND Flash badly supports random writes. We observed this same bad behavior both with raw NAND Flash chips and SD/micro-SD cards. Our own measurements shown in Table 1 corroborate the ones published in [6].

Table 1. Measured performance of common SD cards

Time (ms) per I/O	Read	Seq.Write	Rand. Write
Kingston µSDHC	1.3	6.3	160
Lexar SDMI4GB-715	0.8	2.1	180
Samsung µSDHC Plus	1.3	2.9	315
SiliconPower SDHC	1.4	3.4	40

2.2 Search Engine Requirements

As in [17], we consider that the search engine of interest in this paper has similar functionality as a Google Desktop for smart objects. Hence, we use the terminology introduced in the Information Retrieval literature for full-text search. Then, a document refers to any form of data files, terms refers to any forms of metadata elements, term frequencies refer to metadata element weights and a query is equivalent to a full-text search.

Full-text search has been widely studied by the information retrieval community (see [24] for a recent survey). The core problem is, given a collection of documents and a user query expressed as a set of terms $\{t_i\}$, to retrieve the k most relevant documents according to a ranking function. In the wide majority of the related works, the *tf-idf* score, i.e., term frequency-inverse document frequency, is used to rank the query results. A document can be of many types (e.g., text file, image, etc.) and is associated with a set of terms (or keywords) describing its content and weights indicating their respective importance in the document. For text documents, the terms are words composing the document and their weight is their frequency in the document. For images, the terms can be tags, metadata or visterms describing image subparts [22]. For a query $Q=\{t\}$, the *tf-idf* score of each document d containing at least a query term can be computed as:

$$tf-idf(d) = \sum_{t \in Q} \log\left(f_{d,t}+1\right) \cdot \log\left(\frac{N}{F_t}\right)$$

where $f_{d,t}$ is the frequency of term t in document d, N is the total number of indexed documents and F_t is the number of documents that contain t. This formula is given for illustrative purpose, the weight between $f_{d,t}$ and N/F_t varying depending on the proposals.

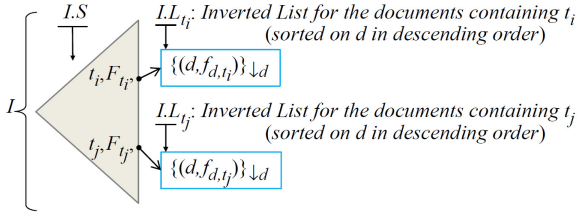


Figure 2. Typical inverted index structure.

Classically, full-text search queries are evaluated efficiently using an inverted index, named I hereafter (see Figure 2). Given $D=\{d_i\}$ a set of documents, the inverted index I over D consists of two main components [24]: (*i*) a search structure $I.S$ (also called dictionary) which stores for each term t appearing in the documents the number F_t of documents containing t and a pointer to the inverted list of t; (*ii*) a set of inverted lists $\{I.L_t\}$ where each list stores for a term t the list of (d, $f_{d,t}$) pairs where d is a document identifier in D that contains t and $f_{d,t}$ is the weight of the term t in the document d (typically the frequency of t in d). The dictionary is constituted by all the distinct terms t of the documents in D, and is large in practice, which requires organizing it into a search-efficient structure such as a B-tree.

A query $Q=\{t\}$ is traditionally evaluated by: (*i*) accessing $I.S$ to retrieve for each query term t the inverted lists elements $\{I.L_t\}_{t \in Q}$; (*ii*) allocating in RAM one container for each unique document identifier in these lists; (*iii*) computing the score of each of these documents using a weight function, e.g., *tf-idf*; (*iv*) ranking the documents according to their score and producing the k documents with the highest scores.

2.3 State-of-the-Art Solutions

Data management embedded in smart objects is no longer a new topic. Many proposals from the database community tackle this problem in the context of the Internet of Things, strengthening the idea that smart objects must now be considered as first-class data sources. However, many works [3, 4, 19] consider a traditional database context and do not address the full-text search problem, leading to different query processing techniques and indexing structures. Therefore, we focus below on works specifically addressing embedded search engines and then extend the review to a few works related to Flash-based indexing when the way they tackle the MCU or Flash constraints can enlighten the discussion.

Embedded search engines. A few pioneer works recently demonstrate the interest of embedding search engine techniques into smart objects equipped with extended Flash storage to manage collections of files stored locally [16, 17, 20, 22]. These works rely on a similar design of the embedded inverted index as proposed in [17]. Instead of maintaining one inverted list per term in the dictionary, each term is hashed to a bucket and a single inverted list is built for each bucket. The inverted lists are stored sequentially in Flash memory, within chained pages, and only a small hash table referencing the head of each bucket is kept in RAM. The number of buckets is kept small such that the main part of the RAM can be used as an insertion buffer for the inverted lists elements, i.e., (t, d, $f_{d,t}$) triples. This approach complies with a small RAM and suits well the Flash constraints by precluding fine grain random (re)writes in Flash. However, each inverted lists corresponds to a large number of different terms, which leads to a high query evaluation cost that increases proportionally with the size of the data collection. The less RAM available, the smaller the number of hash buckets and the more severe the problem is. In addition, these techniques do not support document deletions, but only data aging mechanisms, where old index entries automatically expire when overwritten by new ones. A similar design is proposed in [22] that builds a distributed search engine to retrieve images captured by camera sensors. A local inverted index is embedded in each sensor node to retrieve the relevant images locally, before conducting the distributed search. However, this work considers powerful sensors nodes (with tens of MB of local RAM) equipped with custom SD card boards (with specific performances). At the same time, the underlying index structure is based on inverted lists organized in a similar way as in [17]. All these methods are highly efficient for document insertions, but fail to provide scalable query processing for large collections of documents. Therefore, their usage is limited to applications that require storing only a small number (few hundreds) of documents.

B-tree indexing in NAND Flash. In the database context, adapting the B-tree to NAND Flash has received a great attention. Indeed, the B-tree is a very popular index and its standard implementation performs poorly in Flash [21]. Many recent proposals [2, 13, 21] tackle this problem or the closely related problem of a key-value stored in Flash [7]. The key idea in these approaches is to buffer the updates in log structures that are written sequentially and to leverage the fast (random) read performance of Flash memory to compensate the loss of optimality of the lookups. When the log is large enough, the updates are committed into the B-tree in a batch mode, to amortize the Flash write cost. The log must be indexed in RAM to ensure performance. The different proposals vary in the way the log and the in-memory index are managed, and in the impact it has on the commit frequency. To amortize the write cost by a significant factor, the log must be seldom committed, which requires more RAM. Conversely, limiting the RAM size leads to increasing the commit frequency, thus generating more random writes. The RAM consumption and the random write cost are thus conflicting parameters. Under severe RAM limitations, virtually no reduction of random writes can be obtained.

Partitioned indexes. In another line of work, partitioned indexes have been extensively employed in environments with insert-intensive workloads and concurrent queries on magnetic disks. Prominent examples are the LSM-tree (i.e., the Log-Structured Merge-tree) [14] and its many variants (e.g., the Partitioned Exponential file [10] and the bLSM-tree [15] to name but a few). The LSM-tree consists in one in-memory B-tree component to buffer the updates and one on-disk B^+-tree component that indexes the disk resident data. Periodically, the two components are merged to integrate the in-memory data and free the memory. The benefit is twofold. First the updates are integrated in batch, which amortizes the write cost per update. Second, the merge operation uses sequential I/Os, which reduces the disk arm movements and thus increases the throughput. Note that Google's Bigtable and Facebook's Cassandra employ a similar partitioning approach to optimize the indexing of key-value stores in massively parallel and distributed architectures. The search engine proposed in this paper shares the general idea of index partitioning. However, the similarity stops at the general level because of major differences regarding the targeted hardware platforms (embedded systems versus high-end servers) and the queries of interest (top-k keyword search versus classical key-value search). Consequently, our solution differs from the above mentioned ones in a number of aspects (e.g., RAM usage, partitioning organization, management of updates and deletions, way of computing the top-k with a minimal RAM consumption).

As a conclusion, tiny RAM and NAND Flash persistent storage introduce conflicting constraints and lead to split state of the art solutions in two families. The *insert-optimized family* reaches insertion scalability thanks to a small indexed structure buffered in RAM and sequentially flushed in Flash, thereby precluding costly random writes in Flash. This good insertion behavior is however obtained to the detriment of query scalability, the performance of searches being roughly linear with the index size in Flash. Conversely, the *query-optimized family* reaches query scalability by adapting traditional indexing structures to Flash storage, to the detriment of insertion scalability, the number of random (re)writes in Flash (linked to the log commit frequency) being roughly inversely proportional to the RAM capacity. In addition, we are not aware of works addressing the crucial problem of random document deletions in the context of an embedded search engine.

2.4 Problem Formulation

In the light of the preceding sections, the problem addressed in this paper can be formulated as *designing an embedded full-text search engine that has the following two properties*:

- *Bounded RAM agreement*: the proposed engine must be able to respect a predefined RAM consumption bound (RAM_Bound),

precluding any solution where this consumption depends on the size of the document set.
- *Full scalability*: it must be scalable for queries and updates (insertion, deletion of documents) without distinction.

The Bounded RAM agreement is required to comply with the widest population of smart objects. The consequence is that the full-text search engine must remain functional even when very little RAM (a few KB) is made available to it. Note that the RAM_Bound size is a subpart of the total physical RAM capacity of a smart object considering that the RAM resource is shared by all software components running in parallel on the platform, including the operating system. The RAM_Bound property is also mandatory in a co-design perspective where the hardware resources of a given platform must be precisely calibrated to match the requirements of a particular application domain.

The Full scalability property guarantees the generality of the approach. By avoiding to privilege a particular workload, the index can comply with most applications and datasets. To achieve update scalability, the index maintenance needs to be processed without generating random writes, which are badly supported by the Flash memory. At the same time, achieving query scalability means obtaining query execution costs in the same order of magnitude with the ideal query costs provided by a classical inverted index *I*.

3. DESIGN PRINCIPLES

Satisfying the Bounded RAM agreement and Full scalability properties simultaneously is challenging, considering the conflicting MCU and Flash constraints mentioned above. To tackle this challenge, we propose in this paper an indexing method that relies on the following three design principles.

P1. Write-Once Partitioning: *Split the inverted index structure I in successive partitions such that a partition is flushed only once in Flash and is never updated*.

By precluding random writes in Flash, Write-Once Partitioning aims at satisfying update scalability. Considering the Bounded RAM agreement, the consequence of this principle is to parse documents and maintain *I* in a streaming way. Conceptually, each partition can be seen as the result of indexing a window of the document input flow, the size of which is limited by the RAM_Bound. Therefore, *I* is split in an infinite sequence of partitions $<I_1, I_2,\ldots, I_p>$, each partition I_i having the same internal structure as *I*. When the size of the current I_i partition stored in RAM reaches RAM_Bound, I_i is flushed in Flash and a new partition I_{i+1} is initialized in RAM for the next window.

A second consequence of this design principle is that document deletions have to be processed similar to document insertions since the partitions cannot be modified once they are written. This means adding compensating information in each partition that will be considered by the query process to produce correct results.

P2. Linear Pipelining: *Compute each query Q with respect to the Bounded RAM agreement in such a way that the execution cost of Q over* $<I_1, I_2,\ldots, I_p>$ *is in the same order of magnitude as the execution cost of Q over I*.

Linear Pipelining aims at satisfying query scalability under the Bounded RAM agreement. A unique structure *I* as the one pictured in Figure 2 is assumed to satisfy query scalability by nature and is considered hereafter as providing a lower bound in terms of query execution time. Hence, the objective of Linear pipelining is to keep the performance gap between *Q* over $<I_1, I_2,\ldots, I_p>$ and *Q* over *I*, both small and predictable (bounded by a given tuning parameter). Computing *Q* as a set-oriented composition of a set of Q_i over I_i, (with i=0,...p) would unavoidably violate the Bounded RAM agreement as *p* increases, since it will require to store all Q_i's intermediate results in RAM. Hence the necessity to organize the processing in pipeline such that the RAM consumption remains independent of *p*, and therefore of the number of indexed documents. Also, the term *linear* pipelining conveys the idea that the query processing must preclude any iteration (i.e., repeated accesses) over the same data structure to reach the expected level of performance. This disqualifies brute-force pipeline solutions where the *tf-idf* scores of documents are computed one after the other, at the price of reading the same inverted lists as many times as the number of documents they contain.

However, Linear Pipelining alone cannot prevent the performance gap between *Q* over $<I_1, I_2,\ldots, I_p>$ and *Q* over *I* to increase with the increase of *p* as (*i*) multiple searches in several small $I_i.S$ are more costly than a single search in a large *I.S* and (*ii*) the inverted lists in $<I_1, I_2,\ldots, I_p>$ are likely to occupy only fractions of Flash pages, multiplying the number of Flash I/Os to access the same amount of data. A third design principle is then required.

P3. Background Linear Merging: *To limit the total number of partitions, periodically merge partitions compliantly with the Bounded RAM agreement and without hurting update scalability*.

The objective of partition merging is therefore to obtain a lower number of larger partitions to avoid the drawbacks mentioned above. Partition merging must meet three requirements. First the merge must be performed in pipeline to comply with the Bounded RAM agreement. Second, since its cost can be significant (i.e., proportional to the total size of the merged partitions), the merge must be processed in background to avoid locking the index structure for unbounded periods of time. Since multi-threading is not supported by the targeted platforms, background processing can simply be understood as the capacity to interrupt and recover the merging process at any time. Third, update scalability requires that the total cost of a merge run be always smaller than the time to fill out the next bunch of partitions to be merged.

Taken together, principles P1 to P3 reconcile the Bounded RAM agreement and Full scalability index properties. The technical solutions to implement these three principles are presented next. To ease the presentation, we introduce first the foundation of our solution considering only document insertions and queries. The trickier case of document deletions is postponed to Section 6.

4. WRITE-ONCE PARTITIONING AND LINEAR PIPELINING

These two design principles are discussed together because the complexity comes from their combination. Indeed, Write-Once Partitioning is straightforward on its own. It simply consists in splitting *I* in a sequence $<I_1, I_2,\ldots, I_p>$ of small indexes called partitions, each one having a size bounded by RAM_Bound. The difficulty is to implement a linear pipeline execution of any query *Q* on this sequence of partial indexes.

Executing Q over I would lead to evaluate:

$$Top_k\left[\sum_{t\in Q} W\left(f_{d,t}, \frac{N}{F_t}\right)\right], \text{ with } d \in D$$

where Top_k selects the k documents $d \in D$ having the largest *tf-idf* scores, each score being computed as the sum, for all terms $t \in Q$, of a given weight function W taking as parameter the frequency $f_{d,t}$ of t in d and the inverse document frequency N/F_t. Our objective is to remain agnostic regarding W and then let the precise form of this function open. Let us now consider how each term of this expression can be evaluated by a linear pipelining process on a sequence $<I_1, I_2, \ldots, I_p>$.

Computing N. We assume that the number of documents is a global metadata maintained at insertion/deletion time and needs not be recomputed for each Q.

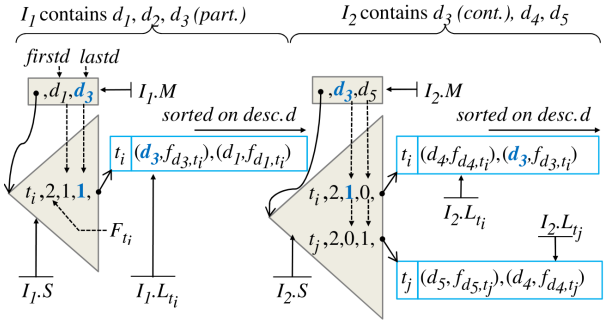


Figure 3. Consecutive index partitions (overlapping docs).

Computing F_t**.** F_t should be computed only once for each term t since F_t is constant for Q. This is why F_t is materialized in the dictionary part of the index ($\{t, F_t\} \subset I.S$), as shown in Figure 2. When I is split in $<I_1, I_2, \ldots, I_p>$, the global value of F_t should be computed as the sum of the local F_t of all partitions. The complexity comes from the fact that a same document d may cross several partitions with the consequence of contributing several times to the global F_t if a simple sum is performed. The Bounded RAM agreement precludes maintaining in RAM a history of all the terms already encountered for a given document d across the parsing windows, the size of this history being unbounded. Accessing the inverted lists $\{I_i.L_t\}$ of successive partitions to check whether they intersect for a given d would also violate the Linear Pipelining since these same lists will be accessed again when computing the *tf-idf* score of each document.

The solution is then to store in the dictionary of each partition the boundary of that partition, namely the identifiers of the first and last documents considered in the parsing window. Then, two bits *firstd* and *lastd* are added in the dictionary for each inverted list to register whether this list contains one (or both) of these documents, i.e., $\{t, F_t, firstd, lastd\} \subset I.S$. As illustrated in Figure 3, this is sufficient to detect the intersection between the inverted lists of a same term t in two successive partitions. Whether an intersection between two lists is detected, the sum of their respective F_t is decremented by 1. Hence, the correct global value of F_t can easily be computed without physically accessing the inverted lists. During the F_t computation phase, the dictionary of each partition is read only once and the RAM consumption sums up to one buffer to read each dictionary, page by page, and one RAM variable to store the current value of each F_t.

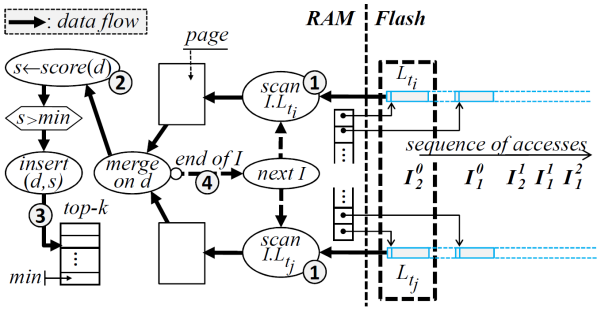


Figure 4. Linear Pipeline computation of Q over terms t_i and t_j.

Computing $f_{d,t}$ **.** If a document d overlaps two consecutive partitions I_i and I_{i+1}, the inverted list L_t of a queried term $t \in Q$ may also overlap these two partitions. In this case the $f_{d,t}$ score of d is simply the sum of the (last) $f_{d,t}$ value in $I_i.L_t$ and the (first) $f_{d,t}$ value in $I_{i+1}.L_t$. To get the $f_{d,t}$ values, the inverted lists $I_i.L_t$ have to be accessed. The pointers referencing these lists are actually stored in the dictionary which has already been read while computing F_t. According to the Linear pipelining principle, we avoid reading again the dictionary by storing these pointers in RAM during the F_t computation. The extra RAM consumption is minimal and bounded by the fact that the number of partitions is itself bounded thanks to the merging process (see Section 5).

Computing Top_k**.** Traditionally, a RAM variable is allocated to each document d to compute its *tf-idf* score by summing the results of $W(f_{d,t}, N/F_t)$ for all terms $t \in Q$ [24]. Then, the k best scores are selected. Unfortunately, this approach conflicts with the Bounded RAM agreement since the size of the document set is likely to be much larger than the available RAM. Hence, we organize the query processing in a pure pipeline way, allocating a RAM variable only to the k documents having currently the best scores. This forces the complete computation of *tf-idf*(d) to be done for each d, one after the other. To meet this requirement while precluding any iteration on the inverted lists, these lists are maintained sorted on the document id. Note that if document ids reflect the insertion ordering, the inverted lists are naturally sorted. Hence, the *tf-idf* computation sums up to a simple linear pipeline merging process of the inverted lists for all terms $t \in Q$ in each partition (see Figure 4). The RAM consumption for this phase is therefore restricted to one variable for each of the current k best *tf-idf* scores and to one buffer (i.e., a RAM page) per query term t to read the corresponding inverted lists $I_i.L_t$ (i.e., $I_i.L_t$ are read in parallel for all t, the inverted lists for the same t being read in sequence). Figure 4 summarizes the data structures maintained in RAM and in Flash to handle this computation.

5. BACKGROUND LINEAR MERGING

The background merging process aims at achieving scalable query costs by timely merging several small indexes into a larger index structure. As mentioned in Section 3, the merge must be a pipeline process in order to comply with the Bounded RAM agreement while keeping a cost compatible with the update rate. Moreover, the query processing should continue to be executed in Linear Pipelining (see Section 4) on the structure resulting from the successive merges. Therefore, the merges have to preserve the global ordering of the document ids within the index structures.

To meet these requirements, we introduce a Sequential and Scalable Flash structure, called *SSF*, pictured in Figure 5. The

SSF consists in a hierarchy of partitions of exponentially increasing size. Specifically, each new index partition is flushed from RAM into the first level of the *SSF*, i.e., L_0. The *merge* operation is triggered automatically when the number of partitions in a level becomes b, the branching factor of *SSF*, which is a predefined index parameter. The merge combines the b partitions at level L_i of SSF, denoted by $I_1^i,...I_b^i$, into a new partition at level L_{i+1}, denoted by I_j^{i+1} and then reclaims all partitions at level L_i.

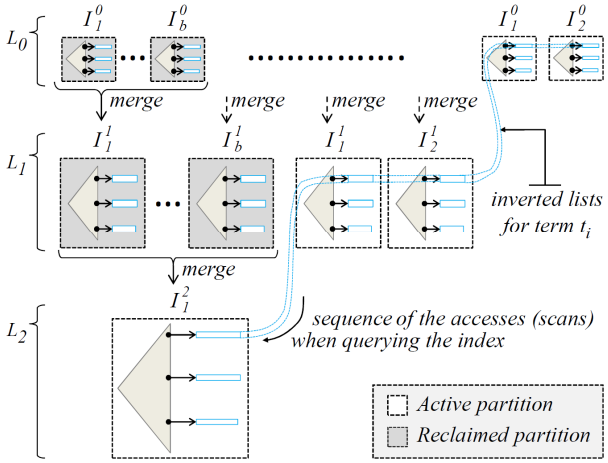


Figure 5. The Scalable and Sequential Flash structure.

The *merge* is directly processed in pipeline as a multi-way merge of all partitions at the same level. This is possible since the dictionaries of all the partitions are already sorted on terms, while the inverted lists in each partition are also sorted on document ids. So are the dictionary and the inverted lists of the resulting partition at the upper level. More precisely, the algorithm works in two steps. In the first step, the *I.L* part of the output partition is produced. Given b partitions in the index level L_i, b+1 RAM pages are necessary to process the merge in linear pipeline: b pages to merge the inverted lists in *I.L* of all b partitions and one page to produce the output. The indexed terms are treated one after the other in alphabetic order. For each term t, the head of its inverted lists in each partition is loaded in RAM. These lists are then consumed in pipeline by a multi-way merge. Document ids are encountered in descending order in each list and the output list resulting from the merge is produced in the same order. A particular case must be distinguished when two pairs $(d, f1_{d,t})$ and $(d, f2_{d,t})$ are encountered in separate lists for the same d; this means that document d overlaps two partitions and these two pairs are aggregated in a single $(d, f1_{d,t} + f2_{d,t})$ before being added to *I.L*. In the second step, the metadata *I.M* is produced (see Figure 3), by setting the value of *firstd* (resp. *lastd*) with the *firstd* (resp. *lastd*) value of the *first* (resp. *last*) partition to be merged, and the *I.S* structure is constructed sequentially, with an additional scan of *I.L*. The *I.S* tree is built from the leaves to the root. This step requires one RAM page to scan *I.L*, plus one RAM page per *I.S* tree level. For each list encountered in *I.L*, a new entry (t, F_t, *presence_flags*) is appended to the lowest level of *I.S*; the value F_t is obtained by summing the $f_{d,t}$ fields of all $(d, f_{d,t})$ pairs in this list; the presence flag reflects the presence in the list of the *firstd* or *lastd* document. Upper levels of *I.S* are then trivially filled sequentially. This Background Merging process generates only sequential writes in Flash and previous partitions are reclaimed in large blocks after the merge. This pipeline process sequentially scans each partition only once and produces the resulting partition also sequentially. Hence, assuming b+1 is strictly lower than RAM_bound, one RAM buffer (of one page) can be allocated to read each partition and the merge is I/O optimal. If b is larger than RAM_bound, the algorithm remains unchanged but its I/O cost increases since each partition will be read by page fragments rather than by full pages.

Search queries can be evaluated in linear pipeline by accessing the partitions one after the other from partitions b to 1 in level 1 up to level n. Thus, the inverted lists are scanned in descending order of the document ids, from the most recently inserted document to the oldest one, and the query processing remains exactly the same as the one presented in Section 4, with the same RAM consumption. The merging and the querying processes could be organized in opposite order (i.e., in ascending order of the document ids) with no impact. However, order matters as soon as deletions are considered (see Section 6). *SSF* provides scalable query costs since the amount of indexed documents grows exponentially with the number of levels, while the number of partitions increases only linearly with the number of levels.

Note that merges in the upper levels are exponentially rare (one merge in level L_i for b^i merges in L_0) but also exponentially costly. To mitigate this problem, we perform the merge operations in background (i.e., in a non-blocking manner). Since the merge may consume up to b pages of RAM, we launch/resume it each time after a new partition is flushed in L_0 of the *SSF*, the RAM being empty at this time. A small quantum of time (a few hundred milliseconds in practice) is allocated to the merging process. Each time this quantum expires, the merge is interrupted and its execution status (i.e., a cursor indicating the current Flash page position in each partition) is memorized. The quantum of time is chosen so that the merge of a given *SSF* level ends before the next merge of the same level need to be triggered. In this way, the cost of a merge operation is spread among the flush operations and remains almost transparent. This basic strategy is simple and does not make any assumption regarding the index workload. However, it could be improved in certain contexts, by taking advantage of the idle time of the platform.

6. DOCUMENT DELETIONS

To the best of our knowledge, our proposal is the first embedded search index to implement document deletions. This problem is actually of primary importance because deletions are required in many practical scenarios. Unfortunately, index updating increases significantly the complexity of the index maintenance by reintroducing the need for random updates in the index structure. We extend here the index structure to support the deletions of documents without generating any random write in Flash.

6.1 Solution Outline

Implementing the delete operation is challenging, mainly because of the Flash memory constraints which proscribe the straightforward approach of updating in-place the inverted index. The alternative to updating in-place is *compensation*, i.e., the deleted documents' identifiers (*DDIs*) are stored in an appropriate way and used as a filter to eliminate the ghost documents retrieved by the query evaluation process.

A basic solution could be to organize the *DDIs* as a sorted list in Flash and to intersect this list at query execution time with the inverted lists in the *SSF* corresponding to the query terms. However, this solution raises several problems. First, the documents are deleted in random order, according to users' and application decisions. Hence, maintaining a sorted list of *DDIs* in Flash would violate the Write-Once Partitioning principle since the list has to be rewritten each time a set (e.g., a page) of new *DDIs* is flushed from RAM. Second, the computation of the F_t for each query term t during the first step of the query processing cannot longer be achieved without an additional merge operation to subtract the sorted list of *DDIs* from the inverted lists of the *SSF*. Third, the full *DDI* list has to be scanned for each query regardless of the query selectivity. These two last elements make the query cost dependent of the total number of deleted documents and then conflict with the Linear pipelining principle.

Therefore, instead of compensating the query evaluation process, we propose a solution based on compensating the indexing structure itself. In particular, a document deletion is treated similarly to a document insertion, i.e., by re-inserting the metadata (terms and frequencies) of all deleted documents in the *SSF*. The objective is threefold: (*i*) to be able to compute, as presented in Section 4, the F_t for each term t of a query based on the metadata only (of both existing and deleted documents), (*ii*) to have a query performance that depends on the query selectivity (i.e., number of inserted and deleted documents relevant to the query) and not on the total number of deleted documents and (*iii*) to effectively purge the indexing structure from the largest part of the deleted documents at Background Merging time, while remaining compliant with the Linear Pipelining principle. We present in the following the required modifications of the index structure to integrate this form of compensation.

6.2 Impact on Write-Once Partitioning

As indicated above, a document deletion is treated similarly to a document insertion. Assuming a document d is deleted in the time window corresponding to a partition I_i, a pair $(d, -f_{d,t})$ is inserted in each list $I_i.L_t$ for the terms t present in d and the F_t value associated to t is decremented by 1 to compensate the prior insertion of that document. To distinguish between an insertion and a deletion, the frequency value $f_{d,t}$ for the deleted document *id* is simply stored as a negative value, i.e., $-f_{d,t}$.

6.3 Impact on Linear Pipelining

Executing a query Q over our compensated index structure sums up to evaluate:

$$Top_k\left[\sum_{t\in Q} W\left(|f_{d,t}|, \frac{N}{F_t}\right)\right], \text{with } d \in (D^+ - D^-)$$

where D^+ (resp. D^-) represents the set of inserted (resp. deleted) documents.

Computing N. As presented earlier, N is a global metadata maintained at update time and then already integrates all insert and delete operations.

Computing F_t. The global F_t value for a query term t is computed as usual since the local F_t values are compensated at deletion time (see above). The case of deleted documents that overlap with several consecutive partitions is equally treated as with the inserted documents.

Computing $f_{d,t}$. The $f_{d,t}$ of a document d for a term t is computed as usual, with the salient difference that a document which has been deleted appears twice: with the value $(d, f_{d,t})$ (resp. $(d, -f_{d,t})$) in the inverted lists of the partition I_i (resp. partition I_j) where it has been inserted (resp. deleted). By construction $i < j$ since a document cannot be deleted before being inserted.

Computing Top_k. Integrating deleted documents makes the computation of Top_k more subtle. Following the Linear Pipelining principle, the *tf-idf* scores of all documents are computed one after the other, in descending order of the document ids, thanks to a linear pipeline merging of the insert lists associated to the queried terms. To this end, the algorithm introduced in Section 4 uses k RAM variables to maintain the current k best *tf-idf* scores and one buffer (i.e., a RAM page) per query term t to read the corresponding inverted lists. Some elements present in the inverted lists correspond actually to deleted documents and must be filtered out. The problem comes from the fact that documents are deleted in random order. Hence, while inverted lists are sorted with respect to the insertion order of documents, a pair of the form $(d, -f_{d,t})$ may appear anywhere in the lists. In case a document d has been deleted, the unique guarantee is to encounter the pair $(d, -f_{d,t})$ before the pair $(d, f_{d,t})$ if the traversal of the lists follows a descending order of the document ids. However, maintaining in RAM the list of all encountered deleted documents in order to filter them out during the follow-up of the query processing would violate the Bounded RAM agreement.

The proposed solution works as follows. The *tf-idf* score of each document d is computed by considering the modulus of the frequencies values $|\pm f_{d,t}|$ in the *tf-idf* score computation, regardless of whether d is a deleted document or not. Two lists are maintained in RAM: $Top_k = \{(d, score(d))\}$ contains the current k best *tf-idf* scores of documents which exist with certainty (no deletion has been encountered for these documents); $Ghost = \{(d, score(d))\}$ contains the list of documents which have been deleted (a pair $(d, -f_{d,t})$ has been encountered while scanning the inverted lists) and have a score better than the smallest score in Top_k. Top_k and *Ghost* lists are managed as follows. If the score of the current document d is worse than the smallest score in Top_k, it is simply discarded and the next document is considered (step 2 in Figure 6). Otherwise, two cases must be distinguished. If d is a deleted document (a pair $(d, -f_{d,t})$ is encountered), then it enters the *Ghost* list (step 3); else it enters the Top_k list unless its id is already present in the *Ghost* list (step 4). Note that this latter case may occur only if the id of d is smaller than the largest *id* in *Ghost*, making the search in *Ghost* useless in many cases. An important remark is that the *Ghost* list has to register only the deleted documents which may compete with the k best documents, to filter them out when these documents are later encountered, which makes this list very small in practice.

While simple in its principle, this algorithm deserves a deeper discussion in order to evaluate its real cost. This cost actually depends on whether the *Ghost* list can entirely reside in RAM or not. Let us compute the nominal size of this list in the case where the deletions are evenly distributed among the document set. For illustration purpose, let us assume k=10 and the percentage of deleted documents δ=10%. Among the first 11 documents encountered during the query processing, 10 will enter the Top_k

list and 1 is likely to enter the *Ghost* list. Among the next 11 documents, 1 is likely to be deleted but the probability that its score is in the 10 best scores is roughly 1/2. Among the next 11 ones, this probability falls to about 1/3 and so on and so forth. Hence, the nominal size of the *Ghost* list is $\delta \cdot k \cdot \sum_{i=1}^{n} 1/i$, which can be approximated by $\delta \cdot k \cdot (\ln(n) + \varepsilon)$. For 10.000 queried documents, n=1000 and the size of the *Ghost* list is only $\delta \cdot k \cdot (\ln(n) + \varepsilon) \approx 10$ elements, far beyond the RAM size. In addition, the probability that the score of a *Ghost* list element competes with the Top_k ones decreases over time, giving the opportunity to continuously purge the *Ghost* list (step 5 in Figure 6). In the very improbable case where the *Ghost* list overflows (step 6 in Figure 6), it is sorted in descending order of the document ids, and the entries corresponding to low document ids are flushed. This situation remains however highly improbable and will concern rather unusual queries (none of the 300 queries we evaluated in our experiment produced this situation, while allocating a single RAM page for the *Ghost* list).

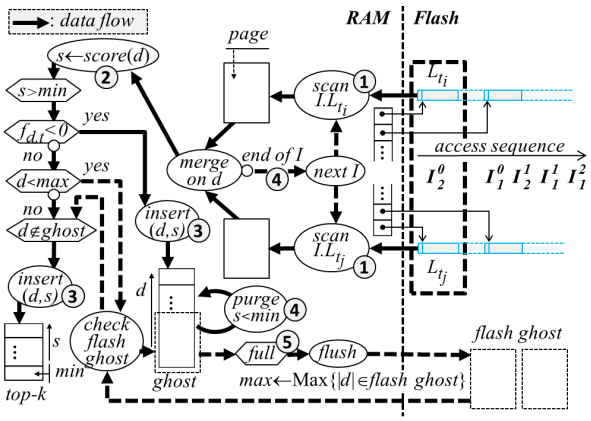


Figure 6. Linear pipeline computation of *Q* with deletions.

6.4 Impact on Background Pipeline Merging

The main purpose of the Background Merging principle, as presented in Section 5, is to keep the query processing scalable with the indexed collection size. The introduction of deletions has actually a marginal impact on the merge operation, which continues to be efficiently processed in linear pipeline as before. Moreover, given the way the deletions are processed in our structure, i.e., by storing couples (d, $-f_{d,t}$) for the deleted documents, the merge acquires a second function which is to absorb the part of the deletions that concern the documents present in the partitions that are merged. Indeed, let us come back to the Background Merging process described in Section 5. The main difference when deletes are considered is the following. When inverted lists are merged during step 1 of the algorithm, a new particular case may occur, that is when two pairs (d, $f_{d,t}$) and (d, $-f_{d,t}$) are encountered in separate lists for the same d; this means that document d has actually been deleted; d is then purged (the document deletion is absorbed) and will not appear in the output partition. Hence, the more frequent the Background Merging, the smaller the number of deleted entries in the index.

Taking into account the supplementary function of the merge, i.e., to absorb the data deletions, we can adjust the absorption rate of deletions by tuning the branching factor of the last index level since most of the data is stored in this index level. By setting a smaller value to the branching factor b' of the last level, the merge frequency in this level increases and consequently the absorption rate also increases. Therefore, in our implementation we use a smaller value for the branching factor of the last index level (i.e., b'=3 for the last level and b=10 for the other levels). Typically, about half of the total number of deletions will be absorbed for b'=3 if we consider that the deletions are uniformly distributed over the data insertions.

7. EXPERIMENTAL EVALUATION

7.1 Experimental Setup

Hardware platform. All the experiments have been conducted on a development board ST3221G-EVAL (see www.st.com/web/en/catalog/tools/PF251702) equipped with the MCU STM32-F217IG (see www.st.com/web/catalog/mmc/FM141/SC1169/SS1575/LN9/PF250172) connected to a micro-SD card slot. This hardware configuration is representative of typical smart objects [4, 17, 19]. The board runs the embedded operating system RTOS 7.0.1 (see freertos.svn.sourceforge.net/viewvc/freertos/tags/V7.1.0/). The search engine code is stored on the internal NOR Flash memory of the MCU, while the inverted index is stored on a micro-SD NAND Flash card. We used for data storage two commercial micro-SD cards (i.e., Kingston MicroSDHC Class 10 4GB and Silicon Power SDHC Class 10 4GB) exhibiting different performance (see lines 1 and 4 of Table 1). The MCU has 128KB of available RAM. However, the search engine only uses a maximum amount of 5KB of RAM, to validate our design whatever the available RAM of existing smart objects and the RAM consumption of the OS and the communication drivers.

Table 2. Desktop and synthetic datasets and query sets

	Desktop	Synthetic
Number of documents	27000	100000
Total Raw Text	63 MB	129 MB
Total Unique Words	337952	10,000
Total Word Occurrences	35624875	10000000
Average Occurrences per Word	26	988
Frequent Words	20752	1968
Infrequent Words	317210	8032
Frequent Word Occurrences	6.14%	19.68%
Infrequent Word Occurrences	93.85%	80.32%
KB per documents (avg, max)	8, 647	1.3, 1.3
Words per documents (avg, max)	1304, 105162	100, 100
Total number of queries	837	1000
Queries with 1, 2 & 3 terms	85, 255, 272	200, 200, 200
Queries with 4 & 5 terms	172, 82	200, 200

Datasets and queries. Selecting a representative data and query set to evaluate our solution is challenging considering the diversity and quick evolution of smart object usages, explaining the absence of recognized benchmarks in this area. We then validate our proposal using two use-cases where an embedded keyword-based search engine is called to play a central role and which exhibit different requirements in terms of document indexing with the objective to assess the versatility of the solution.

Table 3. Statistics of the flush and merge operations

	Flush [$RAM \rightarrow L_0$]	Merge [$L_0 \rightarrow L_1$]	Merge [$L_1 \rightarrow L_2$]	Merge [$L_2 \rightarrow L_3$]	Merge [$L_3 \rightarrow L_4$]	Merge [$L_4 \rightarrow L_5$]
Number of Read IOs	1 (1)*	90 (92)	503 (617)	2027 (2570)	11010(15211)	50997(73026)
Number of Write IOs	9 (9)	71 (100)	339 (548)	1485 (2085)	9409 (14027)	47270 (66335)
Exec. time on Kingston (sec.)	0.008 (0.0084)	0.58 (0.77)	2.9 (4.44)	13.2 (19.1)	84.6 (124.4)	436 (615)
Exec. time on Silicon Power (sec.)	0.004 (0.0045)	0.38 (0.48)	1.94 (2.84)	8.67 (11.7)	54.5 (79.3)	278 (393)
Total number of occurrences	73277	9160	1145	143	18	2
Inserted docs between flushes/merges	0.42 (16)	3 (42)	24 (232)	189 (1193)	1453 (6496)	8906 (10547)

* The numbers given in brackets are maximum values, other values are average values.

The first use-case relates to the *Personal Cloud* where a secure smart object embedding a Personal Data Server [3, 4, 18] is used to securely store, query and share personal user's files (documents, photos, emails). These files can be stored encrypted on a regular server (e.g., in the Cloud or in a PC) but the metadata describing them (keywords extracted from the file content, date, type, tags set by the user herself, etc.) are indexed and stored by the embedded Personal Data Server acting as a Google Desktop or Spotlight for the user's dataspace [12]. Indeed, the metadata are as sensitive as the files themselves and must be managed in the secure smart object (e.g., a mass storage smartcard connected to a PC or an internet gateway) to prevent any privacy breach. This use-case is representative of situations where the indexing documents have a rich content (tens to hundreds of thousands of terms) and documents updates and deletes can be performed randomly. To capture the behavior of our solution in such context, we use the pseudo-desktop data collection and query set provided in [11] which is considered as representative of a personal desktop where searches, updates and deletes are performed.

The second use-case is in the smart sensor context. Sensors can be smart meters deployed at home to enable new generation of energy services, home gateways capturing a variety of events issued by a growing number of smart appliances or car trackers registering our locations and driving habits to compute insurance fees and carbon tax [3]. In this case, documents are time windows and terms are events occurring during this time window. Top-k queries are useful for analytic tasks and executing them at the sensor side helps reducing evaluation time, energy consumption linked to data transmission costs and risk of private information leakage. We consider that in this use-case the indexing documents have a poorer content (hundreds to thousands of terms/event types per time window). We are not aware of publicly available datasets for this context and then generate a synthetic one.

The statistics of the desktop dataset and query set are given in Table 2. This dataset contains five representative types of personal files (i.e., email, html, pdf, doc and ppt). The desktop search is an important topic in the IR community, but real personal collections of desktop files cannot be published for evident privacy issues. Instead, the authors in [11] propose a method to generate pseudo desktop collections and show that such collections have the same properties as real collections. As recommended in [11], we preprocess the files in this collection by removing the stop words and stemming the remaining terms using the Krovetz stemmer. The obtained number of terms in the vocabulary is large, i.e., 337952. We also use a set of 837 queries prepared for this dataset and provided in [11]. Different, the synthetic dataset has a much smaller vocabulary, i.e., 10000 terms, and an average of 100 terms per document, chosen using a zipfian distribution with a skew of 0.7. For this dataset, we generated a query set of 1000 random queries with up to five terms per query. In the experiments, we analyze the cost of insertions and queries separately, one operation at a time. Indeed smart objects, contrary to central servers, rarely support parallel or multi-task processing. Moreover, the RAM consumption increases linearly with the number of operations executed in parallel, a serious constraint in our context. Due to space limitation, we focus in this paper on the results obtained with the desktop dataset and provide results issued from the synthetic dataset only when the differences are significant. The reader interested in the full set of measures can refer to a technical report [5]. This report also gives the pseudo-code of the measured algorithms and discusses results obtained using a third larger dataset (the ENRON email dataset available at : https://www.cs.cmu.edu/~enron/) which actually gives similar results as those presented here.

7.2 Index Maintenance

According to the algorithms presented earlier, the insertions and deletions of documents produce a sequence of index partitions which are subsequently merged in the *SSF*. Given the RAM_Bound of 5KB, we set the branching factor b (intermediate levels in the *SSF*) to ss8 to decrease the merge frequency, and the branching factor b' (last level in the *SSF*) to 3, to absorb faster the document deletions. The insertion or deletion of a single document is very efficient, since the document metadata is preliminary inserted in RAM. Also, given the small size of the RAM_Bound, flushing the RAM content into the level L_0 of the *SSF* is fast; it takes on average 6ms to write a partition in L_0 in all our experiments (see Table 3).

Table 3 shows also the *SSF* merge cost, which is periodically triggered (i.e., each time the number of flushed partitions in L_i reaches the branching factor b). The table presents the number of IOs for the flush and merge operations performed in the different *SSF* levels, and their execution times for the two tested SD cards, while inserting the 27K documents of our dataset and randomly deleting 10% of them. In our experiments, the deletions are uniformly distributed over the inserted documents and uniformly interleaved with the insertions. All these operations lead to an *SSF* with six levels. As expected, the merge time grows exponentially from L_0 to L_5, since the size of the partitions also increases by (nearly) a factor of b. It requires a few seconds to merge the partitions in the levels L_0 to L_3 and up to a few minutes in L_4 to L_5. The merge time is basically linear with the size of the merged partitions in the number of reads and writes. The merge time can vary especially in the first three levels of the *SSF*, depending on the term distribution in the indexed documents. However, the

partitions in L_3 contain most of the term dictionary and the variation of the merge time in the upper levels is less significant.

Table 3 indicates that the most costly merges are also the less frequent. Only 20 merges costing more than 15 seconds are triggered while inserting the complete set of documents and deleting 10% of them. However, blocking the index for a long duration may be problematic for some applications and justifies the non-blocking merge implementation presented in Section 5. Table 4 compares the maximum and average insertion/deletion time in the index between blocking and non-blocking merge implementations. The time is measured as the RAM flush time plus the merge time, if a merge is triggered (for the blocking merge) or is currently in progress (for the non-blocking merge). We observe that the cost of a blocking merge in L_5 can take up to 393 seconds with Silicon Power, while this cost is spread among the next 13844 insert/delete operations (each time the RAM is flushed) in the non-blocking case. This leads to an extremely large gap between the maximum and average insertion times in the blocking case. The insertion of the synthetic dataset also leads to an SSF with 6 levels [5]. The average insertion time is similar to the desktop dataset, e.g., with a non-blocking merge, the average cost is 0.29s for Kingston and 0.20s for Silicon Power.

Table 4. Blocking vs. non-blocking merge performance (sec.)

	Max.	Avg.
Blocking merge (Kingst./Silicon P.)	615/393	0.16/0.10
Non-blocking merge (Kingst./Silicon P.)	0.23/0.15	0.21/0.13

7.3 Index Search Performance

We evaluated the search performance of our index on our test board with the two SD cards, with both the blocking and non-blocking merge implementations. Due to the similarity of the results, we present hereafter only the results obtained with the Silicon Power card. Figure 7 shows the average query time for the 837 search queries of the query set as a function of the number of documents. The curves present the query cost before ("max" curve) and after ("min" curve) each merge occurring in the higher index levels, i.e., from level 3 to level 5. We can observe that locally, the query cost increases linearly with the number of partitions in the lower levels, and then decreases significantly after every merge operation. The large variations in the query cost correspond to the creation of a new partition in the fifth level of the *SSF*, while the intermediary peaks correspond to the creation of a partition in level 4 of the *SSF* (see the arrows in Figure 7).

Globally, the query time increases linearly with the number of indexed documents, but with a low factor. For example, after inserting 27K documents and deleting 10% of them, the average query execution time is only 0.18s (maximum of 0.35s) with the non-blocking merge implementation. The query times are lower with a blocking merge, i.e., an average execution time of 0.14s and a maximum of 0.28s. Indeed, in the non-blocking merge implementation, the number of lower level partitions can temporarily exceed *b*, so that more partitions have to be visited. In our setting, the increase is on average of about 25% and represents approximately 0.04s seconds. This appears to be a fair trade-off for applications that cannot accept unpredictable or unbounded update index latencies.

With the synthetic dataset [5], our index exhibits an average execution time of only 0.21s and a maximum of 0.42s. The slightly higher query times compared to the desktop dataset are explained by the larger index size (see Table 6) and by the smaller vocabulary (which reduces the average query selectivity).

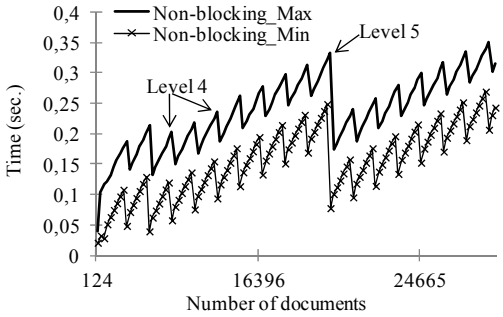


Figure 7. Query times with the non-blocking merge strategy

7.4 Impact of the Deletion Rate

Table 5 shows the average query performance for different deletion rates with Silicon Power storage (we obtained similar results with the Kingston storage). We considered two cases. First, we inserted the whole dataset while deleting a number of documents corresponding to the deletion rate (first line in Table 5). In this case, the higher the deletion rate is, the lower the query time is since a good part of the deleted documents (app. 50%) will be purged from the index and decrease the query processing time. In the second case, the total number of active documents in the index is the same (i.e., 13500) regardless the deletion rate (second line in Table 5). Hence, the higher the deletion rate is, the more documents we insert to compensate the deletions. In this case, a higher deletion rate leads to larger query times since part of the deletions are present in the index and have to be processed by the queries. However, the increase of the query times is relatively small compared to the case with no deletions, i.e., less than 16% for deletion rates up to 50%. Globally, the index is robust with the number of deletions in both cases.

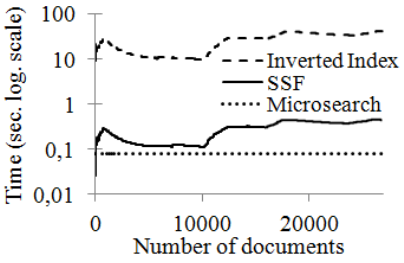
Table 5. Avg. query time (in sec.) with deletion (Silicon P.)

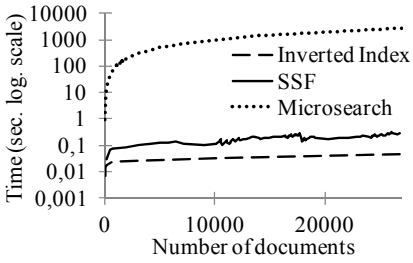
Deletion rate	0%	10%	30%	50%
Avg. query time (27k docs)	0.18	0.17	0.16	0.16
Avg. query time (13k docs)	0.12	0.13	0.13	0.14

Table 6. Index (*I.L*/*I.S*) size (MB) varying the deletion rate

Deletion rate	0%	10%	30%	50%
Desktop (SSF)	81/1.24	76/1.14	60/0.82	55/0.73
Desktop (classic)	81/0.4	73/0.4	57/0.4	40/0.4
Synthetic (SSF)	78 /0.97	74 /0.88	66 /0.78	58 /0.69
Synthetic (classic)	78 /0.13	70 /0.13	55 /0.13	40 /0.13

Table 6 shows the index size for the desktop and synthetic datasets after the insertion of all the documents in the collection and the uniform deletion of a certain percentage of the indexed documents. In each table cell, the first number indicates the cumulated size in MB of all the *I.L* parts of the *SSF* (i.e., the global size of the inverted lists), while the second number gives the cumulated size of all the *I.S* parts of the *SSF* (i.e., the global size of the search structures). Also, Table 6 gives for each dataset





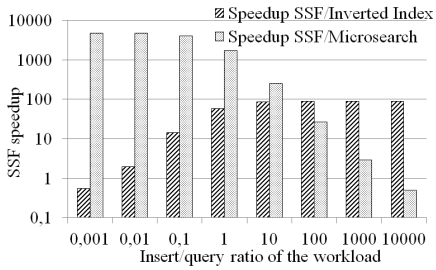


Fig. 8. Insert time comparison (Silicon P.) **Fig. 9. Query time comparison(Silicon P.)** **Fig. 10. Overall performance comparison**

and deletion rate the index size of the classical inverted index used as reference (i.e., a single standard B^+-Tree built with no consideration for the Flash cost). Without deletions, the SSF index size is comparable to the classical inverted index size. Indeed, the size of the *I.L* part (inverted lists) is similar in both indexes and the overhead introduced by *SSF* for the *I.S* part (each partition having its own search structure) is negligible since the search structure represents less than 1% of the global index size. As the deletion rate augments, the difference between *SSF* and the classical index increases. The explanation is that the deleted documents are reinserted in SSF, which temporarily increases the index size, until a merge is triggered and absorbs part of the deleted documents. Typically, we observed that about 45% to 55% of deletions are not absorbed after a high number of document insertions and deletions. This makes the SSF index size to be at most 40% larger than the inverted index size.

7.5 Comparison with the State-of-the-Art

This section compares our search engine with state of the art indexing methods. We choose the classical inverted index (see figure 2) to represent the query-optimized index family (although it has not been designed with embedded constraints in mind) and Microsearch [17] to represent the insert-optimized index family. Note that the other embedded search engines presented in Section 2.3 rely on similar index structures with Microsearch. We used the same test conditions as above, i.e., a RAM_Bound equals to 5KB. The insertions in the classical inverted index are first buffered in RAM until the RAM_Bound is reached, then applied in batch to the index structure in Flash, generating random writes at this time. To be able to evaluate the queries under the RAM constraint, the inverted lists have to be maintained sorted on document ids, which permits applying a linear pipeline query processing similar to the *SSF*. In the case of Microsearch, we used a hash function with 8 buckets, since this value leads to the most balanced query-insert performance given the 5KB of RAM. Besides, we only considered data insertions and queries in the tests below, since Microsearch does not support deletions.

Insertion performance. Figure 8 shows the average insertion time for the three methods (i.e., *SSF*, Microsearch and the classical Inverted Index) while inserting the 27,000 documents in the desktop dataset. Microsearch and *SSF* have similar insert performance. On average, a document insertion in Microsearch takes about 0.08s and 0.33s in SSF (with Silicon Power). The insertion time in the classical Inverted Index is at least two orders of magnitude higher (30s with Silicon Power) because of the costly random writes in Flash memory, clearly dismissing this method in the context of smart objects. For the synthetic dataset, the average insertion times are 0.02s, 0.07s and 15s for Microsearch, *SSF* and the inverted index respectively (with Silicon Power). The higher insertion cost of the *SSF* compared to Microsearch is generated by the *SSF* merges. But this gap seems acceptable for most of the applications and is outweighed by the query performance and scalability benefit of the *SSF* (see next).

Query performance. Figure 9 shows the query execution time for the three methods in function of the number of indexed documents. Unsurprisingly, the Inverted Index reaches the best query performance. SSF appears as a good challenger (0.18s on the average to process a query compared to 0.07s with the Inverted Index, with Silicon Power), this difference being explained by the fragmentation of the *SSF* index structure. Microsearch exhibits the worst query performance and can definitely not scale to a large number of documents. On average, Microsearch takes 880 seconds (14 minutes) to process a query. Even for a low number of documents, SSF outperforms Microsearch. The first reason is that Microsearch merges many terms in the same inverted lists, so that large part of the index has to be scanned. Second, Microsearch requires two passes over the inverted lists, one to compute the global F_t value of the term and a second one to compute the *tf-idf* score of the documents containing the term. For the synthetic dataset, the average query times are 0.05s, 0.2s and 355s for the inverted index, SSF and Microsearch respectively (with Silicon Power).

Overall performance. Figure 10 shows the speedup of SSF (i.e., the ratio between the throughput of SSF and of the competitors) with Silicon Power for workloads containing insertions and queries (with the pseudo-desktop collection) in different ratios. We obtained similar results with the other dataset or storage card. In most cases, SSF has (much) better throughput with both insert- and query-oriented workloads, while being the sole versatile method. Practically, SSF will be the preferred index method unless the expected workload contains in an overwhelming proportion either insertions or queries.

Other concerns. The pros and cons of the SSF approach compared to its competitors originates from the specific way the index grows (by partitioning and merging) and the deletes are managed (by reinserting deleted documents and absorbing deletes during merges). While performances have been extensively studied and compared above, the specificities of SSF may impact other aspects of an indexing structure, more difficult to weight. First, partitioning introduces some variability in the query cost (see the stairway-like curves in Figures 8 and 9) which could be prejudicial for real-time applications and could perturb a query optimizer. Note however that the occurrences of stairs are fully predictable, though it complexifies the optimization process. Second, the way deletes are processed lead to an increase of the

index size and consequently, of the query cost. Nevertheless, this negative effect is limited by the merge operations that permit to purge some of the deleted documents (see Table 6). Moreover, we do not see how to manage deletions differently without violating any of our design principles. Finally, we do not consider the problem of concurrent accesses in *SSF*, i.e., multiple processes that query/update the index at the same time. While this seems not a primary concern today, this problem may deserve a deeper interest as smart objects become more powerful.

8. ACKNOWLEDGMENTS

This work is partially funded by project ANR-11-INSE-0005 KISS.

9. CONCLUSION

This paper presents the design of an embedded search engine for smart objects equipped with extremely low RAM and large Flash storage capacity. This work contributes to the current trend to endow smart objects with more and more powerful data management techniques. Our proposal is founded on three design principles, which are combined to produce an embedded search engine reconciling high insert/delete rate and query scalability for very large datasets. By satisfying a RAM_Bound agreement, our search engine can accommodate a wide population of smart objects, including those having only a few KBs of RAM. Satisfying this agreement is also a mean to fulfill co-design perspectives, i.e., calibrating a new hardware platform with the hardware resources strictly necessary to meet a given performance requirement. The proposed search engine has been implemented on a hardware platform representative of smart objects and the experimental evaluation validates its efficiency and scalability. We feel that our three design principles may have a wider applicability and could pave the way to the definition of other embedded indexing techniques. It is part of our future work to try to validate this assumption.

10. REFERENCES

[1] Aggarwal, C. C. Ashish, N. and Sheth, A. The internet of things: A survey from the data-centric perspective. *Managing and mining sensor data*, Springer, 2013.

[2] Agrawal, D., Ganesan, D., Sitaraman, R., Diao, Y. and Singh, S. Lazy-adaptive tree: An optimized index structure for flash devices. In *PVLDB 2*, 1 (2009), 361-372.

[3] Anciaux, N., Bonnet, P., Bouganim, L., Nguyen, B., Sandu Popa, I. and Pucheral, P. Trusted cells: A sea change for personal data services. In *Conference on Innovative Data Systems Research* (*CIDR'13*), 2013.

[4] Anciaux, N., Bouganim, L., Pucheral, P., Guo, Y., Folgoc, L. L. and Yin, S. MILo-DB: a personal, secure and portable database machine. *Distribut. and Parallel Databases, 32*, *1* (2014), 37-63.

[5] Anciaux, N., Lallali, S., Sandu Popa, I., Pucheral, P. A Scalable Search Engine for Mass Storage Smart Objects. PRISM Technical Report. http://www.prism.uvsq.fr/~isap/files/RT.pdf

[6] Bjorling, M., Bonnet, P., Bouganim, L. and Jonsson, B. T. uflip: Understanding the energy consumption of flash devices. *IEEE Data Eng. Bull.*, *33*, *4* (2010), 48–54.

[7] Debnath, B., Sengupta, S. and Li, J. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *ACM SIGMOD*, 2011, 25–36.

[8] Diao, Y., Ganesan, D., Mathur, G. and Shenoy, P. J. Rethinking data management for storage-centric sensor networks. In *Conference on Innovative Data Systems Research* (*CIDR'07*), 2007.

[9] Huang, Y.-M. and Lai, Y.-X. Distributed energy management system within residential sensor-based heterogeneous network structure. In *Wireless Sensor Networks and Ecological Monitoring, 3* (2013), 35–60.

[10] Jermaine, C., Omiecinski, E. and Yee, W.G. The partitioned exponential file for database storage management. *The VLDB Journal 16*, *4* (2007), 417-437.

[11] Kim, J. Y. and Croft, W. B. Retrieval Experiments using Pseudo-Desktop Collections. In *ACM CIKM*', 2009, 1297-1306.

[12] Lallali, S., Anciaux, N., Sandu Popa, I. and Pucheral, P. A Secure Search Engine for the Personal Cloud. In *Sigmod*, 2015. http://dx.doi.org/10.1145/2723372.2735376

[13] Li, Y., He, B., Yang, R. J., Luo, Q. and Yi, K. Tree Indexing on Solid State Drives. In *PVLDB 3,* 1-2 (2010), 1195-1206.

[14] O'Neil, P.E., Cheng, E., Gawlick, D. and O'Neil, E.J. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica, 33, 4* (1996).

[15] Sears, R. and Ramakrishnan, R. bLSM: a general purpose log structured merge tree. In *ACM SIGMOD*, 2012, 217-228.

[16] Tan, C. C., Sheng, B., Wang, H. and Li, Q. Microsearch: When search engines meet small devices. In *Pervasive Computing*, 2008, 93-110.

[17] Tan, C. C., Sheng, B., Wang, H. and Li, Q. Microsearch: A search engine for embedded devices used in pervasive computing. *ACM Transactions on Embedded Computing Systems, 9*, *4* (2010).

[18] To, Q.-C., Nguyen, B. and Pucheral, P. Privacy-Preserving Query Execution using a Decentralized Architecture and Tamper Resistant Hardware. In *EDBT,* 2014, 487-498.

[19] Tsiftes, N. and Dunkels, A. A database in every sensor. In *ACM Embedded Networked Sensor Systems* (SenSys'11), 2011, 316-332.

[20] Wang, H., Tan, C. C. and Li, Q. Snoogle: A search engine for pervasive environments. In *IEEE Transactions on Parallel and Distributed Systems, 21*, *8* (2010), 1188–1202.

[21] Wu, C.-H., Kuo, T.-W. and Chang, L.-P. An efficient b-tree layer implementation for flash-memory storage systems. *ACM Trans. on Embedded Computing Systems, 6*, *3* (2007).

[22] Yan, T., Ganesan, D. and Manmatha, R. Distributed image search in camera sensor networks. In *ACM Embedded Network Sensor Systems* (SenSys'08), 2008, 155–168.

[23] Yap, K.-K., Srinivasan, V. and Motani, M. Max: Wide area human-centric search of the physical world. *ACM Transactions on Sensor Networks*, *4*, *4* (2008), 1-34.

[24] Zobel, J. and Moffat, A. Inverted files for text search engines. *ACM Computing Surveys*, *38*, *2* (2006).