Supporting secure keyword search in the personal cloud

Saliha Lallali, Nicolas Anciaux, Iulian Sandu-Popa, Philippe Pucheral

▶ **To cite this version:**

Saliha Lallali, Nicolas Anciaux, Iulian Sandu-Popa, Philippe Pucheral. Supporting secure keyword search in the personal cloud. Information Systems, 2017, 72, pp.1 - 26. 10.1016/j.is.2017.09.003 . hal-01660599

HAL Id: hal-01660599
https://inria.hal.science/hal-01660599v1

Submitted on 11 Dec 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Supporting Secure Keyword Search in the Personal Cloud

Saliha Lallali[a], Nicolas Anciaux[a,b], Iulian Sandu Popa[b,a], Philippe Pucheral[b,a]

[a] *INRIA Saclay-Ile-de-France, Université Paris-Saclay, 1 Rue H. d'Estienne d'Orves, 91120 Palaiseau, France*
[b] *DAVID Lab, University of Versailles Saint-Quentin-en-Yvelines, Université Paris-Saclay, 45 avenue des Etats-Unis, 78035, Versailles Cedex, France*

Abstract

The Personal Cloud paradigm has emerged as a solution that allows individuals to manage under their control the collection, usage and sharing of their data. However, by regaining the full control over their data, the users also inherit the burden of protecting it against all forms of attacks and abusive usages. The Secure Personal Cloud architecture relieves the individual from this security task by employing a secure token (i.e., a tamper-resistant hardware device) to control all the sensitive information (e.g., encryption keys, metadata, indexes) and operations (e.g., authentication, data encryption/decryption, access control, and query processing). However, secure tokens are usually equipped with extremely low RAM but have significant Flash storage capacity (Gigabytes), which raises important barriers for embedded data management. This paper presents a new embedded search engine specifically designed for secure tokens, which applies to the important use-case of managing and securing documents in the Personal Cloud context. Conventional search engines privilege either insertion or query scalability but cannot meet both requirements at the same time. Moreover, very few solutions support data 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 matching the hardware constraints of secure tokens and reconciling high insert/delete/update rate and query scalability. Our experimental results, obtained with a prototype running on a representative hardware platform, demonstrate the scalability of the approach on large datasets and its superiority compared to state of the art methods. Finally, we also discuss the integration of our solution in another important real use-case related to performing information retrieval in smart objects.

Keywords: Embedded search engine,indexing techniques, conditional top-k queries, flash memory, secure token, secure personal cloud, smart object

1. Introduction

We are witnessing an exponential accumulation of personal data on central servers: data automatically gathered by administrations, companies and web sites but also data produced by the individuals themselves and deliberately stored in the cloud for convenience (e.g., photos, agendas, raw data produced by smart appliances and quantified-self devices). Unfortunately, there are many examples of privacy violations arising from abusive use or attacks, and even the most secured servers are not spared.

The Personal Cloud paradigm has recently emerged as a way to allow individuals to manage under their control the collection, usage and sharing of their data, as requested by the World Economic Forum [1]. Initiatives like Blue Button and Green Button in the US, MiData in Great Britain and MesInfos in France bring about this paradigm by returning personal data retained by companies and administrations to individuals. Personal cloud platforms also arise in the market place (e.g., Cozy Cloud, Own Cloud, SeaFile and Younity to cite [2] only a few). This user-centric vision illustrates the gravity shift of information management from organizations to individuals. However, at the time individuals recover their sovereignty of their data, they also inherit the burden of organizing this personal data space and more importantly of protecting it against all forms of attacks and abusive usages, a responsibility that they cannot endorse.

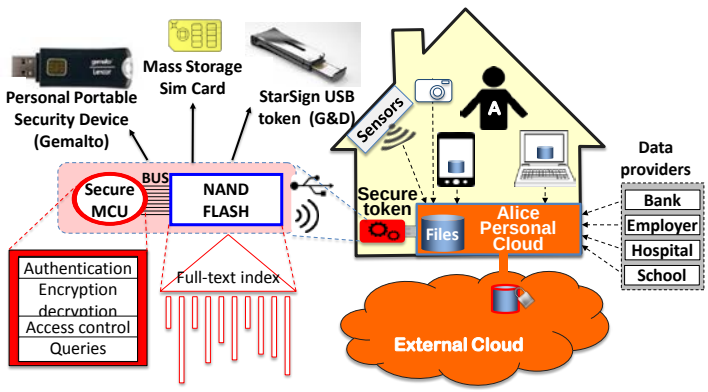

Fig. 1. Example of a Secure Personal Cloud platform

The Secure Personal Cloud paradigm that we proposed in [7] relieves the individual from this security task. The corresponding architecture, illustrated in Figure 1, combines a traditional personal cloud server (e.g., running on a plug computer or an internet gateway at home) and a *secure token* (i.e., a tamper-resistant hardware device). Heterogeneous data issued by external sources and by personal appliances are all transformed into documents. Document metadata (keywords extracted from the file content, date, type, authors, tags set by the user herself, etc.) is extracted at insertion time, stored in the secure token and indexed so that the secure token can act as a privacy preserving Google Desktop or Spotlight for the user's dataspace. Documents themselves are encrypted by the secure token before being stored in the Personal Cloud, locally or remotely. Thus, the secure token plays the role of a gatekeeper for the whole Personal Cloud by managing all

[1] The World Economic Forum. Rethinking Personal Data: Strengthening Trust. May 2012.
[2] OwnCloud: https://owncloud.org/ ; CozyCloud: https://www.cozycloud.cc ; SeaFile: http://www.seefile.com/ ; Younity: http://getyounity.com/

the sensitive information (e.g., encryption keys, metadata, indexes) and operations (e.g., authentication, data encryption/decryption, access control, and query processing) [22, 23].

In the context of the Personal Cloud, embedding a full-text search engine in a secure token will allow a user to securely search through her file collection in a simple way (i.e., using keywords), without exposing any metadata to the outside world. A file can be any form of document, mail, picture, music or video file, etc., that is 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. Designing such embedded search engine is however very challenging. Indeed, data storage in secure tokens is usually provided by large capacity removable SD or μSD cards or by soldered raw Flash chips while computing power is provided by microcontrollers (MCU) equipped with tiny RAM (tens of KB). This conjunction of hardware constraints raises critical issues deeply discussed in [5]. Typically, NAND Flash badly adapts to random fine-grain updates while state-of-the-art indexing techniques either consume a lot of RAM or produce a large quantity of random fine-grain updates.

We propose in this paper an efficient and scalable search engine adapted to the highly constrained architecture of secure tokens. Specifically, we make the following contributions[3]:

- We define two mandatory properties, namely *Bounded RAM agreement* (capturing the hardware constraints) and *Full scalability* (capturing the performance requirements) and then introduce three design principles, namely *Write-Once Partitioning*, *Linear Pipelining* and *Background Linear Merging*, to devise an inverted index complying with these properties.
- 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 show that our solution can be extended towards a *conditional top-k* query engine to support flexible and secure searches in a Personal Cloud context.
- Finally, we validate our design in two complementary ways. First, we do a precise analysis of the RAM consumption of each algorithm and demonstrate that each satisfies the Bounded RAM agreement. Second, we conducted a comprehensive set of experiments on a real representative secure token platform, using three real and synthetic datasets, and show that query and insertion/deletion/update performance can be met together demonstrating Full scalability compliance.

It is worth noticing that the hardware architecture of secure tokens is very similar to the hardware architecture of smart objects, which also consists in an MCU connected to a NAND Flash storage. Given the capacity of smart objects to acquire, store and process data, new services have emerged. Camera sensors tag photographs and provide search capabilities to retrieve them [40]. Smart objects maintain the description of their surrounding [41], e.g., shops like bookstores can be queried directly by customers in search of a certain product enabling the Internet of Things [1]. Smart meters record energy consumption events and GPS devices track locations and moves. This explains the growing interest for transposing traditional data management functionalities directly into smart objects. The embedded search engine proposed in this paper can be equally

[3] This paper is an extended version of [6]. The new material covers three significant contributions. First, we provide all algorithms underlying our search engine, some of them being non trivial, and perform a thorough analysis of their RAM consumption to demonstrate the compliance of our design with the Bounded RAM agreement. Second, we extended the performance measurements performed in [6] with a third real dataset (ENRON), representative of the personal Cloud context and more generally of any context manipulating rich documents with random updates. We also extended the type of situations measured in order to demonstrate the compliance of our design with the Full scalability property. Taken together, Sections 8 and 9 validate the approach in the most comprehensive and complementary way. Third, Section 7 proposes an extension of our work to support *conditional top-k* queries in the personal Cloud context. Finally, note that an operational prototype of the complete design has been developed and different facets of this prototype have been demonstrated [22, 23].

employed in the context of smart objects to search relevant objects in their surroundings based on their description, search pictures by using tags or perform analytic tasks (i.e., top-k queries) over sets of events (i.e., terms) captured during time windows (i.e., files).

The rest of the paper is organized as follows. Section 2 details the search engine requirements, the secure tokens' 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. Section 7 introduces the need for conditional top-k queries and discusses the extension of our search engine to match such a requirement. Then, we validate our design with respect to the Bounded RAM agreement (Section 8) and Full scalability (Section 9) properties. Finally, Section 10 concludes.

2. Problem Statement

This section describes the main requirements of a full-text search engine, the hardware constraints of secure tokens, and reviews the literature addressing the problem of implementing a search engine under these constraints. Then, in the light of the existing works and their shortcomings, we precisely state the problem addressed in this paper.

2.1. Search Engine Requirements

As in [33], we consider that the search engine of interest in this paper has similar functionality as a Google Desktop embedded in secure tokens. 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 since decades (see [42] 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 [40]. For a query $Q=\{t\}$, the *tf-idf* score of each indexed document d containing at least a query term can be computed as follows:

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

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.

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 [42]: (*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.

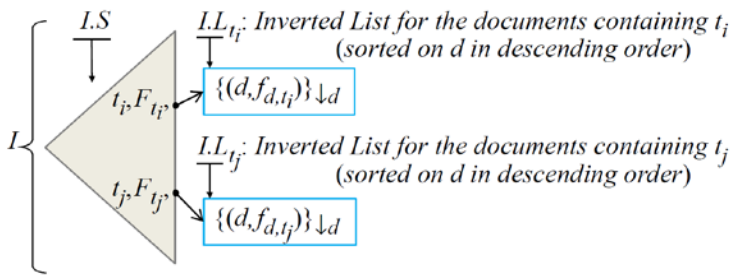

Fig. 2. Typical inverted index structure

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.

2.2. Secure Tokens' Hardware Constraints

Whatever their form factor and usage, secure tokens share strong commonalities in terms of data management architecture. Indeed, a large NAND Flash storage is used to persistently store the data and the 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 memory.

The MCUs embedded in secure tokens 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 base unit for a read and a write operation is the sector (usually 512 bytes) with raw NAND Flash chips or the page (usually 2 Kbytes or four sectors) with SD/micro-SD cards in which the access to the Flash memory is managed by a Flash Translation Layer (FTL). The sectors/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 sectors/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 (see Section 9.1) corroborate the ones published in [12] indicating that random writes are (much) more costly than sequential writes on SD cards. While high-end SSDs use large on-device RAM (e.g., 512MBytes) to reorder random writes and optimize their performance, secure tokens equipped with very scarce RAM and basic NAND Flash storage cannot hide NAND Flash constraints and as such are exposed to

large performance degradation. Hence, in the embedded context, random writes in Flash storage must be proscribed.

We also mention that in case of accessing the Flash through an FTL (e.g., with an SD card), the access can be done at a granularity smaller than the base unit (e.g., reading/writing at a sector granularity instead of a page granularity) with the inconvenient of a reduced read/write throughput of the device. However, the sequential/random write ratio remains practically the same with the ratio at the base granularity (see Table 1). The advantage in this case is the reduction of the amount of RAM memory required to process the data read from or written to the Flash storage, which is a major benefit in the context of secure tokens given their tiny RAM memory. In this work, we employ this approach since our secure tokens use a micro-SD card storage, but we prefer to access it at a sector granularity (i.e., 512 bytes) to reduce the RAM consumption of the proposed method. For simplicity, use the terms *page* and *sector* interchangeably in the rest of the paper to denote a data unit of 512 bytes in Flash or in RAM.

Finally, it is worth observing that given the strong similarity between the hardware architecture of secure tokens and of smart objects, we can consider that the secure tokens represent a specific instance of smart objects, i.e., smart objects having a tamper-resistant MCU. Hence, in this paper we use the terms *secure token* and *smart object* according to this observation.

2.3. State-of-the-Art Solutions

Data management embedded in secure tokens [4, 34] or more generally 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 [11], strengthening the idea that smart objects must now be considered as first-class data sources. For instance, simple query evaluation facilities have been recently proposed for sensor nodes equipped with large Flash memory [14, 16] 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 [5, 35]. More complex treatments such as facial recognition and the related indexing techniques have been investigated also [15]. However, several works [5, 25, 35] 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 and 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 [32, 33, 37, 38, 40]. These works rely on a similar design of the embedded inverted index as proposed in [33]. 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 first Flash page of each bucket is kept in RAM. The number of buckets is kept small, such that (*i*) the large dictionary of terms (usually tens of MB) is replaced by a small hash table stored in RAM, and (*ii*) 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 unavoidably 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 [40] 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 [33]. 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.

Key-value pair indexing in NAND Flash. In the key-value store context, SkimpyStash [13], LogBase [36], SILT [26] and Hyder [10] propose Flash-aware structures to store and query key-value pairs. Hyder [10] proposes a multiversion key-value database stored in Flash and shared over the network. It makes use of a single binary balanced tree index in Flash to find any version of any tuple corresponding to a given key. The binary tree is not updated in place, the path from the inserted or updated node being rewritten up to the root. Unfortunately, this technique cannot be used to implement full-text searches where each term may appear in many documents (i.e., binary trees are not adequate to index non-unique keys). SkimpyStash [13], LogBase [36] and SILT [26] organize key-value pairs in a log structure to exploit sequential writes, but require some form of in-memory (RAM) indexing with a size proportional to the database size. Thus, the memory consumption may easily exceed the RAM size of an MCU (i.e., these methods require at least 1B per indexed record).

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 [39]. Many recent proposals [2, 24, 39] tackle this problem. 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, the gain on random writes definitely vanishes.

Partitioned indexes. In another line of work, partitioned indexes have been extensively employed especially to improve the storage performance in environments with insert-intensive workloads and concurrent queries on magnetic disks. A prominent example is the LSM-tree (i.e., the Log-Structured Merge-tree) [30] and its many variants (e.g., the Stepped Merge Method [17], the Y-tree [18], the Partitioned Exponential file [19], and the bLSM-tree [31] 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 of such an approach 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, highly increases the throughput. If the indexed dataset becomes too large, the index disk component can be divided into several disk components of exponentially increasing size to reduce the write amplification of merges. Many works have proposed optimized versions of the LSM-tree. For instance, bLSM [31] fixes several limitations of the LSM-tree. Among the improvements, the main contribution is an advanced merge scheduler that bounds the index write latency without impacting its throughput. Also, the FD-tree [24]

proposes a similar structure with the LSM-tree to optimize the data indexing on SSDs. Furthermore, the storage systems of the major web service provider, e.g., Google's Bigtable and Facebook's Cassandra, employ a similar partitioning approach to implement key-value stores. The idea is to buffer large amounts of updates in RAM and then flush them in block on disk as a new partition. Periodically, the small partitions are merged into a large partition.

The proposed search engine shares the general idea of index partitioning and merging with the above mentioned works. However, the similarity stops at the general level since the specific hardware constrains and type of queries in our context cannot be satisfied by the existing solutions. In particular, the small amount of RAM requires frequent flushes of the buffered updates. This leads to a specific organization of the partitions and merge scheduling in our structure. The type of query in our context (i.e., top-k keyword search) represents another major difference with the existing partitioning methods that only consider the classical key-value search. To be able to evaluate full text search queries in the presence of deletions and limited amount of RAM, our search engine proposes a novel index organization with a specific query processing.

In general, designing access methods is often a matter of tradeoff between minimizing read times, update cost and memory/storage overhead as recently observed in [8]. Given the specific architecture of secure tokens, this tradeoff translates to a tension between memory and Flash storage in the embedded context [3]. Specifically, 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*: the proposed engine 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 secure tokens. 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 secure token 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 data sets. 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 $<I_0, I_1,\ldots, I_p>$, 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 $<I_0, I_1,\ldots, I_p>$ 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 $<I_0, I_1,\ldots, I_p>$ 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 $<I_0, I_1,\ldots, I_p>$ 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 $<I_0, I_1,\ldots, I_p>$ 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 in a way compliant 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 in the next sections. 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 $<I_0, I_1,\ldots, I_p>$ 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 $<I_0, I_1,\ldots, I_p>$.

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.

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 usually materialized in the dictionary part of the index ($\{t, F_t\} \subset I.S$), as shown in Figure 2. When I is split in $<I_0, I_1,\ldots, I_p>$, 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 the 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 principle 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 must be decremented by 1. Hence, the correct global value of F_t can easily be computed without physically accessing the inverted lists.

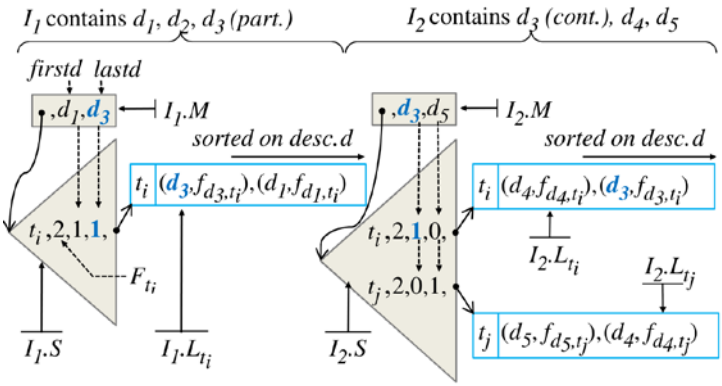

Fig. 3. Consecutive index partitions with overlapping documents

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.

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 $t \in Q$ [42]. 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.

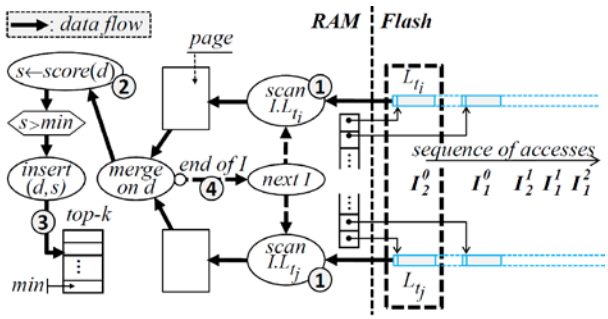

Fig. 4. Linear Pipeline computation of Q over terms t_i and t_j.

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.

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.

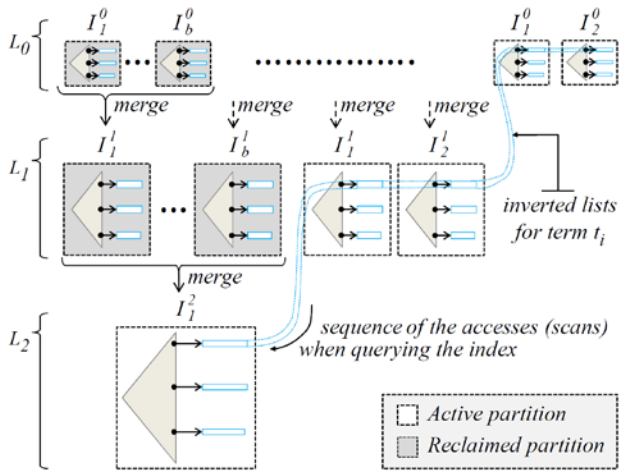

Fig. 5. The Scalable and Sequential Flash structure

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. In this way, 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. In this section we extend 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:

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

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.

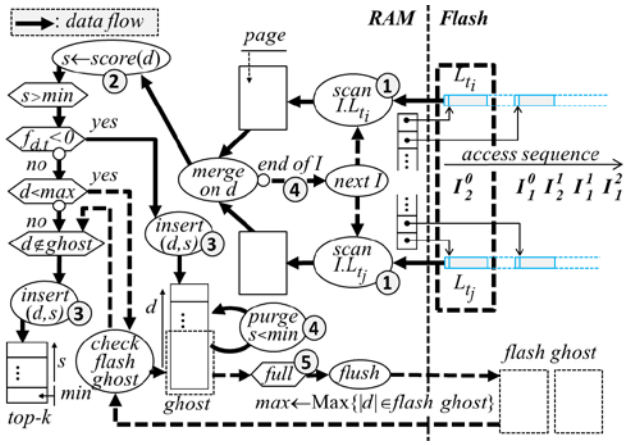

Fig. 6. Linear pipeline computation of Q in the presence of deletions

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:

$$\delta \cdot k \cdot \sum_{i=1}^{n} 1/i$$

This value 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).

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. Towards Conditional Top-k Queries

In this section, we discuss the extension of the proposed search engine to support *conditional top-k queries*, i.e., top-k queries combined with conditions expressed over documents' metadata. Specifically, we present two major use-cases in the context of the Personal Cloud, i.e., keyword search combined with file metadata and tag-based access control. Then, we show that these two use-cases require conditional top-k queries and explain how the SSF data structure and algorithms can be extended to support it in a natural way.

7.1. The Need for Conditional Top-k Queries in the Personal Cloud

Combining keyword search and file metadata search. When a new file is added to the user's collection, both the keywords extracted from the file content (if any) and the file metadata (e.g., creation date, filename, file type and extension, tags set by the user herself, etc.) can be indexed by the search engine and then used to evaluate the score of the documents relevant for the user queries. Similar to the classical inverted index, the base implementation of our search engine (see Section 4) does not make any distinction between the terms extracted from the file content and the file metadata terms. Hence, the score of a document is computed using the classical tf-idf formula (see Section 2.1) regardless if the query terms are content keywords or metadata terms in the document.

Nonetheless, from a semantic point of view, it makes sense to separate the *content terms* from the *metadata terms* in the query evaluation like in the existing file search engine implementations (e.g., Google desktop or Spotlight). The idea is that the query terms are matched only with the content terms of the indexed documents, while users can specify additional constraints regarding the document metadata. For instance, a user can combine the query terms "research meeting Bordeaux" with the constraints "file type = pptx or (file type = mail and sender = Bob)". In this case, the query results will only consist of documents having the extension "pptx" or documents of type "mail" sent by "Bob", which contain at least one term among "research meeting Bordeaux" and ranked based on the weight of these three words in the documents. In general, the metadata search consists in one or several metadata terms that are combined by AND/OR to form a logical expression, i.e., disjunctions of conjunctions of metadata terms. Formally, a metadata search rule is a *logical expression*: $R_{metadata} = \bigvee_j (\bigwedge_i t_{i,j}^{pred})$, where each $t_{i,j}^{pred}$ is a metadata term predicate which for a given document d is evaluated true only if the metadata term $t_{i,j}$ is associated with d. For each document matching the query content terms, the logical expression on the metadata terms is evaluated and the document is considered in the query results only if the metadata expression is evaluated true.

Tag-based access control. The owner of a personal cloud might want to share some of her documents with other users or applications. To this end, she needs to customize the sharing of her documents by adding a personalized access control (AC) policy for each user/application accessing her personal cloud. She also wants to be sure that the personal cloud is able to securely enforce the defined policies. The latter is achieved by integrating the access control engine within the secure token together with the search engine as depicted in Figure 1. The definition of the AC policies depends on the employed AC model. Access control is a well-established topic in the databases field. However, traditional AC models like MAC (Mandatory Access Control), DAC (Discretionary Access Control) or R-BAC (Role-Based Access Control) are less suitable for the Personal Cloud paradigm than Tag-Based Access Control (TBAC) models [9, 21, 27, 28, 29]. Several works consider TBAC models, due to their simplicity, for non-technical computer users that require to define access control policies over their personal data. For example, [9] proposes a semantic access control for online photo albums based on descriptive tags found in social networks such as Flickr, Delicious and linked data. [29] goes in the same direction and shows that tags related to data organization can be easily used by home users to express coherent policies for access control. In [21, 27, 28], access control models to share files based on the TBAC model are proposed and in vivo studies involving non-specialist users show the usability of such models. A TBAC model is thus considered as a good candidate for the personal cloud context.

TBAC can be easily integrated with our inverted index if we consider that (1) an appropriate set of *access control terms* can be inserted to tag the documents at insertion, and (2) these access terms can be used at query processing time to evaluate *logical expressions* leading to discard or not each document from the query results. For illustration purpose, we define hereafter a simple TBAC model. Let D be the set documents referred by the inverted index. Each document is associated with set of terms (regular terms extracted from the documents content and access terms derived from the document content or metadata). All terms are extracted automatically at insertion time. For each document $d \in D$, let T_d be the set of access terms that was assigned to d at insertion time. Let U be set of all users/applications that can access the Personal Cloud. Any user/application has to be authenticated before gaining access to the Personal Cloud. We assume that by default everything is private and that applications are allowed to query (i.e., read) only the subset of the documents which match the access control rule defined for that application. For each user $u \in U$, the Cloud owner can define an access control rule R_u as disjunctions of conjunctions of access control terms, (i.e., $R_u = \bigvee_j (\bigwedge_i t_{i,j}^{pred})$, where each $t_{i,j}^{pred}$ is a term predicate which for a given document d is evaluated true only if the term $t_{i,j}$ appears in the access term list T_d. Therefore, at query time, the embedded access control engine evaluates for any document d in the Personal Cloud if u has the right to access d. This is realized internally by the Boolean function *Filter(u, d)*, which returns true iff R_u is true on d.

Let us consider a simple example. Bob is a friend of Alice with whom she shares the passion for country music. Bob asks Alice to share with him her country music collection. A collection rule has been settled such that any music file inserted in the personal cloud of Alice is associated with the access terms "music" and "<music_style>" where the value of music style is derived from the domain of music styles (e.g., "country", "variety", "jazz", …). Alice may define a permission rule for user Bob in her Personal Cloud: “Bob: music ∧ country”. Thus, Bob can query all the documents in the Alice’s server that contain both the access term “music” and “country”. Alice herself needs to access her email archive over the last two years from her smartphone, but does not want that the rest of her personal data space to be accessible from her smartphone. Therefore, assuming a collection rule associates any email file with the access terms "email" and "<date>", Alice may define the rule “Alice_smartphone: (email ∧ 2014) ∨ (email ∧ 2015)”.

Note that the logical expression used to define the AC policies has the same format (i.e., disjunctions of conjunctions) as the document metadata logical expression that can be used in the scenario that combines keyword search with file metadata search describe above. Hence, these two scenarios require the same

modifications to be integrated in the proposed search engine. The transformed keyword search queries are called *conditional top-k queries* in the sequel.

7.2. Search Engine Adaptation to Conditional Top-k Queries

To support conditional top-k queries, the modifications in our search engine have to be done at three levels: (i) update the top-k scoring function; (ii) adjust the insertion process and the inverted index structure; (iii) modify accordingly the query evaluation process. We detail here below these three modifications.

Modification of the top-k scoring function. Conditional top-k queries require to evaluate a logical expression for all the candidate documents for the query results, i.e., documents having a tf-idf score high enough to enter the top-k results. Executing a query *Q* in this context is equivalent to evaluate the following function:

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

Compared with the initial top-k scoring function defined in Section 4, the only difference is the appearance of the *Filter*() element. *Filter* is a function that takes as input a logical expression *le* (e.g., a TBAC policy or a file metadata expression) and the list of metadata terms T_d of a document. The function returns 1 if *le* is true on T_d and 0 otherwise. In other words, the tf-idf score of the documents whose metadata verify the logical expression remains unchanged, whereas the tf-idf score is set to 0 for the rest of the documents, which are thus eliminated from the query results.

Impact on the data insertion and the inverted index structure. The conditional top-k functionality requires separating semantically the content terms from the metadata terms of a document in the index structure. We use hereafter the notion of metadata term to refer to any kind of metadata (i.e., access control terms, file metadata terms or other types of metadata). To obtain this separation, we prefix all the metadata terms with a reserved tag, which will lexically distinguish the two types of terms without having any repercussion on the search structure (i.e., *I.S*) of the inverted index in a partition. Therefore, the search of a content term is done normally, while to search for a metadata term one has to prefix the term value with the metadata tag. Finally, the inverted lists of metadata terms have the same structure (i.e., pairs (d, $f_{d,t}$)) and organization (i.e., sorted on descendant value of d) as the inverted lists of content terms. The only difference is that the $f_{d,t}$ can only have two values, i.e., +1 for a document insertion and -1 for a document deletion.

Impact on the query evaluation. The enriched query execution is depicted in Figure 7. The query process starts as before and considers only the content terms of the query. Whenever the score of a document d is within the k best current scores, the identifier of d is searched in the inverted lists of the metadata terms involved in the conditional expression. If *Filter* evaluates this condition to *false*, the document d is discarded. Otherwise, d it is kept and inserted in the top_k buffer. The search within the inverted lists of the access terms *requires only one additional RAM page*. This page is used to search the occurrence of d in the inverted of each metadata term of the logical expression. To avoid reading the complete inverted list of a metadata term, the search of document d is performed using a dichotomy. Also, to evaluate the conjunctive parts of the logical expression, metadata terms are accessed from the less frequent term (with the smallest inverted list) to the most frequent term, to potentially stop the search as soon as d is absent from a list. The overhead in terms of memory consumption for the evaluation of the logical expression is kept minimal (i.e., one RAM page) and

the execution time overhead is low since: (i) only the documents entering within the *k* best current scores trigger *Filter*; (ii) deleted documents can be inserted in the *Ghost* list without checking the logical expression to minimize IO cost; (iii) the search is performed using dichotomy if the inverted list is larger than one Flash page; and (iv) a subset of the access terms (and corresponding inverted lists) have to be accessed when the expression turns to be *false* (typically, for conjunctive expressions). Our experiments (see Section 9) demonstrate that the execution time overhead due to logical expression evaluation is very acceptable.

We should note that file metadata search and TBAC functionalities can be implemented together in the search engine. In this case, the metadata terms have to be grouped in two classes corresponding to the two functionalities. Then, *Filter* has to evaluate both the logical expression on the file metadata terms and the logical expression on the access control terms and return 1 only if both expressions are true.

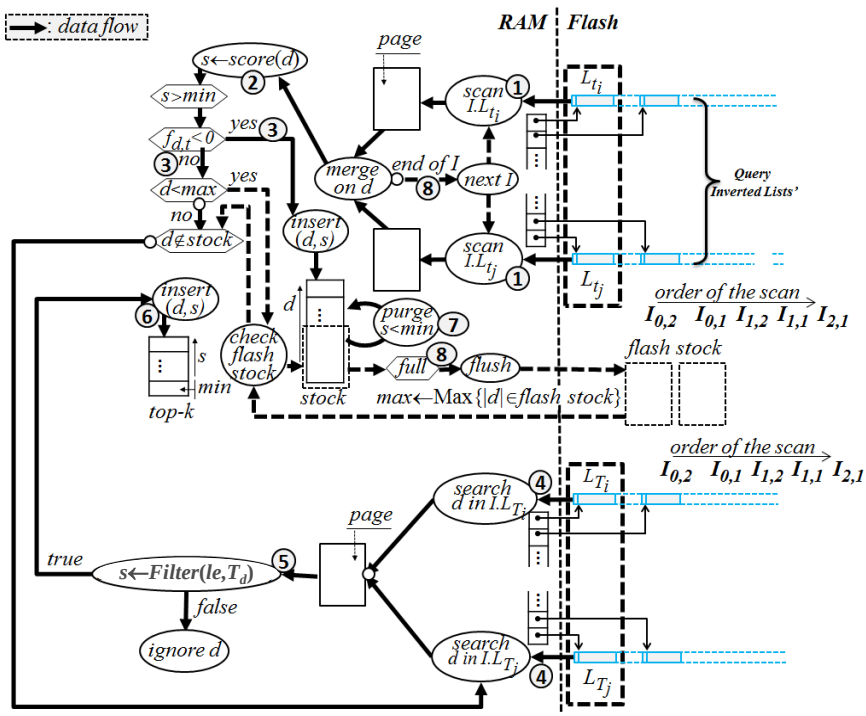

Fig. 7. Linear pipeline computation of a conditional top-k query over terms t_i and t_j and metadata terms L_{Ti} and L_{Tj}

8. Bounded RAM Agreement Conformance

Intuitively, the three design principles introduced in Section 3 conduct to algorithmic solutions complying by construction with the *Bounded RAM agreement* and *Full scalability* properties. This section concentrates on the RAM consumption while the performance and scalability analyses are postponed to Section 9. The primary objective of the Bounded RAM agreement is to ensure that the RAM demand does not depend on the size of the document set, thereby guaranteeing the generality of the studied solution. A rough analysis of the algorithms is sufficient to assess this behavior. However, the second objective of this agreement is to guarantee that the corresponding RAM bound is kept small enough to comply with the widest population of secure tokens. This requires a more subtle analysis of each algorithm, the devil being often hidden in

algorithmic details, especially when algorithms are non-trivial. Thus, Section 8.1 presents the detailed version of the SSF algorithms and Section 8.2 analyses their respective RAM consumption.

8.1. SSF Algorithms

Algorithm 1 presents the pseudocode of the insertion and deletion operations. Since a deletion is treated as an insertion by the SSF, the algorithm is the same for both operations. The only difference is that for an insertion the frequency f_t of each document term is a positive value, whereas for a deletion the term frequencies f_t are negative values. An inserted/deleted document is represented in the index by a set of triples $\{(t, f_t, d)\}$ with t a (distinct) term and f_t its frequency in a document and d the identifier of that document. Each triple is initially in RAM (line 21 in Algorithm 1) until the buffered RAM partition reaches the RAM_Bound limit. Then the RAM partition is written in Flash and its inverted lists are indexed on the terms with a non-dense B^+-tree (see line 8 in Algorithm 1 and also Algorithm 4). After the RAM flush, a merge (see Algorithm 2) is triggered if the number of partitions in the level 0 of the SSF reaches the branching factor b. The algorithm also checks if other merges are required in the upper SSF levels (lines 12 to 18 in Algorithm 1), since a merge in an SSF level creates a new partition in the subsequent level.

Algorithm 1. SSF Insertion / Deletion

Input: A input stream D of triples $\{(t, f_t, d)\}$ with t a (distinct) term and f_t its frequency in a document and d the identifier of that document.
Output: $\emptyset$.

```
1.  P_RAM the RAM partition made of {<t, f_t, {(d, f_d,t)}↓d>}↓t;
2.  for each triple e ∈ D do /* fetch the next triple e from the input stream */
3.     if sizeof(P_RAM) == RAM_Bound then  /* Flush the RAM partition in Flash */
4.        write {t, f_t, {(d, f_d,t)}↓d}↓t from P_RAM to a chained list of Flash sectors;
5.        ptr = address of the first written Flash sector of the new partition;
6.        i = smallest index i | P[0][i] = ∅; /* this smallest i always exists here and i ∈ [1, b] */
          /* Build the hierarchical index and return the address of the root sector */
7.        P_RAM = ∅; /* free P_RAM */
8.        root = Build_hierachical_index(ptr); /* see Algorithm 4 -- NB: P_RAM is released at this stage */
9.        P[0][i] = root;
10.       if (i == b) /* Merge the partitions if nb of partitions in the level reaches the branching factor b */
11.          l = 0;
12.          while P[l][p] ≠ ∅, ∀p ∈ [1, b] do  /* while level l contains b partitions */
13.             Merge(l); /* see Algorithm 2 */
14.             for p = 1 to k do /* free all partitions in level l */
15.                P[l][p] = ∅;
16.             end
17.             l++;
18.          end
19.       end
20.    end
21.    insert the triple e into P_RAM;
22. End
```

Algorithm 2 presents the pseudocode for the merge operation. 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 (i.e., the set of inverted lists) of the output partition is produced (lines 6 to 27 in Algorithm 2). 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 (line 1 in Algorithm 2) and one page to produce the output (line 2 in Algorithm 2). The indexed terms are treated one after the other in alphabetic order (line 6 in Algorithm 2). For each term t, the head of its inverted lists in each partition is loaded in RAM (line 10 in Algorithm 2). These lists are then consumed in pipeline by a multi-way merge (lines 21 and 22 in Algorithm 2). Document ids are encountered in descending order in each list and the output list resulting from the merge is produced in the same order. For the b partition pages loaded in RAM, a border term is computed (line 14 in Algorithm 2). Then, all the terms inferior to the border term can be safely merged and their inverted lists written in the new partition. The border term is updated whenever a new partition page is loaded in RAM (line 10 in Algorithm 2). In the second step (line 29 in Algorithm 2), the *I.S* structure is constructed sequentially, with an additional scan of *I.L* (see Algorithm 4). 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. Hence, the memory consumption of the merge operation will be lower than the RAM_Bound in all cases. The merge algorithm also requires storing b+1 pointers in RAM. However, the RAM consumption for these variables represents only a fraction of a RAM page and is negligible compared to the b+1 RAM pages required by the multi-way merge.

Algorithm 2. SSF Merge

Input: l level of the SSF with the partitions to be merged.
Output: $\emptyset$.

1. $In[b]$ an array of b sectors allocated in RAM ; /* b RAM pages to concomitantly read the b partitions in level l */
2. *Out* a sector allocated in RAM ; /* one RAM page to temporarily buffer the partial result of the merge */
3. *Ptr* the address of a Flash sector ;
4. $Ptr[b]$ an array of b addresses of Flash sectors ;
5. memset(*Out*, $\emptyset$); /* *initialize Out* */
6. $\forall x \in [1, b]$, $Ptr[x]$ = Access_hierarchical_index ($P[l][x]$, *-1 /* lowest term * /*) ;
   /* *initialize Ptr*[] *with the first Flash sector of each partition* */
   /* *Perform in pipeline the Multi-way merge* */
7. **while** $\exists\ Ptr[x] \neq \emptyset \mid x \in [1, b]$ **do**
8. **for** x = 1 to b **do**
9. **if** $Ptr[x] \neq \emptyset$ and $Ram[x]$ *is empty* **then**
10. $In[x]$ = load Flash sector $Ptr[x]$; /* $In[x] \supset$ elements of $\{t, f_t, \{(d, f_{d,t})\}_{\downarrow d}\}_{\downarrow t}$ */;
11. $Ptr[x] = Ptr[x]$ + sizeof(Flash_sector); /* *address of the next partition sector in Flash* */
12. **end**
13. **end**
14. $t_{frontier}$ = min ({ $\forall x \in [1, b]$, max ({ $t \in In[x]$ }) }) ; /* *find the border term for all the terms in RAM, i.e., all the terms inferior to the border term can be safely merged* */
15. **while** $\exists\ x \in [1, b]$ and $t \in In[x] \mid t \leq t_{min}$ **do**
16. **if** *Out* is full **then**

17. write *Out* in the next free Flash sector ;
18. memset(*Out*, ∅); /* *Delete content of Out* */
19. **end**
20. t_{next} = min ({ $t \in In[x] \mid x \in [1, b]$ }) ;
21. E = { elements e_x of type <t, f_t, {(d, $f_{d,t}$)}> | $e_x \in In[x]$, $e_x.t = t_{next}$, $x \in [1, b]$ } ;
22. write in *Out* the element < t_{next} , $\sum_{1 \le x \le b} e_x.f_t$, { e_1.{(d, $f_{d,t}$)} + ... + e_b.{(d, $f_{d,t}$)} } > ;
    /* *the resulted list of* t_{next} *the concatenation of all the partial lists of* t_{next} *in the merged partitions* */
23. remove all the elements $e_x \in E$ from *In* ;
24. **end**
25. write *Out* in the next free Flash sector ;
26. memset(*Out*, ∅);
27. **end**
28. free(*In*); free(*Out*);
29. *root* = Build_hierarchical_index (*Ptr*) ; /* *index the terms of the newly created partition* */
23. p = smallest index $p \mid P[l+1][p] = \emptyset$;
24. $P[l+1][p]$ = *root* ; /* *store in the index metadata the root of the index of the new partition* */
25. **end**

Algorithm 3 presents the query processing algorithm in the SSF. Given a set of query terms an integer value *k*, the algorithm returns an array of *k* couples (*d, tfidf_score*) of document identifiers and their *tf-idf* score. The search algorithm consists in two steps. First, the Ft value of each query term is computed (line 3 in Algorithm 3). This part is described in detail in Algorithm 5. Second, the list of top-k documents with the highest scores is obtained (line 4 in Algorithm 3). This part is described in detail in Algorithm 6.

Algorithm 3. SSF Search

Input: $Q = \{q_i\}$ a set of q query terms; k the requested number of results (top-k).
Output: $R[k]$ an array of k couples (d, *tfidf_score*) of document identifiers and their *tf-idf* score.

1. $Ft[q]$ an array of q values to store the *Ft* frequency for each query term initialized at 0 ;
2. $Ptr[q][l][b]$ a set of pointers to the start of the inverted lists of each query term in each partition in each index level;
3. *Ptr* = Compute_Ft (*Q, Ft*) ; /* *compute the* F_t *for each query term, see Algorithm 5* */
4. *R* = Compute_Top_k (*Q, Ft, Ptr*, *k*) ; /* *compute the top-k results (in pipeline), see Algorithm 6* */
5. **return** *R*;

Algorithm 4 presents the pseudocode of the construction of the hierarchical index (i.e., the *I.S* structure) on top of the set of inverted lists in each partition. The algorithm is invoked in Algorithm 1 (i.e., after the creation of a new partition in the first SSF level) and in Algorithm 2 (i.e., after the creation of a new partition by merging the partitions of an SSF level). The *I.S* structure is constructed sequentially and requires a single full scan of *I.L* previously created. The *I.S* tree is built from the leaves to the root. This requires one RAM page to scan *I.L* (line 1 in Algorithm 4), plus one RAM page to write the *I.S* (line 2 in Algorithm 4). For each Flash page of the I.L containing at least one head-list (line 8 in Algorithm 4), the maximum term and its Flash address are indexed in the index leaves (lines 10 and 13 in Algorithm 4). Once the bottom index level is created, the upper levels of *I.S* are trivially filled sequentially in the same manner through a recursive call (line 24 in Algorithm 4).

Algorithm 4. Build hierarchical index

Input: *ptr* the beginning address of the sorted inverted lists in a partition.
Output: *root* the address of the root Flash sector of the index.

```
1.  ram_in = Alloc_RAM (sizeof(Flash_sector) ) ;
2.  ram_out = Alloc_RAM (sizeof(Flash_sector) ) ;
3.  ptr_in = ptr ; /* pointer to the head Flash sector of the list {<t, f_t, {(d, f_d,t)}↓d>}↓t */
4.  ptr_out = address of the next free sector in Flash ;
5.  no_nodes = 0; /* number of nodes in the currently built index level */
6.  while ptr_in ≠ ∅ do
7.     load in ram_in the Flash sector at address ptr_in ;
8.     get max(t) in ram_in ; /* find the last vocabulary term in this page */
9.     if t ≠ ∅ then
10.       append <t, ptr_in> to ram_out ;
11.    end
12.    if ram_out is full then
13.       write ram_out at address ptr_out ;
14.       no_nodes++ ;
15.       if no_nodes == 1 then
16.          ptr = ptr_out ; /* keep the address of first index node in the current index level for recursive call */
17.       end
18.       ptr_out = ptr_out + sizeof(Flash_sector); /* address of the next free sector in Flash */
19.    end
20.    ptr_in = ptr_in + sizeof(Flash_sector); /* get the address of the next Flash sector */
21. end
22. free (ram_in); free (ram_out);
23. if no_nodes > 1 /* if the current index level has more than one node then recursively index this level */
24.    ptr = Build_hierachical_index(ptr);
25. end
26. return ptr ;
```

Algorithm 5 presents the computation of the F_t values for the query terms, which represents the first phase of the query processing. F_t is 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. Since I is split in $<I_0, I_1,\ldots, I_p>$, the global value of F_t is computed as the sum of the local F_t of all partitions (lines 2 to 10 in Algorithm 5). The algorithm visits all the SSF partitions (lines 2 and 3 in Algorithm 5) and in each partition it uses the *I.S* structure to access the inverted lists corresponding to the query terms (lines 4 and 5 in Algorithm 5). If a query term is found, its global F_t value is increased with the local f_t value (lien 10 in Algorithm 5). For the sake of simplicity, we do not consider in Algorithm 5 the case of the documents overlapping between consecutive partitions. The overlapping documents are detected by checking the two bits (i.e., *firstd* and *lastd*) in *I.S* (see Figure 3). Whether an intersection between two lists is detected, the sum of their respective F_t must be 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. In addition, a set of pointers to the start Flash addresses of the inverted list for each query term in each SSF partition is also stored in RAM to avoid re-accessing the *I.S* of each partition in the second phase of the query processing.

Algorithm 5. Compute *Ft*

Input: a set $Q = \{q_i\}$ of q query terms, $Ft[q]$ an array of Ft frequency values with $Ft[i]$ the value for q_i.
Output: $Ptr[q][l][b]$ a set of pointers with $Ptr[q_i][l_j][p]$ having the start Flash address of the inverted list for query term q_i of partition p in level l_j.

```
1.  ptr a pointer to store the address of a Flash sector;
2.  for l = 0 to max ({ i | P[i][0] ≠ ∅}) do   /* for each level starting from the first one */
3.     for p = 1 to max ({ j | P[l][j] ≠ ∅) do          /* for each partition of that level */
4.        for i = 1 to q do                        /* for each query term */
5.           ptr = Access_hierarchical_index(P[l][p], qi);
            /* find the Flash sector containing the inverted list of qi in this partition */
6.              load in RAM the Flash sector at address ptr;
7.              search in RAM the entry of the element e = {qi, ft, {(d, fd,t)}↓d}↓t ;
8.              if e exists then
9.                 Ft[i] = Ft[i] + e.ft;
10.                compute Ptr[i-1][l][p-1] as the address in Flash of the start of the list e.{(d, fd,t)}↓d ;
11.             end
12.       end
13.    end
14. end
15. return Ptr;
```

Algorithm 6 presents the pseudocode to compute the top-k document identifiers and their scores for a set of query terms, which represents the second phase of the query processing in the SSF. The algorithm takes as input the F_t values of the query terms and set of pointers to the start Flash addresses of the inverted list for each query term in each SSF partition, priorly computed by Algorithm 5. The proposed algorithm works as follows. For each SSF level from the lowest to the highest one (line 5 in Algorithm 6), all the partitions of the SSF level are accessed from the most recent to the oldest one (line 6 in Algorithm 6). For each partition, the *tf-idf* computation sums up to a simple linear pipeline merging process of the inverted lists for all terms $t \in Q$ (lines 8 to 32 in Algorithm 5). The RAM consumption (line 1 in Algorithm 6) for this phase is therefore restricted 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). In addition, two lists are maintained in RAM (lines 2 and 3 in Algorithm 6) : $R[k] = \{(d, score(d))\}$ 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 $R[k]$. The *tf-idf* score of each document d is computed (line 15 in Algorithm 6) 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. $R[k]$ and *Ghost* lists are managed as follows. If the score of the current document d is worse than the smallest score in $R[k]$, it is simply discarded and the next document is considered (line 16 in Algorithm 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 (line 18 in Algorithm 6); else it enters the $R[k]$ list unless its id is already present in the *Ghost* list (lines 20 to 22 in Algorithm 6). 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 (line 23 in Algorithm 6), to filter them out when these documents are later encountered, which makes this list very small in practice. In addition, the probability that the score of a *Ghost* list element competes with the $R[k]$ ones decreases over time, giving the opportunity to continuously purge the *Ghost* list (line 23 in Algorithm 6). If the *Ghost* list overflows (i.e., exceeds the size of a RAM page), it is sorted in descending order of the

document ids, and the entries corresponding to low document ids are flushed. For simplicity, we omitted the flushing process from Algorithm 6. Note also that this situation highly improbable as explained in Section 6.3. None of the queries we evaluated in our experiments has produced an overflow of the RAM page allocated to the *Ghost* list.

Algorithm 6. Compute Top-*k*

Input: $Q = \{q_i\}$ a set of q query terms ; $Ft[q]$ an array of size q with $Ft[i]$ the Ft values of the query term q_i ; $Ptr[q][l][b]$ a set of pointers with $Ptr[q_i][l_j][p]$ storing the address in Flash of the inverted list element $<t, f_t, \{(d, f_{d,t})\}_{\downarrow d}>$ for partition p, level l, and a term q_i ; k the number of documents identifiers requested in result.

Output: $R[k]$ an array of couples (d, *tfidf_score*) of document identifiers and their *tf-idf* score.

1. *Ram*[ ] an array of q sectors allocated in RAM ;
2. $\forall x \in [0, k\text{-}1]$, $R[x].tfidf\_score = 0$; /* *initialize the tfidf scores in the result at 0* */
3. *Ghost* one RAM page to temporarily maintain the current best ranked deleted documents still appearing in the index
4. /* *Multi-way merge of the inverted lists of the query terms in each index partition* */
5. **for** $l = 0$ **to** max ($l \mid \exists\ Ptr[q_i][l_j][p] \in Ptr, p \in [1, b], i \in [1, q]$) **do** /* *for each level containing query terms* */
6. **for** p = max ($p \mid \exists\ Ptr[q_i][l_j][p] \in Ptr$, $i \in [1, q]$) **to** 1 **do** /* *for each partition with query terms* */
7. **for** each $i \mid \exists\ Ptr[q_i][l_j][p] \in Ptr$ **do** /* *for each query term* */
8. load in $Ram[i]$ the first sector of the inverted list $\{(d, f_{d,t})\}_{\downarrow d}$ of q_i from address $Ptr[i][l][p\text{-}1]$;
9. $Ptr[i][l][p\text{-}1] = Ptr[i][l][p\text{-}1]$ + sizeof(Flash_sector);
   /* *compute the next sector address of the current inverted list of* q_i */
10. **end**
11. $d_{frontier}$ = min ($\{\forall i \in [1, q]$, max ($\{d \in Ram[i]\}$) $\}$); /* *find the border document id for all the documents in RAM, ie. all the doc ids inferior to the border doc id can be safely scored* */
12. **while** ($\exists$ a couple $(d, f_{d,t}) \in Ram \mid d \leq d_{frontier}$) **do** /* *compute tfidf for the next docs* */
13. d_{next} = min ($\{d \in Ram\}$);
14. E = {couples $(d, f_{d,t}) \in Ram \mid d = d_{next}\}$;
15. *tfidf_score* = Compute_TFIDF(d_{next}, E, F_t); /* *compute TF-IDF for* d_{next} */
16. **if** *tfidf_score* > min ($\{tfidf\_score \in R\}$) **then** /* *if the score of* d_{next} *enters current best k scores* */
17. **if** d_{next} is a deleted document **then**
18. insert (d_{next}, *tfidf_score*) into *Ghost*
19. **else**
20. **if** $d_{next} \notin Ghost$ **then**
21. delete from R the entry with minimum *tfidf_score* ;
22. insert in R (d_{next}, *tfidf_score*) ;
23. delete from *Ghost* all entries d_{ghost} having $score(d_{ghost})$ < min_score(R) ;
24. remove from *Ram* all the couples $e \in E$;
25. **end**
26. **end**
27. **end**
28. **for** each empty sector $Ram[i] \in Ram \mid Ptr[i][l][p\text{-}1] \neq \emptyset$ **do** /* *scan next sectors* */
29. load in $Ram[i]$ the inverted list $\{(d, f_{d,t})\}_{\downarrow d}$ of q_i from address $Ptr[i][l][p\text{-}1]$;
30. $Ptr[i][l][p\text{-}1] = Ptr[i][l][p\text{-}1]$ + sizeof(Flash_sector);
31. **end**
32. $d_{frontier}$ = min ($\{\forall i \in [1, q]$, Max ($\{d \in Ram[i]\}$) $\}$) ;
33. **end**
34. **end**

35. **end**
36. free(*Ram*); free(*Ptr*);
37. **return** *R*;

8.2. Memory Consumption Analysis and Bounds

We show in this section that the RAM consumption of the algorithms presented above, which implement all the SSF operations (index maintenance and search), never exceeds a predefined and small bound. As indicated in Section 2.2, we consider that a Flash sector has 512 bytes here after. The extension of the analysis to a page granularity is straightforward.

Insertion/deletion. These operations are managed by Algorithm 1 (*SSF insertion/deletion*), which may subsequently call Algorithms 2 (*SSF Merge*) and 4 (*Build hierachical index*). The memory consumption of Algorithm 1 is limited to the current triple *e* to be inserted or deleted (line 2 of Algorithm 1) and the RAM partition of the SSF called P_{RAM} (line 1 in Algorithm 1), which is systematically flushed to the Flash storage and then freed from the RAM (line 7) whenever it reaches a predefined fixed size, i.e., *RAM_Bound*. The *Build_hierachical_index* algorithm is always called after P_{RAM} has been released from the RAM (line 8) and consumes the size of two Flash sectors allocated in RAM (lines 1 and 2 of Algorithm 2) and a few local variables (i.e., two pointers and one integer). Although *Build_hierachical_index* can be called recursively (line 24 in Algorithm 4), a recursive call always happens after the two sectors allocated in RAM are freed (line 22). The *Merge* algorithm is called at line 13 of Algorithm 1 after P_{RAM} is freed and after *Build_hierachical_index* has returned and released the memory (at line 22 of Algorithm 4). *Merge* consumes a RAM size of *b*+1 Flash sectors (line 1 and 2 of Algorithm 2), *b* pointers (line 4) and a few local variables. Note that the value of *b* (the branching factor) is fixed and is chosen such that the RAM consumption of *Merge* never exceeds *RAM_Bound*. Omitting the few RAM variables needed in the algorithms, the RAM consumption of Algorithms 1, 2 and 4 is equal to: MAX(sizeof(inserted triple *e*) + sizeof(RAM partition P_{RAM}); 2*sizeof(Flash_Sector); (*b*+1)*sizeof(Flash_Sector)).

Search. The search operation is managed by Algorithm 3 (*SSF search*), which calls Algorithms 5 (*Compute Ft*) and 6 (*Compute Top-k*). For a search on *q* query terms, the RAM consumption of Algorithm 3 is the size of the *q* query terms, the *k* couples (*d*, *tfidf_score*) being the result of the query, an array of *q* numeric values (line 1) used to store the frequency of each query term, and the size of an array of *q***l***b* pointers to the inverted lists of each of the *q* query term in the *b* partitions of the *l* index levels (line 2). Algorithm 5 only consumes few local variables and Algorithm 6 consumes an array of *q* sectors allocated in RAM to scan the SSF partitions. Omitting the few RAM variables needed in these algorithms, the RAM consumption of Algorithms 3, 5 and 6 is equal to: *q**sizeof(*query_terms*) + *k**sizeof(*d*, *tfidf_score*) + *q***l***b**sizeof(pointer) + *q**sizeof(Flash_sector).

The curves in Figure 8 show the RAM consumption of our algorithms with an increasing number of indexed documents in the database. We fix the maximum number of terms *q* in the search queries to 5, the size of the RAM partition P_{RAM} to 1024 bytes, the number of results *k* to 10, the branching factor b to 4 (left curve) and 8 (right curve), and we increase the database size up to 4GB.These settings are similar with the ones we use in the experimental evaluation here below, except the index size which is much larger than the indexes we obtained with our test datasets. We intentionally increase the index size to a very large value to show that the database size has only negligible impact on the RAM consumption after a certain volume of data (e.g., 500MB) has been indexed.

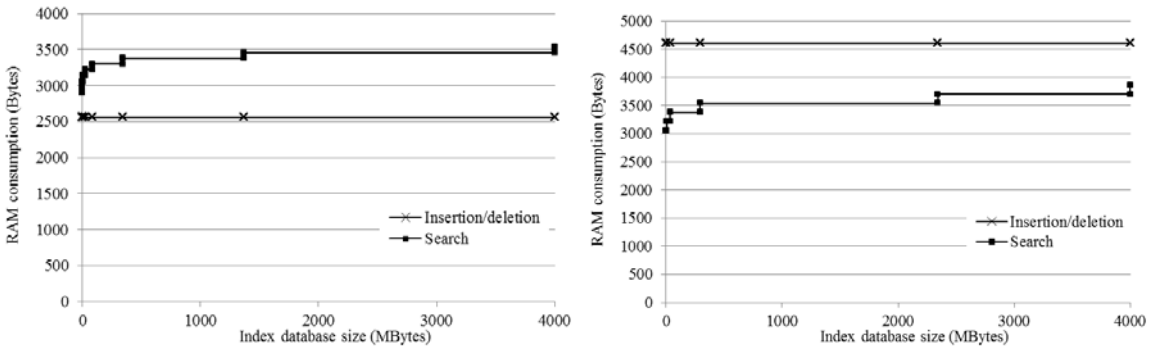

Fig. 8. (a) RAM consumption varying the index database size with b=4 ; (b) RAM consumption varying the index database size with b=8

These curves illustrate the compliance of the proposed structure and algorithms with the *Bounded RAM agreement*. With b=4 (respectively b=8), the RAM requirement never exceeds 3500 bytes (resp. 4600 bytes). Increasing the value of the branching factor b has a negative impact on the RAM consumption required to insert/delete documents. Typically, the *Merge* algorithm requires more memory to comply with the linear pipelining rule. Oppositely, increasing b has a slightly positive impact on the RAM consumption required to evaluate search queries since with less number of levels in the SSF, less pointers are allocated in RAM (line 2 in Algorithm 2). Overall, with these settings, the RAM consumption is lower than 5KB even for an index size of 4GB. Note that the background implementation of the *Merge* does not require additional RAM, since the successive incremental (partial) merge operations are triggered in Algorithm 1, after the Flush of P_{RAM}. We show in the evaluation section below that the background merge process performed incrementally at each P_{RAM} flush terminates far before the next merge operation starts.

We also note that in case a page granularity access (i.e., 2KB I/Os) is used instead of a sector granularity access (i.e., 512B I/Os), the RAM consumption increases proportionally with the size of the I/O since it mainly depends on the I/O size. For instance, the maximum RAM consumption will augment from 5KB to 20KB in the configuration having the branching factor b=8.

9. Full Scalability Conformance

In this section we present an extensive performance evaluation of the proposed search engine to assess whether it complies with the Full scalability property. We introduce the testing hardware platform, the used datasets and the related use-cases in Section 9.1. In Section 9.2, we discuss the index maintenance, i.e., insertion/merge cost and the frequency of the merges. The query performance of the search engine in presented in Section 9.3. The index search performance for advanced keyword-based functionalities is analyzed in Section 9.4. The impact of the deletion rate on the index size and the query performance is discussed in Section 9.5. We also compare both the search and the insert performance of our method with two representative search engines in Section 9.6. Finally, in the light of the obtained experimental results, we discuss the limitations of our approach in Section 9.7.

9.1. Experimental Setup

All the experiments have been conducted on a development board ST3221G-EVAL (see Figure 9) equipped with the MCU STM32F217IG connected to a MicroSD card slot. This hardware configuration is

representative of typical secure tokens [5, 6, 23, 34] or smart objects [33, 35]. The board runs the embedded operating system RTOS 7.0.1 (see https://sourceforge.net/p/freertos/code/HEAD/tree/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 MicroSD NAND Flash card. We tested our index structure with several commercial MicroSD cards (see Table 1). For the complete set of experimental results presented in this section, we selected two representative MicroSD cards (i.e., Kingston MicroSDHC Class 10 4GB and Silicon Power SDHC Class 10 4GB) exhibiting different performance as measured on our development board (see lines 1 and 4 in 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 secure tokens and whatever the fragment of this RAM allocated to the search engine running in competition with other embedded software. To achieve this low RAM_Bound, we access the NAND Flash at a sector granularity (i.e., 512 bytes) as indicated in Section 8.2.

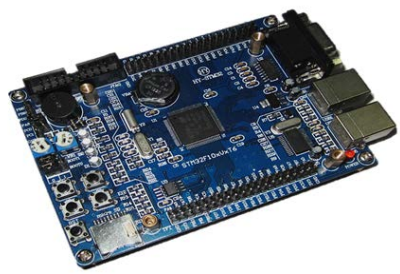

Fig. 9. The development board ST3221G-EVAL used in the experiments

Table 1. Measured performance of common SD cards

Throughput (KB/s) with sector / page granularity	**Read (512B / 2KB)**	**Sequential Write (512B / 2KB)**	**Random Write (512B / 2KB)**
Kingston µSDHC	385 / 1053	79 / 270	3.1 / 11
Lexar SDMI4GB-715	625 / 1667	238 / 833	2.8 / 10
Samsung µSDHC Plus	385 / 1111	172 / 625	1.6 / 6.1
SiliconPower SDHC	357 / 909	147 / 526	12.5 / 36.4
SanDisk µSDHC	417 / 1176	357 / 1111	14.3 / 30.8

We implemented the proposed method and used Microsearch [33] and the classical inverted index [42] for comparison. All the tested methods have been implemented in the C language. The code is compiled and then flushed and executed in the MCU of the development board. The SSF is the most complex method to implement. The inverted index can be seen as a simplified version of the SSF since it does not use partitioning (and therefore there are no merge operations). However, the insertions/deletions are directly applied to the index structure, generating thus a large number of costly in-place updates. Microsearch is also fairly simple to implement since its index structure mainly consists in a set of reversed linked lists with an in-memory table containing a pointer to the Flash address of the last added page of each list. A list corresponds to a hash bucket and consequently contains all the index terms associated with the bucket. Additional details about the implementation of the competing methods are given in Section 9.6.

Datasets and queries. Selecting a representative data and query set to evaluate our solution is challenging considering the diversity and quick evolution of secure token usages, explaining the absence of recognized

benchmarks in this area. We then consider 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 2. Statistics of the datasets and the query sets

ENRON data set and query set	
Number of documents	500000
Total Raw Text	946 MB
Total Unique Words	565343
Total Word Occurrences	52410653
Average Occurrences per Word	92
Frequent Words	30624
Infrequent Words	534719
Frequent Word Occurrences	5.41%
Infrequent Word Occurrences	94.58%
Size of documents in bytes (avg, max)	1KB, 874KB
Size of documents in words (avg, max)	180, 108026
Total number of queries	300
Number of queries with 1, 2, 3, 4 and 5 terms	51, 179, 48, 13, 9
Pseudo-desktop dataset and query set	
Number of documents	27000
Total Raw Text	252 MB
Total Unique Words	337952
Total Word Occurrences	35624875
Average Occurrences per Word	26
Frequent Words	20752
Infrequent Words	317210
Frequent Word Occurrences	6.14%
Infrequent Word Occurrences	93.85%
Size of documents in bytes (avg, max)	8KB, 647KB
Size of documents in words (avg, max)	1304, 105162
Total number of queries	837
Number of queries with 1, 2, 3, 4 and 5 terms	85, 255, 272, 172, 82
Synthetic dataset and query set	
Number of documents	100000
Total Raw Text	129 MB
Total Unique Words	10000
Total Word Occurrences	10000000
Average Occurrences per Word	988
Frequent Words	1968
Infrequent Words	8032

Frequent Word Occurrences	19.68%
Infrequent Word Occurrences	80.32%
Size of documents in bytes (avg, max)	1.3KB, 1.3KB
Size of documents in words (avg, max)	100, 100
Total number of queries	1000
Number of queries with 1, 2, 3, 4 and 5 terms	200, 200, 200, 200, 200

The first use-case is in the Personal Cloud context and considers the use of a secure token embedding a Personal Data Server [4, 5, 34] to securely store, query and share personal files (documents, photos, emails) as presented in Section 1. 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. Protecting and querying the content captured by a set-top-box registering watched TV programs and videos with their related metadata thanks to a secure chip integrated in the set-top-box is no more utopia and is another example of this context. To capture the behavior of our solution in such context, we use two referenced, representative data sets (i.e., ENRON and a pseudo-desktop document collection) and their associated query sets as described below. The interest of using these datasets is that they are large enough to test the index scalability and are well recognized in the IR community.

The second use-case targets the smart sensor context and the case where documents with a poor content are integrated in a Personal Cloud. For instance, home gateways capture a variety of events issued by a growing number of smart appliances, car trackers register our locations and driving habits to compute insurance fees and carbon tax [4]. Here, the documents are time windows, the terms are events occurring during this time window, and top-k queries are useful for analytic tasks. Executing the queries at the sensor side helps reducing the cost, energy consumption and risk of private information leakage. This use-case is representative of situations where the indexing documents have a poor content (hundreds to thousands of terms/event types). Similarly, in the Personal Cloud, poor content documents are typically the binary files (e.g., photo, music or video files) that contain some terms describing the file content (e.g., for a music file the terms may indicate the artist, album, song title, release date, genre, etc.). Therefore, a poor document collection in this case corresponds to a user who mainly stores and indexes binary files in her Personal Cloud. We are not aware of publicly available representative datasets for this context and therefore generate a synthetic dataset for this use-case.

Hence, our evaluation is based on three datasets (see Table 2), which, by their diversity, cover a significant part of the possible use-cases related to smart objects. To evaluate the scalability and efficiency of the search engine, we use the ENRON dataset (available at https://www.cs.cmu.edu/~enron/) composed of 0.5 million emails and a query set of 300 representative queries prepared for this dataset (available at http://www.prism.uvsq.fr/~isap/files/ENRON_queries.zip) built for this dataset. The statistics of the dataset and the queries are given in Table 2. The total number of terms extracted from the ENRON dataset is 565,343 terms, among which 30.624 are frequent. We did not do any text pre-processing (e.g., stemming, feature selection, stop word removal, correction of typos, etc.) of the dataset, which partially explains the very large number of unique terms. However, the text processing is orthogonal to this work since our main concern is the search engine efficiency and not its accuracy. Also, the high number of terms is useful to show the scalability of the proposed search engine.

The second dataset is represented by the pseudo-desktop collection of documents presented in [20]. As ENRON, this dataset applies to the Personal Cloud use-case. The prominent difference from ENRON is that this datasets contains five representative types of personal files (i.e., email, html, pdf, doc and ppt) and not

only emails as in ENRON. The desktop search is an important topic in the IR community. However, real personal collections of desktop files cannot be published because this raises evident privacy issues. Instead, the authors in [20] propose a method to generate synthetic (pseudo) desktop collections and show that such collections have the same properties as real desktop collections. We use in our experiments the pseudo-desktop collection provided in [20]. The statistics of this collection are given in Table 2. As recommended in [20], we preprocess the files in this collection by removing the stop words and stemming the remaining terms using the Krovetz stemmer. This explains the smaller number of terms in the vocabulary (i.e., 337952) compared to ENRON (565343). Nevertheless, the vocabulary is still rich and contains a high number of terms. In addition, the average size of the files is about 8 times larger in the pseudo-desktop collection than in ENRON since the desktop collection contains not only emails but also larger html, pdf, doc and ppt files. In our experiments, we use a set of 837 queries prepared for this dataset and provided in [20].

Finally, the third dataset is a synthetic dataset that we generated to consider the second use-case introduced above. More precisely, we consider the case of a smart meter deployed at home to enable a new generation of energy services. The smart meter records events reported by any smart object in the house (e.g., a specific washing program of a washing machine, played or recorded TV channel for a set-up box, triggering lights or air conditioner in a certain room, etc.). A document in this case corresponds to a time window (e.g., of 1 hour). The document terms are the event identifiers for the events that occur during the time window and their frequency. We generated a synthetic dataset (see Table 2) having a vocabulary of 10000 terms. The synthetic collection contains 100 thousand files, which corresponds to a history of events of about 10 years considering that each file covers a one hour window. On average, each file contains 100 terms. Compared to the previous two datasets, this synthetic dataset covers the use-cases in which the documents have poor content (i.e., small vocabulary and small to average document size). We also generated a set of 1000 random queries to test the index query performance with this dataset.

9.2. 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 in all the experiments the branching factor *b* of the intermediate levels in the *SSF* to 8, to decrease the merge frequency, and the branching factor *b'* of the last level in the *SSF* to 3, to absorb faster the document deletions since the partitions of the last level are the largest.

The insertion or deletion of a new 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 around 6ms to write a partition in L_0 in all our experiments (see Tables 3, 4 and 5). Given the low cost of the metadata insertion, we focus next on the *SSF* merge cost, which is periodically triggered (i.e., each time the number of flushed partitions in L_0 reaches the branching factor *b*). The merge of the partitions in L_0 generates a new partition in L_1, which may generate a subsequent merge from L_1 to L_2 and in subsequent levels.

Tables 3, 4 and 5 present the number of IOs for the flush and merge operations performed in the different *SSF* levels, and their execution times for the three datasets on the two tested SD cards, after the complete insertion of all the documents in the datasets and the random deletion of 10% of documents. 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 7 levels for the ENRON dataset (see Table 3), with 6 levels for the desktop dataset (see Table 4) and for the synthetic dataset (see Table 5). The number of levels of the index structure grows with the number of inserted documents and the average size of a

document. As expected, the merge time grows exponentially from L_0 to L_6, 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 several minutes in L_4 to L_6. 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 distribution of the terms in the indexed documents. However, the partitions begin to contain most of the term dictionary in L_3 and the variation of the merge time in the upper levels is less significant. Note that with the pseudo-desktop collection, only fragments of documents are inserted in the first SSF level since the documents are large. Nevertheless, the document fragments are united in the subsequent SSF levels once the partitions are merged. This fragment condensation also explains the smaller partition sizes of the intermediate levels of the SSF with the desktop dataset compared with the other two datasets.

Table 3. Statistics of the flush and merge operations with the ENRON dataset

	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** [$L_4 \rightarrow L_5$]	**Merge** [$L_5 \rightarrow L_6$]
Number of Read IOs	1 (1)*	67 (72)	585 (738)	3510 (4210)	21034 (23229)	129818 (14550)	404578 (404578)
Number of Write IOs	9 (9)	82 (102)	463 (707)	2738 (3448)	17703 (19771)	119357 (137789)	389774 (389774)
Exec. time on Kingston (seconds)	0.008 (0.0084)	0.67 (0.82)	3.8 (5.6)	22.3 (29.7)	141.3 (158)	944 (1077)	3003 (3003)
Exec. time on Silicon Power (seconds)	0.0044 (0.0045)	0.41 (0.57)	2.54 (3.56)	15.2 (17.7)	92 (102)	604 (688)	1907 (1907)
Total number of occurrences	286265	35783	4473	559	70	9	1
No. of inserted docs between consecutive flushes/merges	2 (27)	17 (134)	132 (964)	1057 (4306)	8410 (20061)	61556 (106601)	150237 (150237)

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

Table 4. Statistics of the flush and merge operations the pseudo-desktop dataset

	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 (seconds)	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 (seconds)	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
No. of inserted docs between consecutive 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.

Tables 3, 4 and 5 indicate that the more costly a merge is, the less frequent it is. Typically, the merges in L_5, which require many minutes to complete, occur after the insertion and deletion of about 61000 ENRON documents or 9000 desktop documents. Only 80 merges costing more than 20 seconds are triggered while inserting the complete set of documents and deleting 10% of the ENRON collection. Even if the costly merges are rare, blocking the index when a merge is triggered for a long duration may be problematic for

some applications. The merge operation is thus implemented in a non-blocking manner as explained in Section 5. After every RAM flush an additional time window of 340ms for Kingston SD card and 210ms for Silicon Power SD card is allocated to resume the current merge operation (if any). The time window is chosen as the minimum time limit (in practice, we increase the minimum time with 10% to avoid any risk of merge overlapping) to guarantee that the merge of a given *SSF* level will end before the next merge of the same level is triggered. After this time delay, the merge is interrupted and its execution is memorized again. In this way, the potentially high cost of a merge operation is spread among a certain number of flush operations.

Table 5. Statistics of the flush and merge operations the synthetic dataset

	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)*	91 (93)	643 (652)	3873 (3934)	20446 (21877)	58985 (58985)
Number of Write IOs	9 (9)	88 (91)	511 (517)	2788 (2817)	17910 (19192)	56037 (56037)
Exec. time on Kingston (seconds)	0.008 (0.0084)	0.69 (0.71)	4.3 (4.3)	24.9 (25.1)	160.3 (172)	516 (516)
Exec. time on Silicon Power (seconds)	0.004 (0.0045)	0.44 (0.45)	2.77 (2.8)	16.4 (16.6)	103 (110)	328 (328)
Total number of occurrences	40195	5025	628	79	10	1
No. of inserted docs between consecutive flushes/merges	2.43 (3)	19 (20)	155 (157)	1238 (1250)	9280 (9995)	22846 (22846)

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

Figure 10 compares the time to execute the merge operations in a blocking and non-blocking manner in the index levels from 3 to 6 with the ENRON dataset (similar results were obtained with the other two datasets). The merges in levels 0, 1 and 2 could not be represented because of their very low cost and high frequency. Also, we only represented the insertion of approximately half of the ENRON dataset in Figure 10, since this is sufficient to capture the overall index update performance, while allowing for reasonable image clarity. We can observe that the cost of the merge in L_6 of the *SSF* is 1907 seconds (32 minutes) with the SP storage, if the merge is performed in a blocking manner. However, this cost will be spread among the next 11483 insert/delete operations (each time the RAM is flushed) using non-blocking merges. Also, a merge triggered in a lower SSF level preempts the current merge in a higher level (if any).

Table 6. Blocking vs. non-blocking merge performance with ENRON

	SD Card	**Max. cost (sec.)**	**Avg. cost (sec.)**
Blocking merge	Kingston	3003	0.23
	SP	1907	0.15
Non-blocking merge	Kingston	0.31	0.29
	SP	0.20	0.18

Table 6 compares the maximum and the average insertion/deletion time in the index with the blocking and the background merge implementation. 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). With the blocking merge, there is large gap between the maximum and the average insertion time, since the maximum insertion time corresponds to the merge time to create a partition in the sixth index level. With the background merge, this gap is much lower, since the cost of the merge operations is amortized over a large number of insertions/deletions. The insertion/deletion time never exceeds 0.31s for Kingston and 0.20s for Silicon Power

(see Table 6). Note also that the non-blocking implementation of merges has a slight impact on the query cost since the number of lower level partitions can temporarily exceed the value of *b* (see next section).

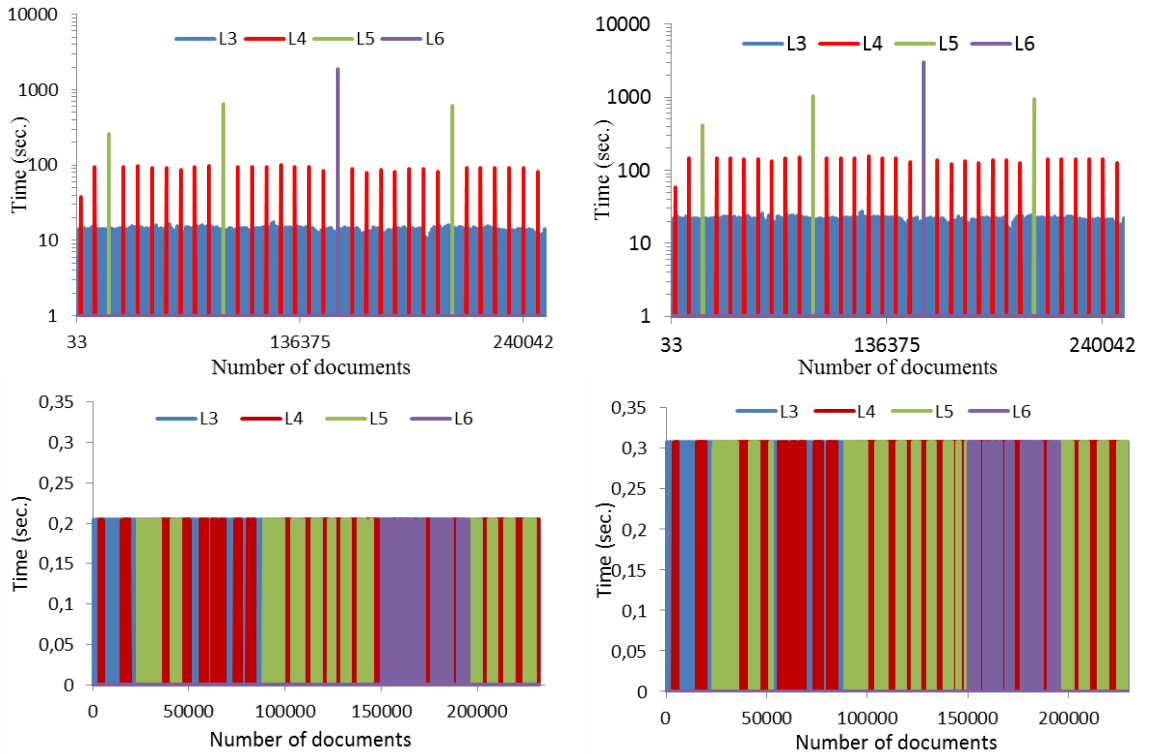

Fig. 10. Insert performances with blocking (up) and non-blocking (down) merge with Silicon Power storage (left column) and Kinston storage (right column) for the ENRON dataset

9.3. Index Search Performance

We evaluated the search performance of the proposed index on our test board and the two SD cards, with both the blocking and non-blocking merge implementations. Because of the similarity of the results, we present in the figures hereafter the results obtained with both storages only with the ENRON dataset. For the other two datasets, we present the results with a single SD card, i.e., Silicon Power. Figure 11 shows the average query time for the 300 search queries in our test query set, as a function of the number of indexed documents in the ENRON collection. The query set mixes queries consisting of 1 and up to 5 terms (see Table 2). For simplicity, the curves in Figure 11 present the query cost before (i.e., the "max" curves) and after (i.e., the "min" curves) each merge occurring in the higher index levels, i.e., from the level 3 to the 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* (see the arrows in Figure 11), while the intermediary peaks correspond to the creation of a partition in level 4 of the *SSF*.

Globally, the query time also increases linearly with the number of indexed documents, but the linear increase is very slow. For example, we see that after inserting 500K documents and deleting 10% of them, our

search engine is able to answer queries with an average execution time of only 0.45s and a maximum time of 0.79s for Kingston and an average of 0.49s and a maximum of 0.89s for Silicon Power (with the non-blocking merge implementation). The query times are lower with a blocking merge, i.e., an average of 0.35s and a maximum time of 0.67s for Kingston and an average of 0.38s and a maximum of 0.72s for Silicon Power. The non-blocking merge leads to an increase of the query cost because 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 28% compared with the query time with a blocking merge. The query time increase is approximately 0.1s seconds for Kingston and 0.11 seconds for Silicon Power and appears to be a fair trade-off for applications that cannot accept unpredictable or unbounded update index latencies. Note that the increase of the query cost with a non-blocking merge can be decreased by enlarging the time window allocated to periodically process a merge.

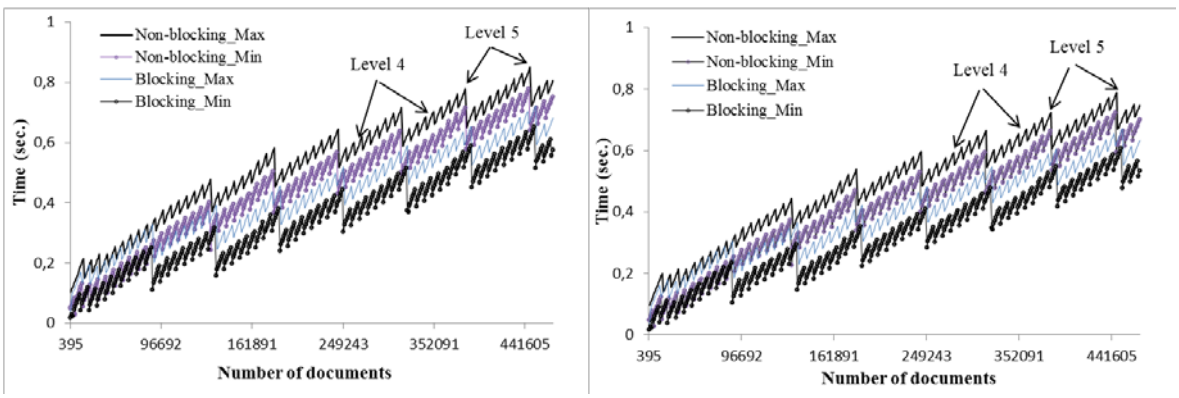

Fig. 11. Query performances with blocking and non-blocking merge with Silicon Power (left) and Kingston (right) storage for the ENRON dataset

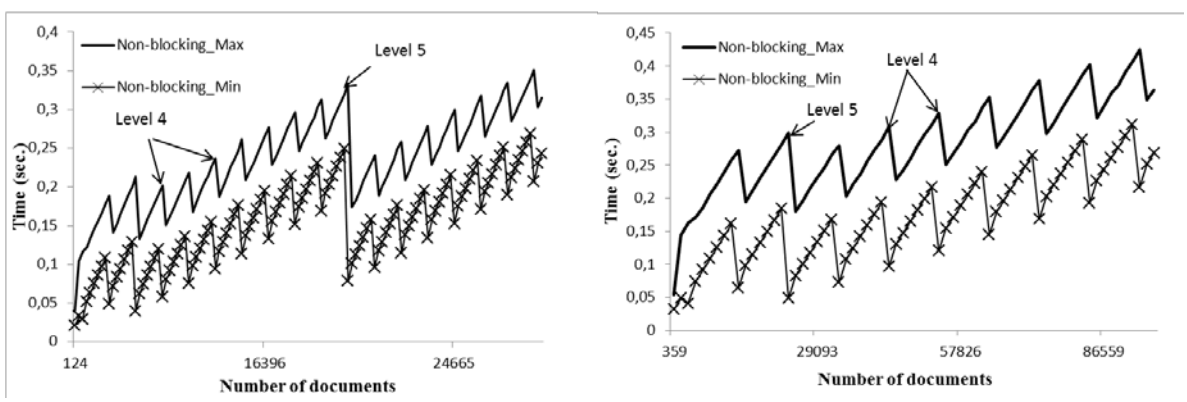

Fig. 12. Query performances with blocking and non-blocking merge with Silicon Power storage for the Pseudo-desktop (left) and the synthetic (right) datasets

Figure 12 presents the evolution of the query time with the number of indexed documents for the pseudo-desktop and synthetic datasets. The deletion rate in this figure is 10% as with ENRON. For better readability of the figures, we present the minimum and the maximum query times only with the non-blocking merge

implementation. As with the ENRON dataset, the query times with a blocking merge are about 25% lower than with the non-blocking merge. With the pseudo-desktop dataset, our index exhibits an average execution time of only 0.17s and a maximum time of 0.32s for Kingston and an average of 0.18s and a maximum of 0.35s for Silicon Power. With the synthetic dataset, the average and the maximum execution times are 0.19s and 0.39s respectively for Kingston, while for Silicon Power the average is 0.21s and the maximum time is 0.42s. As with the ENRON dataset, globally the query cost increases linearly with the index size. The overall index size is lower for the pseudo-desktop and synthetic datasets compared with ENRON (see Section 9.5), which explains the smaller values of the average query times with these two datasets.

9.4. Index Search Performance for Advanced Keyword-based Functionalities

In this section, we present the impact of the advanced keyword search implementation (see Section 7) on the query performance. For the sake of simplicity, we limit the scope of the evaluation to conjunctive logical expressions. To generate queries with logical expressions, we select part of the indexed terms as metadata terms and associate to each query a set of such metadata terms. Regarding the metadata terms, there are two major factors that impact the query performance: the popularity of the term (i.e., number of documents that contain the metadata) and the number of terms in the logical expression. Hence, to test the impact of logical expressions on the search engine performance, we vary the frequency of the involved metadata terms and also the number of metadata terms involved in each logical expression. To be able to compare the results with our previous measures, we consider the same datasets as above, namely the ENRON and the Pseudo-desktop data sets, and the Silicon Power microSD card. In the next experiments, we have fixed the top-k threshold to 10 (i.e., k=10).

Impact of the popularity of the metadata terms on the query performance. To test the impact of the frequency of the metadata terms on the query performance, we select sets of terms appearing with different frequencies in the considered data sets. We build three groups of 100 terms. For Enron, the first group contains 100 terms appearing in the document collections with low frequency (i.e., an average of 27 appearances), the second group contains 100 terms with medium frequency (i.e., an average of 325 appearances) and the third group contains 100 terms with very large frequency (i.e., an average of 43765 appearances). For Pseudo-desktop, the first group of terms has an average of 11 appearances, the second group an average of 76 appearances and the third group an average of 4964 appearances. We then randomly select 100 queries from the query sets, and we associate each query with one metadata term from each group. We measure the average query time without any metadata term, with a logical expression made of a single term taken from the first group of metadata terms, then taken from the second group and lastly taken from the third group.

Table 7. Impact of metadata term popularity on the query performance (in seconds) with Silicon Power storage.

	No metadata	**Group 1 metadata**	**Group 2 metadata**	**Group 3 metadata**
ENRON	0,77	0,82	0,82	1,34
Pseudo-desktop	0,27	0,29	0,29	0,43

Table 7 shows the average query performance after the insertion of the whole data set depending on the query group. We can observe that the impact of the metadata term popularity on the query performance is very low for groups 1 and 2, but higher for group 3. With the ENRON data set, the search engine is able to answer queries with an average execution time of 0,77s without any metadata, 0,82s for the first and second groups of metadata terms, and 1,34s for the third group. With the pseudo-desktop data set, the average execution time of queries without metadata is 0,27s, for metadata terms in groups 1 and 2 is 0,29s, and for

group 3 is 0,43s. Globally, the query time of groups 1 and 2 increases of about 8% compared to queries without metadata and about 68% for the queries in group 3. However, as indicated by the statistics of the data sets in Table 2, only 5% of the terms are frequent in the ENRON data set and 6% in the Pseudo-desktop data set. The frequent terms are not the common case in a data set dictionary and we expect to have a similar distribution for the metadata terms. Therefore, the query performance with one metadata term will be marginally impacted in general as demonstrated by the results of the first two groups in Table 7.

Table 8. : Impact of number of metadata terms on the query performance (in seconds) with Silicon power storage.

	No metadata	**Group 1 metadata**	**Group 2 metadata**	**Group 3 metadata**
ENRON	0,77	1	0.97	0.93
Pseudo-desktop	0,27	0.35	0.34	0.33

Impact of the number of metadata terms on the query performance. To meet the different needs of personal cloud advanced keyword searches, logical expression can contain several metadata terms. To measure the impact of the number of metadata terms on the query performance, we consider logical expressions which contain 3, 5 and 7 terms randomly chosen from the data set. These expressions are then combined with the workloads containing 100 queries for each data set. Table 8 shows the average query time of the queries without logical expressions, and in the presence of a logical expression involving 3 metadata terms (group 1), 5 metadata terms (group 2) and 7 metadata terms (group 3). The queries are evaluated after inserting the complete respective data set. We can see that the search engine is able to answer queries with an average execution time of 1s for queries of group 1, 0.97s for queries of group 2 and 0.93s for queries of group 3 for the ENRON data set. A similar behavior is observed with the pseudo-desktop data set, where the average execution time of queries is 0.35s for queries of group1, 0.34s for group 2 and 0.33s for group 3.

These experimental results show that the query performance is only marginally impacted by the number of terms in logical expression with a maximum increase of 30% compared to basic queries. The better performance for the queries in group 3 compared to the queries in group 1 and 2 can be explained by our evaluation strategy applied to conjunctive expressions. Indeed, we evaluate the logical expression by starting with the least frequent access term, which avoids accessing the subsequent access terms if the condition on the current term is not satisfied. Thus, many inverted lists corresponding to the most frequent access terms do not need to be accessed, which happens with a higher probability if the expression has more terms.

9.5. Index Performance with Various Deletion Rates

In this section we discuss the impact of the deletion rate on the index search performance and index size. Since the deletions are executed as insertions in our search engine, the performance of delete operations is the same with the insert performance. However, the deleted documents are temporarily stored in the index structure (i.e., until the deletions are absorbed during the index merges), which leads to an increase of the index size compared to the optimal index size (i.e., where deleted entries are directly purged). The increased index size can degrade the query performance. Nevertheless, the experimental results show that the query performance is only marginally impacted by deletions even for high deletion rates.

Table 9 shows the average query performance for different deletion rates with the Kingston storage (we obtained similar results with the SP storage). Here, we considered two cases. First, we inserted the whole dataset while deleting a number of documents corresponding to the deletion rate (lines 1, 3 and 5 in Table 9). 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 regardless the deletion rate (lines 2, 4 and 6 in Table 9). 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 12% for deletion rates up to 50%. Globally, the index is robust with the number of deletions in both cases.

Table 9. Average query performance (in sec.) with different deletion rates and Kingston storage

	0%	**10%**	**30%**	**50%**
Avg. query time (ENRON: 500k docs)	0.49	0.45	0.41	0.37
Avg. query time (ENRON: 250k docs)	0.33	0.34	0.35	0.37
Avg. query time (Desktop: 27k docs)	0.18	0.17	0.16	0.15
Avg. query time (Desktop: 13k docs)	0.12	0.13	0.15	0.16
Avg. query time (Synthetic: 100k docs)	0.13	0.13	0.12	0.11
Avg. query time (Synthetic: 50k docs)	0.10	0.10	0.11	0.11

Table 10 shows the index size for the three 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, we give in Table 10 for each dataset and deletion rate the index size of the classical inverted index used as reference. Without deletions, the SSF index size is practically the same with the inverted index size. The SSF requires a little bit more storage for the *I.S* since each partition has its own search structure to index the terms in the partition. Nevertheless, the storage overhead of the SSF is negligible since the search structure represents less than 1% of the global index size (i.e., the inverted lists require much more storage than the search structures). With deletions, the size of the SSF index is larger than the size of the inverted index. Also, the size difference increases with the delete ratio. The explanation is that the deleted documents are reinserted in the SSF, which temporarily increase the index size. However, when a merge is triggered in an index level, a part of the deletions are absorbed and the deleted documents are purged from the index. Therefore, at any given time only a part of the deleted documents are still present in the index. Typically, in our tests we observed that about 45% to 55% of deletions are not absorbed after a high number of document insertions and deletions. This makes that the SSF index size to be at most 40% larger than the inverted index size even for high deletion rates, which we believe is quite acceptable.

Table 10. Index (inverted lists/search structures) size (MB) with different deletion rates (from 0% to 50%)

	0%	**10%**	**30%**	**50%**
SSF I.L/I.S size (ENRON: 500k docs)	439 / 3.5	402 / 3	350 / 2.4	310 / 2.2
Inverted index I.L/I.S size (ENRON: 500k docs)	439 / 0.6	397 / 0.6	305 / 0.6	220 / 0.6
SSF I.L/I.S size (Desktop: 27k docs)	81 / 1.24	76 / 1.14	60 / 0.82	55 / 0.73
Inverted index I.L/I.S size (Desktop: 27k docs)	81 / 0.4	73 / 0.4	57 / 0.4	40 / 0.4
SSF I.L/I.S size (Synthetic: 100k docs)	78 / 0.97	74 / 0.88	66 / 0.78	58 / 0.69
Inverted index I.L/I.S size (Synthetic: 100k docs)	78 / 0.13	70 / 0.13	55 / 0.13	40 / 0.13

9.6. Comparison with the State-of-the-Art Search Engine Methods

The existing solutions face important limitations when used to index large collections of documents in secure tokens. The classical inverted index [42] was designed for magnetic disks and therefore does not comply with the Flash storage constraints. Besides, the few proposed methods that consider the constraints of secure tokens process the insertions efficiently, but do not scale with large datasets and cannot support index updates.

In this section we compare our proposed search engine (called SSF here) with the representative indexing method of each of the aforementioned approaches. Hence, we choose the typical inverted index to represent the query-optimized index family (see Figure 2) and Microsearch [33] for 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 for two competing methods, i.e., by using a RAM_Bound equal to 5KB. The data insertions in the inverted index are processed in a similar way as in our search engine. The insertions are first buffered in RAM until the RAM_Bound is reached. Then, the buffered updates are applied in batch to the index structure. However, different from our approach, the index structure is modified in-place, which requires costly random writes. Also, to be able to evaluate the queries under the RAM constraint with the inverted index, the inverted lists of this structure have to be maintained sorted on the document ids, which permits applying a linear pipeline query processing similar to the SSF. In the case of Microsearch [33], 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. We note also that in [33] Microsearch was implemented using NOR Flash storage, which allows the elements of the linked lists to have different sizes. In our case, the elements of the lists have always the size of a sector (i.e., 512 bytes) but this modification has a limited impact on the global performance of the method.

Insertion performance. Figures 13 and 14 show the average insertion time for the three methods (i.e., SSF, Microsearch and the Inverted Index) with the three datasets. Both Microsearch and SSF exhibit good insert performance. On average, for ENRON, a document insertion in Microsearch takes about 0.094s with Kingston and 0.059s with Silicon Power, and 0.14s with Kingston and 0.09s with Silicon Power in SSF. In comparison, the document insertion time in the Inverted Index is much larger because of the costly random writes in Flash memory. On average, an insertion requires 30s with Kingston and 7.6s with Silicon Power, which is nearly two orders of magnitude higher than with the embedded search engines, and clearly dismisses this method in the context of secure tokens. We obtained similar results with the two other datasets as presented in Figure 14. On average, with Silicon Power storage, for the synthetic collection, a document insertion in Microsearch takes about 0.02, 0.07s in SSF and 15s in the inverted index. For the pseudo-desktop collection, a document insertion in Microsearch takes about 0.08s, 0.33s in SSF and 30s in the inverted index. For all the methods, the insertion times are proportional to document size. Therefore, the insertion times are the highest with the pseudo-desktop collection.

The insertion in the SSF and Microsearch relies on sequential writes in Flash to comply with the Flash memory constraints. The higher insertion cost of the SSF compared to Microsearch is generated by the SSF merges. Although the SSF insertions are less efficient than with Microsearch, the insertion time remains reasonable and will probably satisfy most of the applications especially if they require indexing large collections of documents. Indeed, the slight increase of the insert cost is outweighed by the query performance and scalability of the SSF.

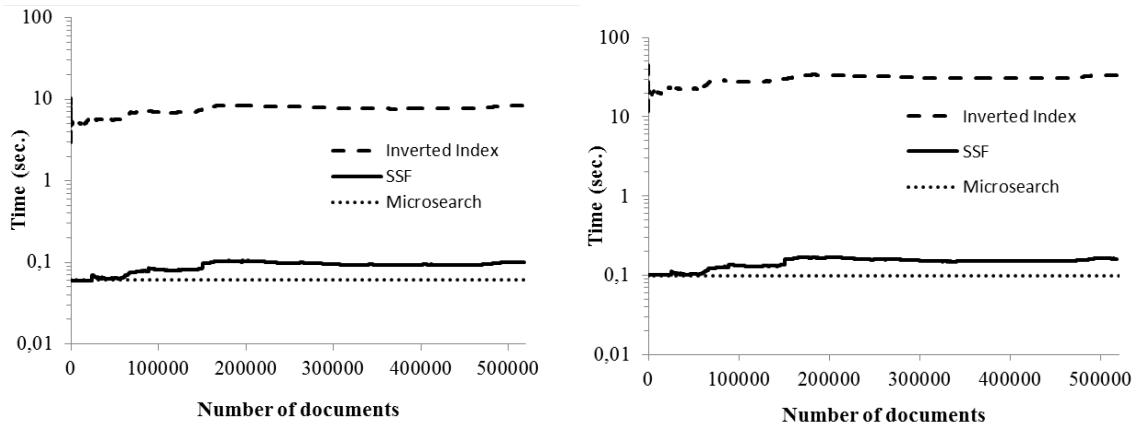

Fig. 13. Average document insertion times of Microsearch, SSF and the Inverted Index with Silicon Power (left) and Kingston (right) storage for the ENRON dataset

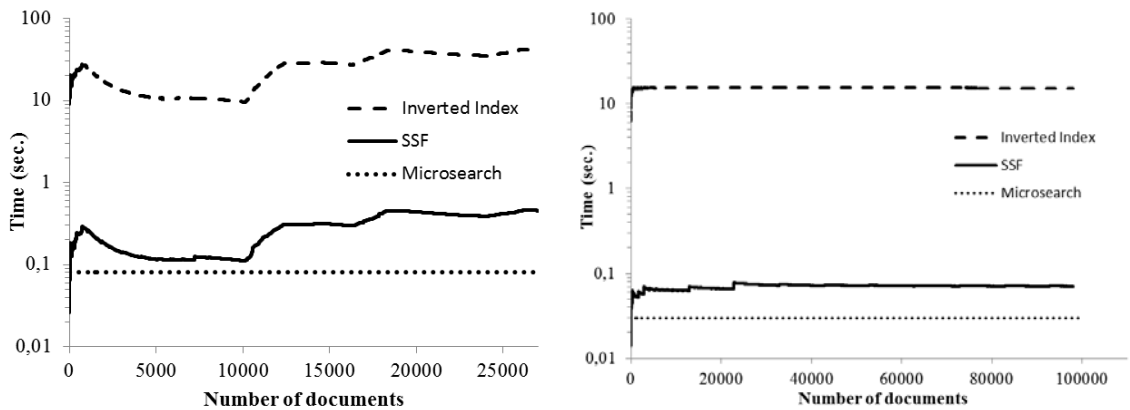

Fig. 14. Average document insertion times of Microsearch, SSF and the Inverted Index with Silicon Power storage for the pseudo-desktop (left) and synthetic (right) datasets

***Query performance*.** Figures 15 and 16 show the query execution time for the three methods in function of the number of indexed documents. The Inverted Index has the best query times and can be considered as the most efficient structure for processing full-text search queries. We can see that query times of the SSF are very close to the Inverted Index times. On average, for ENRON, it takes 0.49s with Kingston and 0.53s of Silicon Power to process a query in the SSF, while for the Inverted Index it takes 0.16s with Kingston and 0.17s with Silicon Power. Also, the average query times with Silicon Power are 0.18s and 0.05s in the SSF, 0.07s and 0.02s in the inverted index, and 880s and 355s in Microsearch, for the pseudo-desktop and the synthetic respectively. The difference in performance between the SSF and the Inverted Index is explained by the fragmentation of the index structure of the SSF. However, the index partitioning in the SSF is largely outweighed when we take into account the insert performance of these two index structures.

Microsearch has the worse query performance, which clearly is not scalable to a large number of documents. On average, for ENRON, it takes 1728.80 second (28 minutes) with Kingston and 1861.78 second

(31 minutes) with Silicon Power to process a query with Microsearch. Given the very large query times, we measured the real query time only up to 10000 indexed documents and estimated the query times above this number of documents, which is fairly simple since the query time linearly increases with the number of documents in Microsearch. Note also that even for a low number of documents, the query times of Microsearch are larger than the query times of SSF. For instance, for 1000 documents it takes 3.1s with Kingston in Microsearch and 0.1s in SSF. The first reason is that in Microsearch an inverted list corresponds to a large number of terms (i.e., all the terms are distributed in the 8 hash buckets). Therefore, a large part of the index data is retrieved by the query. Second, Microsearch requires two passes over the chained list containing a query term. The first pass is done to compute the global F_t value of the term, while the *tf-idf* score of the documents containing the term is computed only in the second pass.

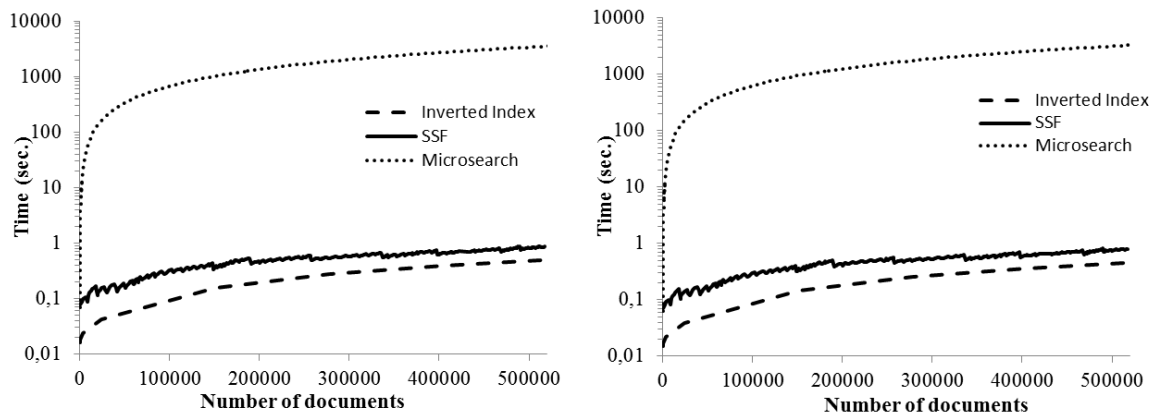

Fig. 15. Query execution times with the Inverted Index, SSF and Microsearch with Silicon Power Power (left) and Kingston (right) storage on the ENRON dataset

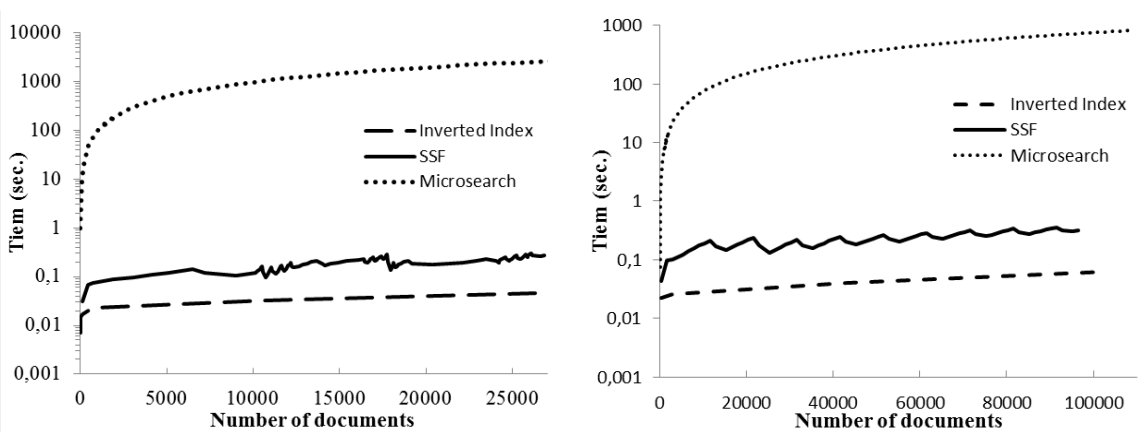

Fig. 16. Query execution times with the Inverted Index, SSF and Microsearch with Silicon Power Power storage for the pseudo-desktop (left) and synthetic (right) datasets

Overall performance. The tiny RAM and the NAND Flash storage of sensors introduce conflicting constraints on the index structure. Figure 17 summarizes the update and the query performance of proposed index structure and the two representative competitors. Our solution is the only one to offer both query and update scalability under these constraints by balancing the query and the update costs for any kind of document collection. At the same time, the loss in both the query and the update performance remains reasonable compared with the query optimized index (i.e., the Inverted Index) and the insert optimized index (i.e., Microsearch). Figure 18 shows the speedup of SSF (i.e., the ratio between the throughput of SSF and of the competitors) with Kinston for workloads containing insertions and queries in different ratios. 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.

A straightforward way to improve the efficiency of all the tested methods would be to increase the I/O granularity, e.g., passing from 512 bytes I/Os to 2 Kbytes I/Os. Specifically, the insertion and query performance of all the methods is expected to improve in this case by taking advantage of the throughput increase of the storage device (see Table 1). However, this leads to increased RAM consumption as discussed in Section 8.2, a thorny problem for secure tokens. Note that by increasing the I/O granularity, the relative performance of the tested methods is expected to remain approximately the same, i.e., we expect to have the similar speedup values of the SSF compared with Microsearch or the inverted index since the throughput augments approximately in the same proportion for all types of I/Os.

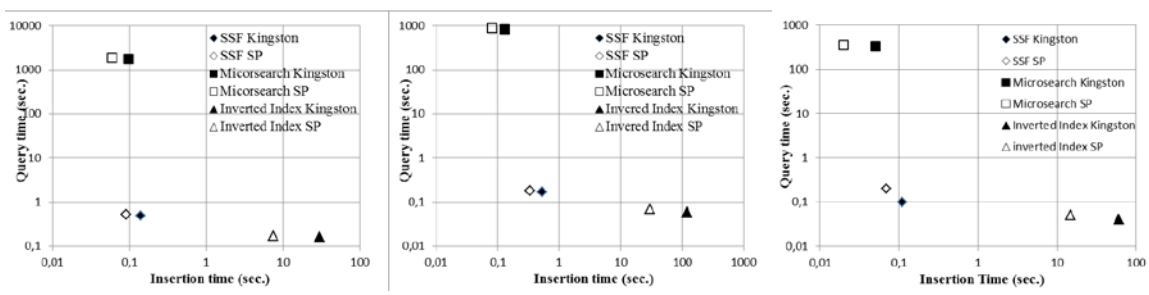

Fig. 17. Overall performance comparison for ENRON (left), pseudo-desktop (middle) and synthetic (right) datasets

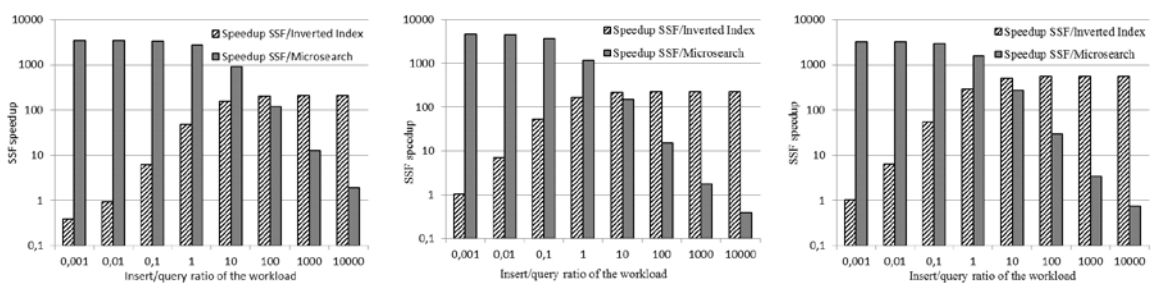

Fig. 18. Overall speedup comparison for ENRON (left), pseudo-desktop (middle) and synthetic (right) datasets with Kingston storage

9.7. Discussion

In the light of the above presented experimental results, we discuss in this section the limitations of the proposed approach. As above, we mainly analyze the limitations of the SSF in comparison with the classical inverted index, which can be considered as the ideal structure to index text documents at least from the query performance and index size points of view. Hence, the shortcomings of the SSF originate from the major differences between the SSF and the inverted index structures, i.e., the index partitioning and the deletion processing.

The partitioning of the SSF influences both the query and the insert performance. The query performance is degraded because of the multiple searches in several small *I.S* (i.e., the search index dictionary of each partition) are more costly than a single search in a large *I.S*. However, given the exponentially increasing size of the partitions in the SSF levels, the loss in the query performance is limited compared with the inverted index (see Figures 15 and 16). Also, the query performance loss is diminishing with the increase of the number of indexed documents, suggesting that our approach is particularly appropriated for indexing large datasets. We should also note that the partitioning introduces some variability in the query cost (see the stairway-like curves in Figures 11 and 12). The partitioning has an impact on the insert performance since already inserted documents have to be rewritten at an index merge. Yet, the insertions only generate sequential writes in Flash and the insert cost is much more scalable than for the inverted index (see Figures 13 and 14). Finally, the partitioning has also an impact on the index size since the search structure that indexes the terms in each partition is redundant. However, the increase in the index size is negligible (see Table 10) as the search structures only take a small fraction from the global index size.

The specific way of processing deletions in the SSF also has an impact on the query performance and the index size. Since a deleted document is first reinserted in the index, deletions lead to a temporal 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. The experimental results show that, even for high deletion rates of up to 50%, the increase in the index size is lower than 40%, while the increase in the query cost is lower than 12% (see Tables 9 and 10).

Finally, another limitation of the proposed search engine is that we do not consider the problem of concurrent access, i.e., multiple processes that query/update the index at the same time. In our workloads combining queries/inserts/deletes, we analyze the cost of each separately, one operation at a time. Indeed secure tokens, contrary to central servers, rarely support parallel or multi-task processing. Moreover, the RAM consumption increases linearly with the number of operation executed in parallel, a serious constraint in our context.

10. Conclusion

In this paper, we present the design of an embedded search engine for secure tokens equipped with low RAM and large Flash storage capacity. The proposed method contributes to the development of the Personal Cloud paradigm by allowing users to securely store, query and share their document collections. In addition, 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 Bounded RAM agreement, our search engine can accommodate a wide population of secure tokens, 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 having a configuration representative of secure tokens. The experimental evaluation validates its efficiency and scalability over three real and synthetic large datasets representative of different smart object scenarios and also demonstrates the superiority of this approach compared to state of the art methods. Finally, we show that, at the price of a direct extension, our search engine can cope with conditional top-k queries, broadening its application scope.

The next step is to study how our three design principles can be generalized for building other kinds of embedded query engines (e.g., NoSQL like), considering that indexing any form of data streams in mass storage smart objects will encounter similar hardware constraints and then lead to similar requirements. Also, in the context of smart objects the energy efficiency of the search engine can be important (e.g., for battery powered smart objects). Another direction of work is to study the energy consumption of our approach in comparison with other existing approaches.

Acknowledgements

This work was partially supported by the ANR KISS project [grant n°ANR-11-INSE-0005] and ANR PerSoCloud project [grant n°ANR-16-CE39-0014-02].

References

[1] Aggarwal, C. C., Ashish, N., and Sheth, A. The internet of things: A survey from the data-centric perspective. In Managing and mining sensor data (pp. 383-428). Springer US, 2013.

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

[3] Agrawal, D., Li, B., Cao, Z., Ganesan, D., Diao, Y. and Shenoy, P.J.: Exploiting the Interplay between Memory and Flash Storage in Embedded Sensor Devices. RTCSA 2010: 227-236, 2010.

[4] Anciaux, N., Bonnet, P., Bouganim, L., Nguyen, B., Sandu Popa, I., and P. Pucheral. Trusted cells: A sea change for personal data services. In CIDR, 2013.

[5] Anciaux, N., Bouganim, L., Pucheral, P., Guo, Y., Folgoc, L. L., and Yin, S. Milo-db: a personal, secure and portable database machine. Distributed and Parallel Databases, pp. 37-63, 2014.

[6] Anciaux, N., Lallali, S., Sandu Popa, I., and Pucheral, P.: A Scalable Search Engine for Mass Storage Smart Objects. PVLDB 8(9): 910-921 (2015)

[7] André, B., Anciaux, N., Pucheral, P. and Tran Van, P.: A Root of Trust for the Personal Cloud, ERCIM News 2016(106), 2016.

[8] Athanassoulis, M., Kester, M.S., Maas, L.M., Stoica, R., Idreos, S., Ailamaki, A. and Callaghan, M.: Designing Access Methods: The RUM Conjecture. EDBT 2016: 461-466, 2016.

[9] Au Yeung, C.-m., Kagal, L., Gibbins, N. and Shadbolt, N.: Providing Access Control to Online Photo Albums Based on Tags and Linked Data, AAAI Spring Symposium on Social Semantic Web: Where Web 2.0 Meets Web 3, pp. 9-14, 2009. conf A*, 33 citations

[10] Bernstein, P.A., Reid, C.W. and Das, S.: Hyder - a transactional record manager for shared flash. In CIDR, pages 9–20, 2011.

[11] Bertino, E.: Data Security and Privacy in the IoT. EDBT 2016: 1-3, 2016.

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

[13] Debnath, B., Sengupta, S., and Li, J. Skimpystash: Ram space skimpy key-value store on flash-based storage. In Proc. of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD ’11, pp. 25–36, 2011.

[14] Diao, Y., Ganesan, D., Mathur, G., and Shenoy, P. J. Rethinking data management for storage-centric sensor networks. In CIDR, pp. 22–31, 2007.

[15] Ehlers, J.H. and Jassim, S.A.: Wavelet library for constrained devices. Proc. SPIE 6579, Mobile Multimedia/Image Processing for Military and Security Applications 2007, 65790P (May 02, 2007); doi:10.1117/12.720713, 2007.

[16] 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, vol. 3, pp. 35–60. 2013.

[17] Jagadish, H.V., Narayan, P.P.S., Seshadri, S., Sudarshan, S., Kanneganti, R.: Incremental organization for data recording and warehousing. In: Proc. of VLDB, pp. 16–25. 1997

[18] Jermaine, C., Datta, A., Omiecinski, E.: A novel index supporting high volume data warehouse insertion. In: Proc. of VLDB. pp. 235–246. 1999

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

[20] Kim, J. Y. and Croft, W. B. Retrieval Experiments using Pseudo-Desktop Collections. In CIKM, 2009.

[21] Klemperer, P., Liang, Y., Mazurek, M., Sleeper, M., Ur, B., Bauer, L., Cranor, L.F., Gupta, N. and Reiter, M.: Tag, you can see it!: using tags for access control in photo sharing, Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'12), pp. 377-386, 2012. conf A*, 61 citations

[22] Lallali, S., Anciaux, N., Sandu Popa, I. and Pucheral, P.: A Secure Search Engine for the Personal Cloud. SIGMOD Conference 2015: 1445-1450

[23] Le, T., Anciaux, N., Guilloton, S., Lallali, S., Pucheral, P., Sandu Popa, I. and Chen, C.: Distributed Secure Search in the Personal Cloud. EDBT 2016: 652-655

[24] Li, Y., He, B., Yang, R. J., Luo, Q., and Yi, K. 2010. Tree Indexing on Solid State Drives. VLDB J (2010), 3:1195-1206.

[25] Li, T., Liu, Y., Tian, Y., Shen, S. and Mao, W.: A storage solution for massive iot data based on nosql. In Green Computing and Communications (GreenCom), 2012 IEEE International Conference on, pages 50–57, Nov 2012.

[26] Lim, H., Fan, B., Andersen, D.G. and Kaminsky, M.: Silt: A memory-efficient, high-performance key-value store. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP ’11, pages 1–13, New York, NY, USA, 2011. ACM.

[27] Mazurek, M.L., Arsenault, J.P., Bresee, J., Gupta, N., Ion, I., Johns, C., Lee, D., Liang, Y., Olsen, J., Salmon, B., Shay, R., Vaniea, K., Bauer, L. Cranor, L.F., Ganger, G.R. and Reiter, M.K.: Access Control for Home Data Sharing: Attitudes, Needs and Practices, Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10), pp. 645-654, 2010. conf A*, 87 citations

[28] Mazurek, M.L., Klemperer, P.F., Shay, R., Takabi, H., Bauer, L. and Cranor, L.F.: Exploring reactive access control, Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11), pp. 2085-2094, 2011. cité 22 fois, A*

[29] Mazurek, M.L., Liang, Y., Melicher, W., Sleeper, M., Bauer, L., Ganger, G.R., Gupta, N. and Reiter, M.K.: Toward Strong, Usable Access Control for Shared Distributed Data, Proceedings of the 12th USENIX conference on File and Storage Technologies, FAST'14, pp. 89-103, 2014. conf A, 8 citations

[30] O’Neil, P.E., Cheng, E., Gawlick, D., O’Neil, E.J.: The Log-Structured Merge-Tree (LSM-Tree). Acta Informatica 33(4), 351–385. 1996

[31] Sears, R. and Ramakrishnan, R. bLSM: a general purpose log structured merge tree. SIGMOD Conference (2012), 217-228.

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

[33] Tan, C. C., Sheng, B., Wang, H., and Li, Q. Microsearch: A search engine for embedded devices used in pervasive computing. ACM Trans. Embed. Comput. Syst., 9(4):43:1-29, 2010.

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

[35] Tsiftes, N. and Dunkels, A. A database in every sensor. In Proc. of the 9th ACM Conf. on Embedded Networked Sensor Systems, SenSys'11, pp. 316-332, 2011.

[36] Vo, H.T., Wang, S., Agrawal, D., Chen, G. and Ooi, B.C.: Logbase: A scalable log-structured database system in the cloud. Proc. of the VLDB Endowment, 5(10):1004–1015, 2012.

[37] Wang, B. and Baras, J.S.: Hybridstore: An efficient data management system for hybrid flash-based sensor devices. In Proceedings of the 10th European Conference on Wireless Sensor Networks, EWSN'13, pages 50–66, Berlin, Heidelberg, 2013. Springer-Verlag.

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

[39] Wu, C.-H., Kuo, T.-W., and Chang, L.-P. An efficient b-tree layer implementation for flash-memory storage systems. ACM Trans. Embed. Comput. Syst., 6(3), 2007.

[40] Yan, T., Ganesan, D., and Manmatha, R. Distributed image search in camera sensor networks. In Proc. of the 6th ACM Conference on Embedded Network Sensor Systems, SenSys'08, pp. 155–168, 2008.

[41] 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):26:1-34, 2008.

[42] Zobel, J. and Moffat, A. Inverted files for text search engines. ACM Comput. Surv., 38(2), 2006.