# MILo-DB: a personal, secure and portable database machine

**Nicolas Anciaux · Luc Bouganim · Philippe Pucheral · Yanli Guo · Lionel Le Folgoc · Shaoyi Yin**

**Abstract** Mass-storage secure portable tokens are emerging and provide a real breakthrough in the management of sensitive data. They can embed personal data and/or metadata referencing documents stored encrypted in the Cloud and can manage them under holder's control. Mass on-board storage requires efficient embedded database techniques. These techniques are however very challenging to design due to a combination of conflicting NAND Flash constraints and scarce RAM constraint, disqualifying known state of the art solutions. To tackle this challenge, we proposes a log-only based storage organization and an appropriate indexing scheme, which (1) produce only sequential writes compatible with the Flash constraints and (2) consume a tiny amount of RAM, independent of the database size. We show the effectiveness of this approach through a comprehensive performance study.

Communicated by Elena Ferrari.

N. Anciaux (✉) · L. Bouganim · P. Pucheral · Y. Guo · L. Le Folgoc
INRIA Paris-Rocquencourt, Le Chesnay, France
e-mail: Nicolas.Anciaux@inria.fr

L. Bouganim
e-mail: Luc.Bouganim@inria.fr

P. Pucheral
e-mail: Philippe.Pucheral@inria.fr

Y. Guo
e-mail: Yanli.Guo@inria.fr

L. Le Folgoc
e-mail: Lionel.LeFolgoc@inria.fr

N. Anciaux · L. Bouganim · P. Pucheral · Y. Guo · L. Le Folgoc
PRISM Laboratory, University of Versailles Saint-Quentin-En-Yvelines, Versailles, France

S. Yin
University of Cergy Pontoise, Cergy-Pontoise, France
e-mail: Shaoyi.Yin@u-cergy.fr

**Keywords** Embedded database · Secure and privacy aware data management · Secure chip · Flash memory · Tiny RAM · Log-only database structure

## 1 Introduction

As any citizen today, Alice receives salary forms, invoices, banking statements, etc., through the Internet. In her everyday life, she continuously interacts with several electronic devices (at home, at work, at hospital, while driving or shopping, taking photos, etc.) acquiring and producing large volumes of personal data. Alice would like to store this mass of data securely, share it with friends and relatives and benefit from new, powerful, user-centric applications (e.g., budget optimization, pay-per-use, health supervision, e-administration procedures and others Personal Information Management applications). Storing this data in a well organized, structured and queryable *personal database* [3, 17] is mandatory to take full advantage of these applications. Recently, KuppingerCole,[1] a leading security analyst company promotes the idea of a "*Life Management Platform*", a "*new approach for privacy-aware sharing of sensitive information, without the risk of losing control of that information*". Several projects and startups (e.g., QiY.com, Personal.com, Project VRM[2]) are pursuing this same objective.

But are there effective technological solutions matching this laudable objective? Any solution where Alice's data is gathered on her own computer would be very weak in terms of availability, fault tolerance and privacy protection against various forms of attacks. Solutions based on third party storage providers (e.g., in the Cloud) nicely answer the availability and fault tolerance requirements but the price to pay for Alice is loosing the control on her data.[3] This fear is fed by recurrent news on privacy violations resulting from negligence, abusive use and internal or external attacks (see e.g., DataLossDB.org). Should Alice follow the FreedomBox initiative [25] which promotes the idea of a small and cheap data server that each individual can plug on her Internet gateway? She will avoid any risk linked to the centralization of her data on remote servers. But what are the real security and availability guarantees provided by her plug computer?

The Personal Data Server vision [3] promotes an idea similar to FreedomBox, that is providing a fully decentralized infrastructure where personal data remains under holder's control, but with stronger guarantees. It builds upon the emergence of new devices combining the tamper resistance of a secure microcontroller [12] with the storage capacity of NAND Flash chips. Tamper-resistance provides tangible security guarantees missing to traditional plug computers. Availability is provided by replicating data in remote encrypted archive. These devices have different form factors (e.g.,

[1] http://www.kuppingercole.com/.

[2] http://cyber.law.harvard.edu/projectvrm/Main_Page.

[3] A recent Microsoft survey states that "*58 percent of the public and 86 percent of business leaders are excited about the possibilities of cloud computing. But more than 90 percent of them are worried about security and privacy of their data as it rests in the cloud*" http://news.cnet.com/8301-1009_3-10437844-83.html.

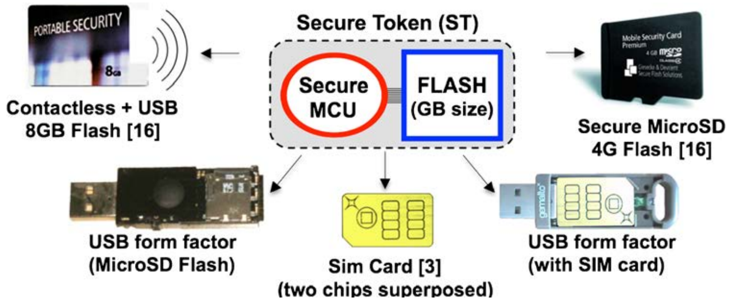

Secure Token (ST)
Secure MCU
FLASH (GB size)
Contactless + USB 8GB Flash [16]
Secure MicroSD 4G Flash [16]
USB form factor (MicroSD Flash)
Sim Card [3] (two chips superposed)
USB form factor (with SIM card)
PORTABLE SECURITY

**Fig. 1** Different form factors for Secure Tokens (STs)

SIM card, USB token, Secure MicroSD [18]) and names (e.g., Personal Portable Security Device [20], Smart USB Token [16], Portable Security Token [18] or Secure Portable Token [3]).

Despite this diversity (see Fig. 1), Secure Tokens (STs) share similar characteristics (low-power, cheap, highly portable, highly secure), holding the promise of a real breakthrough in the management of personal data.

Capitalizing on the Personal Data Server vision, we could devise an effective secure *Life Management Platform* where personal data is stored either locally (in the ST Flash memory) or remotely (e.g., in the Cloud). In the latter case, ST manages only metadata (description and location attributes, keywords, encryption keys, etc.) of encrypted documents stored remotely. Document sharing can be obtained by sharing the corresponding metadata under the ST holder's control, thereby providing ultimate security and privacy to cloud data [1]. The amount of data/metadata to be managed by the ST can be rather huge, embracing potentially the complete digital history of an individual. The heart of the Personal Data Server vision is then a ST turned into a personal database machine able to manage and share personal data under the control of its owner, with strong security guarantees inherited from tamper resistant hardware. More precisely, turning this vision into reality requires solving a core technical challenge, that is designing a DBMS engine embedded in a ST providing a least basic storage, indexing and query facilities with acceptable performance. This challenge holds whatever be the external resources the ST may be connected to (network, cloud, local host, etc.) since this is key to securely manage and share personal data.

*Why is this challenging?* STs provide unquestionable benefits (tamper-resistance, robustness, portability, low cost, low energy consumption), but have inherent hardware constraints. A ST is primarily made of a secure microcontroller (SMCU) connected to an external (i.e., unsecure) mass storage NAND Flash memory. Relatively speaking, SMCU have a powerful architecture (e.g., a 32-bit RISC processor, clocked at about 50 MHz, a cryptographic co-processor and internal stable storage up to 1 MB to store the embedded code and sensitive metadata). Actually, the main ST constraints are twofold. First, SMCU has a tiny RAM (at most 64 KB today). According to SMCU manufacturers, RAM will unfortunately remain a scarce resource in the foreseeable future due to its very poor density. Indeed, the smaller the silicon die, the more difficult it is to snoop or tamper with its processing. Hence, the RAM capacity increases much slowly than the stable storage capacity, worsening the ratio RAM/stable storage over time. Second, the NAND Flash stable store (usually connected by a bus to the SMCU) badly support random writes and cannot benefit from the SMCU tamper resistance. This leads to contradictory objectives: executing queries with acceptable performance with a tiny RAM entails indexing massively the database, while index updates generate fine-grain random writes, then unacceptable NAND Flash write costs and cryptographic overhead. The goal is thus to design an embedded DBMS engine that accommodates the tiny RAM constraints and manage large volume of data stored, in an encrypted form on Flash, without generating any random write.

The contribution of this paper is threefold:

(1) We show that state of the art database technologies cannot tackle the conjunction of ST hardware constraints and propose a *log-only* based database organization, producing only sequential writes matching these constraints.
(2) We show that massive indexing is mandatory and propose the design of indexes compatible with the log-only approach.
(3) We show how to combine these contributions to build a complete embedded DBMS engine, and notably, how to efficiently ensure its security. We then evaluate the performance of a resulting prototype named MILo-DB (Massively Indexed Log-only DB).

The rest of the paper is organized as follows. Section 2 analyzes the related works and states the problem to be solved. Section 3 gives a global overview of the log-only approach. Section 4 is devoted to the design of our indexed scheme and Sect. 5 to the way scalability is achieved. Section 6 sketches the remaining elements of the MILo-DB engine and Sect. 7 assesses the performance of this engine. Finally, Sect. 8 concludes.

## 2 Related work and problem formulation

This section briefly reviews works done in the embedded DBMS context. Then, it concentrates on works related to the main dimensions of the problem tackled in this paper, namely RAM conscious indexing schemes, Flash constraints management, Flash-based indexing schemes and log-structured indexing schemes. It concludes by stating the contradictory objectives pursued by these different approaches and formulates the main challenge to be solved.

*Embedded DBMSs* Embedded DBMS products (e.g., SQLite, BerkeleyDB) and light versions of popular DBMSs (e.g., DB2 Everyplace, Oracle Database Mobile Server) target small but relatively powerful devices (e.g., PDA, smart phone, set top box), far from the constrained hardware considered in this paper. Typically, they address neither the limitations of SMCUs (in particular the very tiny RAM) nor the specificities of NAND Flash. Proposals dedicated to databases embedded on SMCUs [11, 28] consider small databases stored in the SMCU internal stable storage—hundreds of kilobytes—and rely on NOR Flash or EEPROM technologies, both having a different behavior compared to NAND Flash (e.g., byte access granularity).

*Massive indexing schemes* The performance of SQL queries involving Joins and/or Aggregate computations declines sharply when the ratio between the RAM size and the size of the data to be processed falls below a certain threshold. In particular, "last resort" Join algorithms (block nested loop, sort-merge, Grace hash, hybrid hash) quickly deteriorate when the smallest join argument exceeds the RAM size [19]. To give a rough idea, with 10 KB of RAM, joining tables of 100 KB each using the block nested loop Join algorithm[4] may lead to read the second table ten times, the third table one hundred times, etc., which would be far inefficient. More recent algorithms like Jive join and Slam join use join indices [22] but both require that the RAM size is of the order of the square root of the size of the smaller table. In our context, the ratio between the RAM size and the tables' size is so small that the unique solution is to resort to a highly indexed model where all (key) joins are precomputed, as already devised in the Data Warehouse (DW) context. To deal with Star queries involving very large Fact tables (hundreds of GB), DW systems usually index the Fact table on all its foreign keys to precompute the joins with all Dimension tables, and on all Dimension attributes participating in queries [32, 34]. However, the consequence of massively indexing the database is generating a huge amount of fine-grain random writes at insertion time to update the indexes, in turn resulting in an unacceptable write cost in NAND Flash.

*NAND flash behavior and flash translation layers* NAND Flash memory is badly adapted to fine-grain data (re)writes. Memory is divided into blocks, each block containing (e.g., 64) pages themselves divided into (e.g., 4) sectors. The write granularity is the page (or sector) and pages must be written sequentially within a block. A page cannot be rewritten without erasing the complete block containing it and a block wears out after about $10^4$ repeated write/erase cycles. The technology trend is to increase the density of Flash (e.g., MLC vs. SLC), thereby ever worsening these constraints. To tackle Flash constraints, updates are usually managed out of place with the following side effects: (1) *a Translation Layer (TL)* is introduced to ensure the address invariance at the price of traversing/updating indirection tables, (2) *a Garbage Collector (GC)* is required to reclaim stale data and may generate moves of valid pages before reclaiming a non empty block and (3) *a Wear Leveling mechanism (WL)* is required to guarantee that blocks are erased evenly. A large body of work (see [20]) has focused on the design of Flash Translation Layers (FTLs), i.e., the conjunction of TL, GC and WL. However, with scarce RAM, FTL cannot hide NAND Flash constraints without large performance degradation.[5] Typically, random writes are two or three orders of magnitude more costly than sequential writes on SD cards.[6] In addition, FTL are black box firmwares with behaviors difficult to predict and optimize. For all these reasons, we argue that delegating the optimization of the Flash usage to

[4]Bloc nested loop Join is often the only Join algorithm provided in embedded DBMS products (e.g., for SQLite see http://www.sqlite.org/optoverview.html).

[5]This is not the case in high-end SSDs which can use relatively large RAM (e.g., 16 MB) to handle those constraints.

[6]Tests on 20 recent SD cards have shown that random writes are in average 1300 times more costly than sequential writes (min 130$\times$, max 5350$\times$) [30].

a FTL does not make sense in our context. Our design considers that the SMCU has direct access to the NAND Flash[7] but can accommodate Flash access through FTL with minimal extension.

*Indexing techniques for NAND flash* Many studies address the problem of storage and indexing in NAND Flash. Conventional indexes like B+-Tree perform poorly on top of FTL [35]. Most of the recent proposals [2, 9, 21, 35] adapt the traditional B+-Tree by relying on a Flash resident log to delay the index updates. 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 RAM index are managed, and in the impact it has on the commit frequency. To amortize the write cost by large factors, the log is seldom committed, leading to consume more RAM. Conversely, limiting the RAM size means increasing the commit frequency, thus generating more random writes. RAM consumption and random write cost are thus conflicting parameters. Under the RAM limitations of SMCUs the commit frequency becomes de facto very high and the gain on random writes vanishes. PBFilter [36] is an indexing scheme specifically designed for Flash storage in the embedded context. It organizes the index in a sequential way thereby avoiding random writes. The index is made of a key list compacted using a Bloom filter summary, which can further be partitioned. This leads to good lookup times with very few RAM. However, PBFilter is designed for primary keys. With secondary keys the Bloom filter summary becomes non selective and mainly useless (the complete set of keys has to be accessed). This makes PBFilter of little interest for implementing massive index schemes, since most indexes are secondary keys.

*Log-structured indexes* Many proposals for log-structured indexes [7, 26, 27] are based on principles from log-structured file systems [29] (also used in the design of FTLs to hide NAND Flash constraints) or older principles like *Differential File* [31], an overflow area storing new records, merged with the main data file at some future point. These proposals differ in the way indexes are built or maintained but they always make use of relatively large buffers in RAM incompatible with our constraints. More recently, [8] proposed Hyder, a log-structured, multiversion key-value database, stored in flash memory, and shared over the network. Hyder makes use of a single binary balanced tree index 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 massive indexing schemes (binary trees are not adequate to index non unique keys). Still in the key-value store context, SkimpyStash [14], LogBase [33] and SILT [23] organizes key-value pairs in a log structure to exploit sequential writes and maintain some form of in-memory (RAM) indexing with a size proportional to the database size, thus consuming too much RAM for a SMCU (at least 1 Byte per record).

[7] A first layer (the Hardware Adaptation Level) of the controller software manages Low Level Drivers (LLD), Error Correction (ECC) and Bad Block Management (BBM). The second layer is the FTL, and it can be bypassed on most platforms.

*Problem formulation* To conclude, the challenge tackled in this paper lies in the combination of a tiny working memory (RAM) with a huge NAND Flash mass storage badly accommodating random writes. Executing queries with acceptable performance on gigabytes of data with a tiny RAM entails indexing massively the database. The consequence is generating a huge amount of fine-grain random writes at insertion time to update the indexes, which in turn results in an unacceptable write cost in NAND Flash. Conversely, known solutions to decrease the amount of random writes in Flash require a significant amount of RAM. A vicious circle is then established and lets little hope to build an embedded DBMS engine by assembling state of the art solutions. The objective of our work is to break this circle.

## 3 Log-only database organization

To tackle the problem stated above, we propose a new database organization called *Log-Only* trying to reconcile three a priori contradictory objectives: (1) massively indexing the database, (2) producing only sequential writes in Flash and (3) consuming a tiny amount of RAM, independent of the database size. To this end, we propose to organize the complete database (raw data but also indexes, buffers and more generally all data structures managed by the DBMS engine) into *Log Containers*. Log containers are purely sequential persistent recipients.

*Log Container (LC) A LC is a data structure satisfying three conditions: (1) its content is written sequentially within the Flash block(s) allocated to it (i.e., pages already written are never updated nor moved); (2) blocks can be dynamically added to a LC to expand it; (3) a LC is fully reclaimed when obsolete (no partial garbage collection occurs).*

The net effect of organizing the complete database into Log Containers is to avoid random writes by definition. Hence, the dramatic overhead of random writes in Flash is avoided.[8] However, processing sequential structures do not scale well also by definition. Hence, to make the database scalable, the initial sequential database must be iteratively reorganized into more efficient data structures. The resulting data structures must be produced in Log Containers as well to preserve the initial objective. The way a sequential database can be initially produced and then reorganized according to log-only constraints is sketched below.

### 3.1 Log-only initial database

*Base data* Natural log-only solutions can be easily devised to organize the base data (*tables*). Typically, a table can be stored as a sequence of rows in a Row Store scheme or as a set of sequences of attribute values in a Column Store one. Adding new base data is direct in both cases. ↓DATA (↓ stands for log-only) denotes the LCs dedicated to base data.

[8]Moreover, the Flash Translation Layer becomes useless (thereby saving translation costs) and the garbage collection and wear leveling mechanism can be greatly simplified.

*Indexes* While particular index structures like *bitmap indexes* easily comply with the LC definition, most classical indexes (e.g., tree-based or hash-based) are proscribed. Indeed, inserting new base data would generate random node/bucket updates. In Sect. 4, we propose new forms of indexes compatible with LCs to speed up joins and selections. We denote by $\downarrow$IND the LCs dedicated to such indexes.

*Updates and deletes* When updating or deleting base data, directly reporting modifications in $\downarrow$DATA would violate the LC definition. Instead, updates and deletes are logged in dedicated LCs, respectively named $\downarrow$UPD and $\downarrow$DEL. To manage updates, the old and new attribute values of each updated tuple are logged in $\downarrow$UPD. At query execution time, $\downarrow$UPD is checked to see whether its content may modify the query result. First, if a logged value matches a query predicate, the query is adjusted to eliminate false positives (i.e., tuples matching the query based on their old value but not on their new value) and to integrate false negatives (i.e., tuples matching the query based on their new value but not on their old value). Second, $\downarrow$UPD and $\downarrow$DEL are also checked at projection time, to project up-to-date values and to remove deleted tuples from the query result. An important remark is that $\downarrow$DEL only records the tuples that have been explicitly deleted (i.e., it does not records cascaded deletes, that is deletes of referencing tuples). This does not strongly impact query execution since join indexes are created anyway and can be accessed efficiently when evaluating a query (this access is even mandatory for join queries as explained in the last paragraphs of Sect. 4.1). Overheads are minimized by indexing $\downarrow$UPD and $\downarrow$DEL on Flash (similarly as indexing a table), and building dedicated data structures in RAM to avoid accessing Flash for each result tuple.[9]

*Buffers and transactions* $\downarrow$DATA, $\downarrow$IND, $\downarrow$UPD, $\downarrow$DEL being made of fine grain elements (tuples, attribute values, pointers, or index entries), inserting data into LCs without buffering would lead to waste a lot of space in Flash. The objective of buffering is to transform fine-grain writes (e.g., attributes, pointers) to coarse-grain writes (a full flash page) in LCs. To this end, we implement buffers themselves by means of LCs denoted by $\downarrow$BUF. Roughly speaking, fine-grain elements are gathered into buffers until a full page of those elements can be built. The way $\downarrow$BUF is actually managed and the way transaction atomicity (to undo dirty insertions into LCs) is enforced is more deeply discussed in Sect. 6.

The log-only database is denoted by $\downarrow$DB and is thus composed of the LCs of buffers, base data, indexes, update and delete logs.

The log-only database principle leads to a robust and simple design, compliant with all ST constraints. However, such a design scales badly since $\downarrow$IND cannot compete with classical indexes (e.g., B+-Tree), and the accumulation over time of elements in $\downarrow$UPD and $\downarrow$DEL will unavoidably degrade query performance. There is a scalability limit (in terms of $\downarrow$DATA, $\downarrow$IND, $\downarrow$UPD and $\downarrow$DEL size) beyond which performance expectation will be violated, this limit being application dependent.

[9] While the strategy for handling deletes and updates is rather simple, the details on query compensation is a bit tricky and cannot be included in the paper due to size constraint.

## 3.2 Log-only reorganizations of the database

To tackle this scalability issue, we reorganize $\downarrow$DB into another **optimal* log-only database denoted by *DB. By **optimal*, we mean that the performance provided by *DB is at least as good as if *DB were built from scratch by a state of the art method ignoring Flash constraints. For example, the reorganization of $\downarrow$DB into *DB can result in a set of tables where all updates and deletes are integrated and which are indexed by tree-based or hash-based indexes. Note that the reorganization process presented below is independent of the indexing scheme selected for *DB.

For the sake of clarity, we introduce a reorganization counter, named *instance,* added in subscript in the notations. Before any reorganization occurs, the initial log-only database is

$$\downarrow\mathrm{DB}_0 = (\downarrow\mathrm{BUF}_0, \downarrow\mathrm{DATA}_0, \downarrow\mathrm{IND}_0, \downarrow\mathrm{UPD}_0, \downarrow\mathrm{DEL}_0).$$

When the scalability limit of $\downarrow\mathrm{DB}_0$ is reached, $*\mathrm{DB}_0$ needs to be built. The reorganization process triggers 3 actions.

$\downarrow\mathrm{BUF}_0$ elements are flushed into their target LC (i.e., $\downarrow\mathrm{DATA}_0$, $\downarrow\mathrm{IND}_0$, $\downarrow\mathrm{UPD}_0$, $\downarrow\mathrm{DEL}_0$).

All new insertions, updates and deletes are directed to $\downarrow\mathrm{DB}_1 = (\downarrow\mathrm{BUF}_1, \downarrow\mathrm{DATA}_1, \downarrow\mathrm{IND}_1, \downarrow\mathrm{UPD}_1, \downarrow\mathrm{DEL}_1)$ until the next reorganization phase ($\downarrow\mathrm{DB}_1$ is initially empty).

$\downarrow\mathrm{DB}_0$ (which is then frozen) is reorganized into $*\mathrm{DB}_0$, composed of $*\mathrm{DATA}_0$ and $*\mathrm{IND}_0$.$*\mathrm{DATA}_0$ is built by merging $\downarrow\mathrm{DATA}_0$ with all updates and deletes registered in $\downarrow\mathrm{UPD}_0$ and $\downarrow\mathrm{DEL}_0$ (*Merge* operation). $*\mathrm{IND}_0$ is the **optimal* reorganization of $\downarrow\mathrm{IND}_0$, including modifications stored in $\downarrow\mathrm{UPD}_0$ and $\downarrow\mathrm{DEL}_0$ (*ReorgIndex* operation).

The database is then composed of $\downarrow\mathrm{DB}_1$ (receiving new insertions, updates and deletes) and of $*\mathrm{DB}_0$, itself composed of $*\mathrm{DATA}_0$ and $*\mathrm{IND}_0$. When the reorganization process terminates (i.e., $*\mathrm{DATA}_0$ and $*\mathrm{IND}_0$ are completely built), all LCs from $\downarrow\mathrm{DB}_0$ can be reclaimed. $\downarrow\mathrm{DB}_1$ keeps growing until the next reorganization phase. The next time the scalability limit is reached, a new reorganization occurs. Reorganization is then an iterative mechanism summarized in Fig. 2 (without $\downarrow$BUF for the sake of clarity).

```
$\boldsymbol{Reorg(\downarrow DB_i) \rightarrow *DB_i}$
    $\downarrow DB_{i+1} = \varnothing$
    $Flush(\downarrow BUF_i) in \downarrow DATA_i, \downarrow IND_i, \downarrow UPD_i, \downarrow DEL_i$
    $Reclaim(\downarrow BUF_i)$
    $if\ (i > 0)$
        $*DATA_i = Merge(*DATA_{i-1}, \downarrow DATA_i, \downarrow UPD_i, \downarrow DEL_i)$
        $*IND_i = ReorgIndex(*IND_{i-1}, \downarrow IND_i, \downarrow UPD_i, \downarrow DEL_i)$
        $Reclaim(*DATA_{i-1}, *IND_{i-1})$
    $else$
        $*DATA_i = Merge(NULL, \downarrow DATA_i, \downarrow UPD_i, \downarrow DEL_i)$
        $*IND_i = ReorgIndex(NULL, \downarrow IND_i, \downarrow UPD_i, \downarrow DEL_i)$
        $Reclaim(\downarrow DATA_i, \downarrow IND_i, \downarrow UPD_i, \downarrow DEL_i)$
```

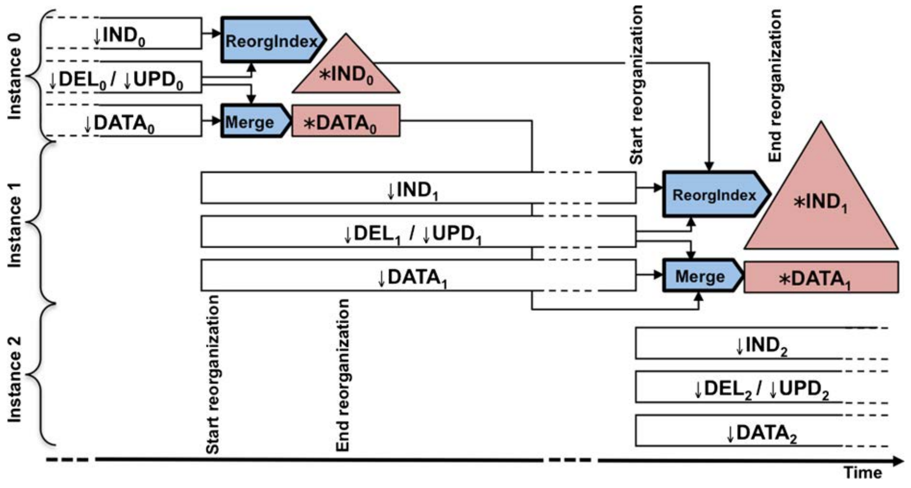


**Fig. 2** The log-only reorganization process

Remark that reorganization is very different in spirit from batch approaches deferring updates thanks to a log. Indeed, deferred updates produce random rewrites while log-only reorganizations produce only sequential writes into LCs. The price to pay is however a complete reconstruction of $*\mathrm{DB}_i$ at each reorganization.

## 4 Log-only indexes

In this section, we discuss the types of indexes that are required in a massively indexed context, and propose a design for constructing them (both ↓IND and *IND) in a way compliant with the log-only constraints. We concentrate on indexing schemes for simple Select/Project/Join/Aggregate queries. We consider this enough to validate the approach in the settings presented in the introduction and let the study of more complex queries for future work.

### 4.1 Which indexes?

Let us first consider Select/Project/Join queries (SPJ). The very small ratio between RAM and database size leads to use generalized selection and join indexes [5, 28, 32, 34]. These indexes are called generalized in the sense that they capture the transitive relationships which may exist between tuples of different tables. In other words, we say that a tuple $t$ of table $\mathrm{T}_i$ *references* a tuple $t'$ of table $\mathrm{T}_j$ (denoted by $t \rightarrow t'$), if $t$ is linked directly or transitively to $t'$ by a join path on foreign/primary keys (i.e., starting with a foreign key of $t$ and ending with $t'$ primary key). On this basis, generalized indexes can be defined as follows:

- **Index $\mathrm{I}_{\mathrm{T}_i \rightarrow \mathrm{T}_j}$ (TJoin)**: *For a given tuple $t$ of $\mathrm{T}_i$, index $\mathrm{I}_{\mathrm{T}_i \rightarrow \mathrm{T}_j}$ returns the identifiers of all tuples of $\mathrm{T}_j$ referenced by $t$*: $\mathrm{I}_{\mathrm{T}_i \rightarrow \mathrm{T}_j}(t \in \mathrm{T}_i) = \{t'.id / t' \in \mathrm{T}_j$ *and* $t \rightarrow t'\}$.

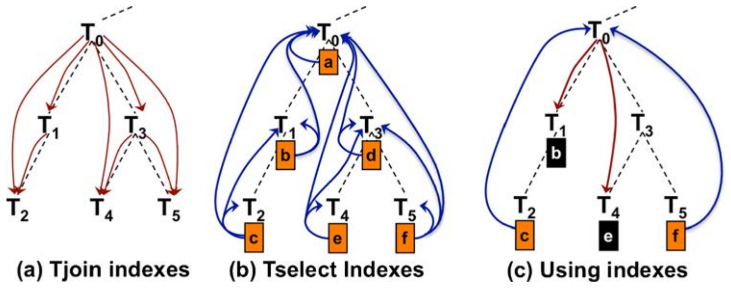


**Fig. 3** The massive indexation scheme and its usage

This index is actually a one-way generalized join index, precomputing natural joins from referencing to referenced tables. Let us call this class of indexes TJoin (Transitive Join).

- **Index $\mathbf{I}_{\mathbf{T}_j.\mathbf{A}\rightarrow\mathbf{T}_i}$ (TSelect)**: *Given a value $v$ from the domain of attribute* $\mathrm{T}_j$.A, *this index returns the identifiers of each tuple $t$ of* $\mathrm{T}_i$ *such that $t$ references a tuple $t'$ of* $\mathrm{T}_j$ *where* $t'.\mathrm{A} = v$: $\mathrm{I}_{\mathrm{T}_j.\mathrm{A}\rightarrow\mathrm{T}_i}$ $(v \in dom(\mathrm{T}_j.\mathrm{A})) = \{t.id/t \in \mathrm{T}_i, t' \in \mathrm{T}_j$ *and* $t \rightarrow t'$ *and* $t'.\mathrm{A} = v\}$.

Note that the result must be ordered on $t.id$ to allow merging (by unions or intersections) sorted lists of t.id with no RAM. In the particular case where $i = j$, $t = t'$ and this index acts as a regular selection index (in other words, each tuple references itself). If $i \neq j$, it implements a selection in a referencing table on an attribute from a referenced table (just as indexing a Fact table on attributes of a dimension table). Let us call this class of indexes TSelect (Transitive Select).

For instance, on the partial database schema presented in Fig. 3a, $\mathrm{T}_0$ references directly $\mathrm{T}_1$ and $\mathrm{T}_3$ and transitively $\mathrm{T}_2$, $\mathrm{T}_4$ and $\mathrm{T}_5$. 8 TJoin indexes are created in this example. Figure 3b shows TSelect indexes, considering that a single attribute in each table $\mathrm{T}_i$ is indexed. For instance, for attribute $c$ of table $\mathrm{T}_2$, we create 3 such indexes: $\mathrm{I}_{\mathrm{T}_2.c\rightarrow\mathrm{T}_2}$, $\mathrm{I}_{\mathrm{T}_2.c\rightarrow\mathrm{T}_1}$, $\mathrm{I}_{\mathrm{T}_2.c\rightarrow\mathrm{T}_0}$.

More generally, to allow an efficient computation of all SPJ queries with a tiny RAM, we consider a massive indexing scheme in the same line as [5, 28, 32, 34]. This scheme is composed by (1) a TJoin index for each ($\mathrm{T}_i$, $\mathrm{T}_j$) such that $\mathrm{T}_i$ references $\mathrm{T}_j$[10] and (2) a TSelect index for each attribute potentially involved in selection predicates and for each ($\mathrm{T}_i$, $\mathrm{T}_j$) such that $\mathrm{T}_i$ references $\mathrm{T}_j$.

Intuitively, SPJ queries are executed by (1) traversing the relevant TSelect indexes, i.e., the $\mathrm{I}_{\mathrm{T}_j.\mathrm{A}\rightarrow\mathrm{T}_i}$ indexes corresponding to all predicates of the form $\mathrm{T}_j$.A in the query and such that $\mathrm{T}_i$ is a common table for all indexes, (2) merging the sorted sets of $\mathrm{T}_i$ tuple identifiers in pipeline (by intersection and/or union); and (3) traversing the relevant TJoin indexes to project the resulting tuples. For instance, Fig. 3c shows the strategy for computing a query which joins all tables, evaluates the selection predicates ($\mathrm{T}_2.c = v1$ and $\mathrm{T}_5.f = v2$) and projects attributes $\mathrm{T}_1.b$ and $\mathrm{T}_4.e$. This leads to: (1) lookup in $\mathrm{I}_{\mathrm{T}_2.c\rightarrow\mathrm{T}_0}(v1) \rightarrow S1$ and $\mathrm{I}_{\mathrm{T}_5.f\rightarrow\mathrm{T}_0}(v2) \rightarrow S2$, (2) intersect sorted sets

[10]For the sake of clarity, we make here the assumption that at most one join path exists between two tables (e.g., a snowflake schema). The indexing scheme can be extended trivially to the multiple paths case.

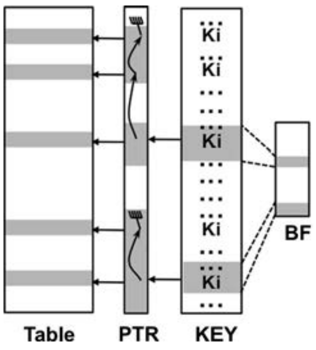


**Fig. 4** TSelect index design in ↓DB

$S1$ and $S2$ in pipeline (3) use indexes $I_{T_0 \to T_1}$ and $I_{T_0 \to T_4}$ to find resulting tuples and project attributes $T_1.b$ and $T_4.e$.

Queries with Group By clauses and aggregates are more complex to execute since the RAM consumption is linked with the number of groups in the results. In that case, the result of the SPJ part of the query is stored in a temporary Log Container. Then the whole RAM is used to perform the aggregation in several steps by computing a fraction of the result at each iteration. Several strategies have been proposed in [5] in another context and can be applied directly here.

## 4.2 TJoin index design

For a given table $T_i$, a set of TJoin indexes must be created, one for each referenced table. All those indexes can be stored in the same Log Container to save IOs when inserting tuples into $T_i$ since they are filled exactly at the same rate. Note that storing these indexes separately would bring marginal IOs savings at query time. Indeed, TJoin indexes are generally not accessed sequentially: they are accessed after evaluating the selection predicates using TSelect indexes which return sets of tuples identifiers. Hence, the set of TJoin indexes of table $T_i$ can be represented as a single regular table with as many columns as they are tables referenced by $T_i$. Such as TJoin table is naturally ordered on $T_i$ tuple identifiers, following the insertion order in $T_i$. Access to this table is direct, using $T_i$ tuple identifiers as positions. At insertion time, the direct references are part of the inserted tuple (foreign keys), the transitive ones being retrieved by accessing the TJoin indexes of referenced tables (e.g., in Fig. 3a., the set $I_{T_0 \to T_j}$ is filled by accessing $I_{T_1 \to T_2}$, $I_{T_3 \to T_4}$ and $I_{T_3 \to T_5}$). Since TJoin indexes behave as normal tables, they are stored as regular tables.

## 4.3 TSelect index design

As pictured in Figs. 4 and 5, TSelect indexes have a more complex structure to comply with the log-only constraints. We first describe the general organization of these indexes, and then explain their usage (lookup and maintenance operations). Considering a TSelect index I, we denote by ↓I the log-only initial structure of this index and by *I its **optimal* reorganization.

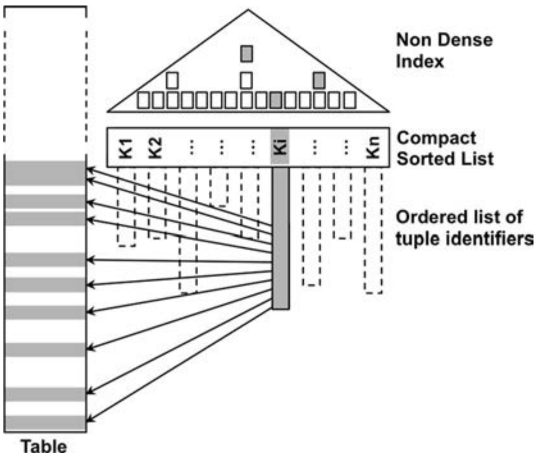


**Fig. 5** TSelect index design in *DB

### *4.3.1 Organization of TSelect indexes in ↓DB*

Three Log Containers are used to materialize an index $I_{T_j.A \rightarrow T_i}$. A first container called KEY is used to sequentially store, for each new tuple $t$ of $T_i$, the $T_j$.A value of the referenced tuple (this key can be efficiently obtained by accessing the appropriate $I_{T_i \rightarrow T_j}$ index using the tuple identifier of $t$). Hence, KEY is a dense index which must be sequentially scanned when looking up to a key. To decrease this cost, we store a compressed representation of KEY, obtained using *Bloom filters* [10], in a second container called BF. A Bloom filter represents a set of values in a compact way and allows probabilistic membership queries with no false negatives and a very low rate of false positive.[11] One new Bloom filter is built for each new Flash page of KEY and is sequentially appended into BF. At lookup time BF is sequentially scanned and a page of KEY is accessed only in case of a match in a Bloom filter. The false positive rate being rather low, the IO cost is roughly decreased by the compression factor of Bloom Filters. To further reduce the lookup cost, index entries sharing the same key value are chained by pointers in a third container, called PTR (to comply with the log-only constraints, this chaining is done backward). Hence, only the head of the chain corresponding to the searched value is done sequentially through BF then KEY. Then, a direct access to PTR is performed to access the pointer chain and follow it to retrieve all other matching tuples' identifiers.

The benefit incurred by PTR comes at the expense of maintaining the pointer chains. Each time a new tuple is inserted, the head of the chain corresponding to its key must be found to be able to expand the chain. While this search is improved by BF, it may become expensive when inserting a tuple having a "rare" key (i.e., a value with few occurrences in KEY). To bound this overhead, the chain is broken if this head is not found after a given threshold (e.g., after having scanned $n$ BF pages) and the new key is simply inserted with a preceding pointer set to NULL. At lookup time,

[11]For example, the false positive rate using 4 hash functions and allocating 16 bits per value is 0.24 % [10]. Hence, Bloom Filters provide a very flexible way to trade space with performance.

when a NULL pointer is encountered, the algorithm switches back to BF, skipping the $n$ next pages of BF, and continues searching the next key occurrence using BF and KEY. A positive consequence of this scheme is that the pointers can be encoded on a smaller number of bits (linked to the number of keys summarized by $n$ pages of BF) thereby reducing the access cost of PTR.

### *4.3.2 Organization of TSelect indexes in *DB*

The construction of *I takes advantage of three properties: (1) *I is never updated and can therefore rely on more space and time efficient data structures than traditional $B^+$-Tree (100 % of space occupancy can be reached compared to about 66 % in B-Tree nodes); (2) at the time the reorganization occurs, all the data items to be indexed are known; (3) the whole RAM can be dedicated to index construction since reorganization can be stopped/resumed with low overhead (see Sect. 5). The constraint however is to build *I in such a way that the lists of tuple identifiers associated to the index entries are kept sorted on the tuples insertion order. This requirement is mandatory to be able to merge efficiently lists of identifiers from multiple indexes. The resulting reorganized index structure is depicted in Fig. 5. A first Log Container contains the sets of compact *Ordered Lists of tuple Identifiers* built for each index key $K_i$. A second container stores the set of index entries, represented as a *Compact Sorted List*. Finally, the last container stores a compact *Non-Dense Index*, made of the highest key of each page of the compact sorted list in a first set of Flash pages, itself indexed recursively up to a root page. The index is built from the leaves to the root so that no pointers are required in the non-dense index. Reorganized indexes are then more compact and efficient than their traditional $B^+$-Tree counterpart. Note that the non-dense index and compact sorted list can be shared between several indexes indexing the same key ($T_j$.A).

### *4.3.3 Combining indexes in ↓DB and *DB*

Tuples of $*DB_{i-1}$ tuples always precede tuples of $\downarrow DB_i$ in their insertion order, and so are the values of their identifiers. To produce ordered tuple identifiers,[12] the lookup operation thus combines the results from $\downarrow I_i$ and $*I_{i-1}$ by a simple concatenation of the lists of matching tuples (see Fig. 6).

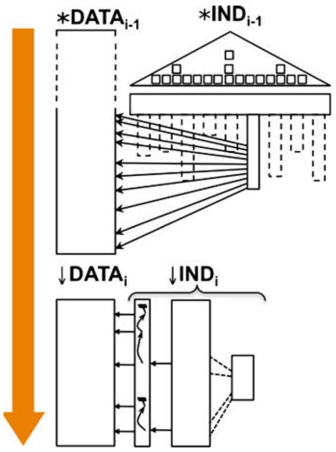
To conclude, the proposed index structure allows an efficient retrieval of the ordered list of identifiers with a bounded insertion cost. It answers all requirements of TSelect indexes while satisfying the log-only constraints assuming reorganization can be done efficiently.

## 5 Reorganization process

In our approach, reorganizing a database sums up to processing the *Merge* and *ReorgIndex* operations as introduced in Sect. 3.2. We focus here on the complex *ReorgIndex* operation since *Merge* is rather trivial (basically, checking in pipeline each

[12]The result tuple identifiers are produced in reversed order, from the most recently inserted to the least recent inserted, which is suitable with the definition of Tselect Index $I_{T_j.a \rightarrow T_i}$ given in Sect. 4.1.

**Fig. 6** Combining Indexes in ↓DB and *DB



tuple in ↓DATA and *DATA using summaries of ↓UPD and ↓DEL in RAM to avoid accessing it on Flash).

Different structures are involved in index reorganization (see Fig. 2). Let us first analyze their organization to infer an adequate reorganization strategy matching the ST constraints.

- *IND contains a set of TSelect indexes[13] sorted on the index key, each key being linked to an *ordered list of tuple identifiers* (see Sect. 4.3), thus each index is sorted on (*Key*, *ID*) where *Key* are index keys and *ID* are tuple identifiers.
- ↓IND contains a set of TSelect indexes, each sorted on tuple identifiers since it is built sequentially.
- ↓UPD and ↓DEL have not specific order since elements are added sequentially when updates and deletes are performed.

These structures being stored in different orders, the reorganization process appears to be complex, especially with large data structures and scarce RAM. Note that after some initial reorganizations, the size of *IND is much larger than ↓IND. In addition, the size of ↓DEL and ↓UPD is expected to be small relatively to ↓IND (i.e., much less updates and deletes than insertions in a traditional database). The idea is thus to adapt the ordering of ↓BUF, ↓UPD and ↓DEL to the one of *IND such that they can be merged in pipeline (i.e., without RAM consumption) to build the new *IND (denoted *newIND in this section, to avoid indices for better readability).

- *Adapting ↓IND*: Each ↓$I_{T_j.A \rightarrow T_i}$ must be sorted on (*Key*, *ID*) such that it will be sorted as the corresponding *$I_{T_j.A \rightarrow T_i}$. Sorting can be done by sort merge using the whole RAM and a temporary LC reclaimed at the end of the process.
- *Adapting ↓DEL*: To avoid querying ↓DEL for each tuple of ↓IND and *IND (a simple but highly costly solution), we compute beforehand the impact of deleted tuples on all TSelect indexes. This requires one query per deleted tuple but the impact on performance is limited given the reduced size of ↓DEL. Once ↓DEL is

[13]Since TJoin indexes behave as normal tables, they are handled in the same way by the *Merge* operations, and thus, are not discussed here.

correctly *expanded* (i.e., integrates cascade deletes and impact on all indexes using $\mathrm{I}_{\mathrm{T}_j.ID\to \mathrm{T}_i}$), the result is sorted on (*#Index, Key, ID*) where *#Index* is the identifier of TSelect Indexes $\mathrm{I}_{\mathrm{T}_j.\mathrm{A}\to \mathrm{T}_i}$. This is described in the pseudo-code below:

***Adapt(↓DEL) → D***

```
1. $Cascade(\downarrow DEL) \rightarrow D*$ // one query per deleted tuple to find cascade deletes
2. $Project\_on\_all\_indexed\_attributes(D*) \rightarrow D*\pi$ // using $I_{T_j}.ID \rightarrow T_i$
3. $Sort(D*\pi)$ on $(\#Index, Key, ID) \rightarrow D$
```

- *Adapting ↓UPD* : ↓UPD roughly follows the same processing as ↓DEL with two main differences. First, we need to build two structures, ↓UPD_FP for false positives, and ↓UPD_FN for false negatives. Second, ↓UPD impacts a more reduced number of indexes than ↓DEL (i.e., only those concerning the updated attributes) making this expansion less costly. As for ↓DEL, after expansion, the result is sorted on (*#Index, Key, ID*).

***Adapt(↓UPD) → (U_FP, U_FN)***

```
1. $Cascade(\downarrow UPD) \rightarrow U*$ // one query per updated tuple to compute the impact
                                              // on all TSelect indexes
2. $Sort(U*)$ on $(\#Index, OldValue, ID) \rightarrow U\_FP$ // False positive i.e. OldValue
                                                        // should be removed from indexes
3. $Sort(U*)$ on $(\#Index, NewValue, ID) \rightarrow U\_FN$ // False negative i.e. NewValue
                                                          // should be added to indexes
```

Once all structures have the same order, we can compute *newIND as described in the pseudo-code below:

***ReorgIndex(*IND, ↓IND, ↓UPD, ↓DEL) → *newIND***

```
4. $Adapt(\downarrow DEL) \rightarrow D$
5. $Adapt(\downarrow UPD) \rightarrow (U\_FP, U\_FN)$
6. For each $I_{T_j.A\rightarrow T_i}$ index
7. $Adapt(\downarrow I_{T_j.A\rightarrow T_i}) \rightarrow S$ // Sort on $(Key, T_i.ID)$
8. $Fusion(*I_{T_j.A\rightarrow T_i}, S, D, U\_FP, U\_FN) \rightarrow *newI_{T_j.A\rightarrow T_i}$
```

The *Fusion* operation merges in pipeline $*\mathrm{I}_{\mathrm{T}_j.\mathrm{A}\to \mathrm{T}_i}$ with S (the sorted version of $\downarrow\mathrm{I}_{\mathrm{T}_j.\mathrm{A}\to \mathrm{T}_i}$), removing deleted entries (D), false positives (U_FP) and adding false negatives (U_FN).

The reorganization is a long process (see Sect. 7.5) and thus should be designed such that it can be stopped and resumed with minimal overhead if the database is queried while a reorganization process is in progress. Actually, the reorganization process combines three types of operations:

- *Pipelined operations*: These operations (e.g, Fusion) can be stopped at very low cost since we only need to store the current state of the operation (i.e., pointers on all operands of the operation). A single page IO need be done (in an operation log).
- *Queries*: ↓UPD and ↓DEL *adaptation* triggers a set of queries to build their expanded versions. Each individual query is very simple (basically following TJoin indexes and projecting some individual values) and can then be completed before stopping the reorganization process.

- *Sort operations*: If queries are triggered when a sort operation is about to be performed (e.g., RAM has been loaded with data to be sorted), we can either abort this operation or complete it (i.e., sort the data and write it back to Flash in temporary LCs), thus delaying queries. The scarce RAM is, this time, an advantage. Indeed sorting 64 KB of data can be done in few ms, as well as storing the result on Flash (about 10 ms for 64 KB).

Thus, we can easily stop and resume the reorganization process, completing any atomic operation and logging the current state of the reorganization process in a single Flash IO. As a consequence, queries can be run while reorganization is incomplete. Note that this is not a problem because LCs from the previous database instance are reclaimed only at the end of the process and can then serve to answer queries.

The reorganization cost is related to the size of $\downarrow$DB, to the number of deletes and updates performed since last reorganization, and to the size of *DB. The essential features of the reorganization are: (1) the reorganized part of the database is read and written only once; (2) the process can be interrupted then resumed, without hindering queries on the database; (3) the process is by nature failure-resistant (e.g., sudden power loss) since no element of $\downarrow$DB is reclaimed before the process completion;[14] (4) since each index is reorganized independently, queries that may be triggered during reorganization can take advantage of already reorganized indexes.

## 6 Buffers, transaction management, security

The preceding sections have presented (1) the log-only approach underlying MILo-DB, (2) a massive indexing scheme based on generalized indexes and its log-only implementation and (3) the reorganization process ensuring the scalability of the whole. These contributions form the core of the MILo-DB design. This section gives some insights on the main other elements of this engine, namely the buffer management, the transaction management and the crypto-protection of the database. Taken together, these elements form a complete embedded DBMS engine capable of addressing the requirements of the targeted use-cases.

### 6.1 Buffer management

$\downarrow$DB structures are made of fine grain elements (tuples, attribute values or index entries). NAND Flash constraints however impose writing a new page for each new element if insertions are done one by one into those structures. This leads to waste a lot of space and to decrease query performance. Indeed, the density of a Log Container determines the efficiency of scanning it. Buffering strategies are then required to transform fine-grain writes in Flash to coarse-grain writes. Each new element targeting a given Log Container *LC* is first gathered into buffers until the set of buffered elements for this LC can fill a complete Flash page, which is then flushed.

[14] The work required to recover from a crash during the reorganization can be minimized by storing on Flash the current state of each operation and analyzing this log at recovery time.

Buffers cannot reside in RAM because of its tiny size, and also because we cannot assume electrical autonomy and no failure.[15] Hence, buffers must be saved in NAND Flash and the density of buffer pages depends on the transactions activity. To increase buffer density (and therefore save writes), elements targeting different LCs are buffered together if they are filled and flushed synchronously. For example, let us consider the Log Containers PTR, KEY and BF used in ↓IND. For a given table $\mathrm{T}_i$, all indexes of type $\mathrm{I}_{\mathrm{T}_j.\mathrm{A}\rightarrow\mathrm{T}_i}$ built over that table $\mathrm{T}_i$ can be buffered together. A new element is inserted into their respective PTR and KEY Log Containers at each insertion in table $\mathrm{T}_i$, and a new entry is inserted into their respective BF Log Containers synchronously each time a KEY page is filled. In Fig. 3b, this means that, at least, indexes $\mathrm{I}_{\mathrm{T}_1.b\rightarrow\mathrm{T}_0}$, $\mathrm{I}_{\mathrm{T}_2.c\rightarrow\mathrm{T}_0}$, $\mathrm{I}_{\mathrm{T}_3.d\rightarrow\mathrm{T}_0}$, $\mathrm{I}_{\mathrm{T}_4.e\rightarrow\mathrm{T}_0}$, and $\mathrm{I}_{\mathrm{T}_5.f\rightarrow\mathrm{T}_0}$ can be buffered together.

Remark also that buffers must be organized themselves as LCs to comply with the log-only constraints. We manage each buffer as a sliding window within its Log Container, using *Start* and an *End* markers to identify its active part (i.e., the part not yet flushed).

## 6.2 Transaction management

Regarding transaction atomicity, we restrict ourselves to a single transaction at a time (which makes sense in our context). Rolling-back a transaction, whatever the reason, imposes undoing all dirty insertions to the Log Containers of ↓BUF. To avoid the presence of dirty data in Log Containers, only committed elements of ↓BUF are flushed in their target structure as soon as a full Flash page can be built. So, transaction atomicity impacts only the ↓BUF management. In addition to the *Start* and *End* markers of ↓BUF, a *Dirty* marker is needed to distinguish between committed and dirty pages. Rolling-back insertions leads (1) to copy after *End* the elements belonging to the window $[Start, Dirty]$ containing the committed but unflushed elements, and (2) to reset the markers ($Dirty = Dirty - Start + End$; $Start = End$; $End = Dirty$) thereby discarding dirty elements.

## 6.3 Cryptographic protections

The NAND Flash being not protected by the tamper-resistance of the secure MCU, cryptographic techniques are required to protect the database footprint against confidentiality and integrity attacks. Indeed, attacks can be conducted by a pirate (if the ST is stolen) or by the ST holder herself (to modify administrative forms, to forge medical prescriptions, etc.). The database must thus be encrypted and protected against four forms of tampering: *modification*, *substitution* (replace valid data by other valid data), *deletion* and *replay* of data items (i.e., replacing a data item by an old valid version).

*Confidentiality attacks*: All data items are encrypted separately using AES. To prevent statistical attacks on encrypted data, all instances of a same value are encrypted differently (see below).

[15] Actually, it is worth managing a very small buffer (e.g., 1 page) in RAM to buffer several insertions of the same transaction.

*Modification, substitution, deletion attacks*: Illicit *modifications* are traditionally detected by computing a Message Authentication Code (MAC: a keyed-cryptographic hash [24]) of each data item. The item address is incorporated in the MAC to prevent *substitution*. Finally, *deletions* are detected by checking the integrity of the container of a data item (e.g., the page containing it).

*Replay attacks*: Replay attacks are trickier to tackle. Traditionally, detecting *replay* attacks imposes including a version number in the MAC computation of each data item and storing the association between a data item and its version in the secure internal storage of the ST. If any data item may have any version number, maintaining these numbers requires either a very large secure memory, unavailable on STs, or maintaining a hierarchy of versions in NAND Flash, storing only the root in secure memory (thereby inducing huge update costs). The log-only database organization proposed in this paper greatly simplifies the problem. Obsolete LCs can only be reclaimed during database reorganization and reallocated to next database instances. Indeed when reorganization starts, all LCs of the current database are frozen. Hence, checking version can be done by storing only the current database instance number in the secure internal storage of the ST and by including this number in the MAC computation of each data item.

Thus, protecting the database is done using state of the art cryptographic techniques. At query execution, each accessed data item is decrypted and its integrity is checked. The integrity of the query result is guaranteed since the evaluation starts by metadata stored in the secure MCU (secure root). The challenge is however to minimize the cryptographic overhead by (1) adapting the granularity of encryption primitives to the granularity of the data items accessed and (2) proposing cryptographic conscious page organization.

*Granularity of encryption primitives*: LCs are written sequentially at flash page granularity, pushing for computing the MAC at that granularity to minimize the storage overhead. LCs are however mainly read randomly at a very small granularity, e.g., when following pointers in indexes or when accessing attribute values. In those cases, computing the MAC of a full page to check the integrity of a small chunk is clearly sub-optimal. For small granularity data, we use Block-Level Added Redundancy Explicit Authentication (AREA) [15], an authenticated encryption mode working as follows. A nonce is concatenated to small granularity plaintext data, and then potentially padded, to obtain an encryption block. After decryption, the nonce must match its initial value. Otherwise, at least one bit of the data or the nonce has been modified. The security of this construction is based on (1) the diffusion property of the block level encryption functions, (2) the uniqueness of the nonce, (3) the nonce size. Using 16 bytes block encryption functions such as AES [24], a 6 bytes nonce and 10 bytes of payload data, the chance of an attacker to correctly build a fake data is $1/2^{48}$. In our scheme, the nonce is composed of the data address (4 bytes) concatenated with the version information. Thus, the nonce is guaranteed to be unique in the whole database life. We use AREA for small granularity data, (e.g., pointers, small keys, small attributes) generally accessed randomly. For instance, PTR is encrypted using AREA, while the ordered list of tuple identifiers in *IND uses a classical MAC since the whole list is always accessed.

*Cryptographic conscious page organization*: Since Flash memory can only be read and written by page, the content of each page can be reorganized before being flushed in Flash to minimize cryptographic costs. This idea can be exploited in several structures. Typically, the structure KEY in ↓IND is always processed with the same access pattern. If a page of KEY is accessed, this means that a match exists in BF for the searched key $k_i$ and for this page. While searching $k_i$ within the page, to avoid decrypting the complete list of keys, we encrypt the searched key $k_i$ and look up its encrypted representation. However, identical keys must have different encrypted representations to prevent any statistical attack. To avoid duplicate values inside the same page we store only the last occurrence of each key in the page, the previous occurrences being obtained using PTR. To avoid duplicate values in different pages, we encrypt each key using an initialization vector (IV) [24] depending on the page and on the version. At lookup time, if $k_i$ is found in the targeted page, its integrity must be checked. In the rare cases where $k_i$ is not found (BF false positive), the integrity of the complete list of keys must be checked. To optimize both cases, we store in the page the MAC of each key in addition to the MAC of the whole set of keys.

## 7 Performance evaluation

Developing embedded software for SMCU is a complex process, done in two phases: (1) development and validation on simulators, (2) adaptation and flashing on SMCU. Our current prototype is in phase 1 and runs on a software simulator of the target hardware platform: a SMCU equipped with a 50 MHz CPU, 64 KB of RAM and 1 MB of NOR Flash, connected to a 4 GB external NAND Flash. This simulator is IO and crypto-accurate, i.e., it computes exactly the number of page read/write, block erase operations and cryptographic operations (by type), by far the most costly operations. In order to provide performance results in seconds, we calibrate[16] the output of this simulation using a set of micro-benchmarks and with performance measurements done on the ST, using a previous prototype named PlugDB. PlugDB has already reached phase 2 (i.e., runs on a real hardware platform), has been demonstrated [6] and is being experimented in the field in an experimental project of secure and portable medical-social folder [4]. While PlugDB is simpler than MILo-DB, it shares enough commonalities with MILo-DB design to allow this calibration.

Let us note that we cannot run existing lite or embedded existing DBMS systems in a SMCU because they cannot adapt to its tiny RAM (most products could even not execute within a SMCU). DBMS systems designed for SMCUs would match the RAM constraint but they all consider eXecute In Place (XIP) memories (e.g., EEPROM in [28], NOR Flash in [11]) enabling bit/word access granularity and updates in place. This is incompatible with NAND Flash which exhibits block/page access granularity and forbids updates in place.

### 7.1 Insertion cost and tuple lifecycle

This section assesses the benefit of the log-only strategy in terms of write I/Os to the NAND Flash.

[16]This calibration is important to take into account aspects that cannot be captured by the simulator (e.g., synchronizations problems when accessing the Flash memory). It impacts negatively the performance shown here roughly by a factor of 1.4.

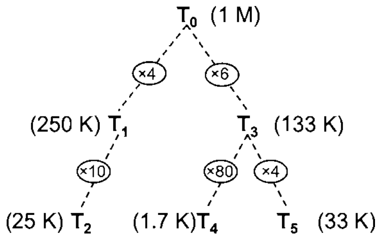


**Fig. 7** Synthetic database schema and tables cardinalities

We consider a synthetic database with 6 tables, $T_0$ to $T_5$ (see Fig. 7). Each of the 6 tables has 5 indexed attributes ID, Dup10, Dup100, MS1, MS10. ID is the tuple identifier, Dup10, DUP100, MS1 and MS10 are all CHAR(10), populated such that exact match selection retrieves respectively 10 tuples, 100 tuples, 1 % and 10 % of the table. Including the required foreign keys and other non-indexed attributes, the tuple size reaches 160 bytes. The tables are populated uniformly. We build a massively indexed schema holding 64 TSelect indexes, from which 29 for $T_0$ (5 from each sub-table + 4 defined at $T_0$ level) and 8 TJoin indexes (stored in 3 dedicated tables associated to $T_0$, $T_1$ and $T_3$).

Let us first study the insertion cost of a single tuple. The number of impacted indexes depends on the table, and so is the insertion cost. Insertion cost varies between 1.5 ms for $T_2$ to about 8 ms for $T_0$, and therefore does not constitute a bottleneck even for massively indexed tables. To illustrate the benefit with respect to state of the art techniques, we used the performance numbers of the 20 SD cards tested in [30] and evaluated the insertion cost of one tuple with indexes built as classical $B^+$-Trees on top of a FTL. Averaging the SD cards performance, this cost varies between 0.8 s ($T_2$) to 4.2 s ($T_0$). The minimal insertion cost (best SD card, table $T_2$) is 42 ms while the maximal cost (worst SD card, table $T_0$) is 15 s. Moreover, these numbers obviously do not include any technique for managing versions (to avoid replay attacks).

## 7.2 Tuple lifecycle and scalability limit

The numbers presented above must be put in perspective with the cost incurred by future reorganizations. Thus, we compare the write IOs induced by the insertion of a single tuple during the complete lifecycle of the database (i.e., through all reorganizations) with the same operation over a FTL without reorganization.

Considering the final size of the database (1 M tuples in $T_0$) and a scalability limit such that $\downarrow$DB = 100 K (i.e., 100 K tuples in $T_0$), the total number of IOs induced by a single tuple is between 4 and 5, depending on the table. This number rises between 12 and 17 with $\downarrow$DB = 10 K (the smaller the scalability limit, the higher the number of reorganizations). The reason for these surprisingly small numbers is twofold: (1) during the initial insertion in $\downarrow$DB, buffers factorize the write cost of all elements inserted synchronously (e.g., attributes and entries of indexes of the same table, see Sect. 6.1); (2) the reorganization incurs the rewriting of the complete database once, the part of this cost attributed to each tuple being proportional to its size (including indexes). In our evaluation, this size is at maximum 300 bytes, leading to about 1/7 IO per tuple for each reorganization. In contrast, an insertion through a FTL would produce between 12 ($T_2$) and 31 ($T_0$) random IOs, under the favorable hypothesis that

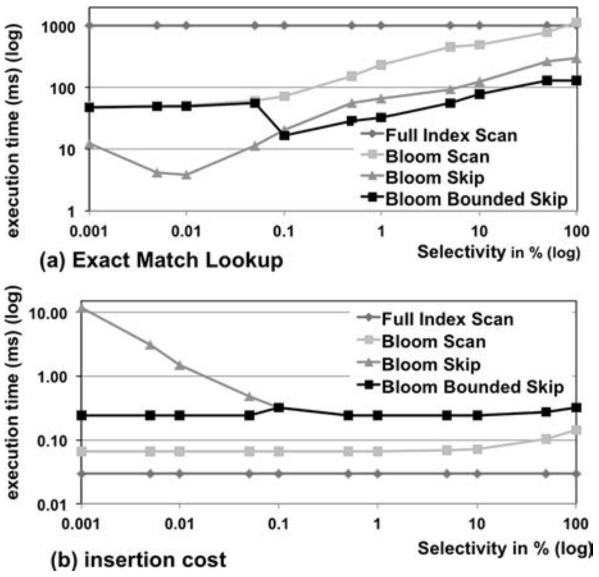


**Fig. 8** Performance of (unreorganized) log-only indexes

each insertion in a $B^+$-Tree index generates a single IO. Each of these IO in turn generates $p$ physical writes, where $p$ is the write amplification factor of the considered FTL.[17] Thus, MILo-DB not only speeds-up the insertion cost at insertion time but also reduces the *total write cost*, thus reducing energy consumption and maximizing the NAND flash lifetime.

## 7.3 Performance of log-only indexes

We now focus on the cost of exact match lookups and insertions for the log-only indexes, before any reorganization, detailed in Sect. 4. Contrary to conventional databases, indexes can be beneficial even with very low selectivity; indeed, random or sequential reads on NAND Flash (with no FTL) have the same performance. Therefore, it makes sense to vary selectivity up to 100 %. To this end, we built a single table of 300 K records, stored in ↓DB, with 11 attributes, populated with a varying number of distinct values (3, 15, 30, . . . , up to 300 K), uniformly distributed.

Figure 8a reports the results for exact match index lookups and Fig. 8b for index insertions (in one single index). For insertions, we considered the worst case (i.e., inserting a single tuple, then committing, thereby limiting the benefit of buffering).

We compare the index proposed in Sect. 4 with 3 other log-only indexes: *Full Index Scan* has neither PTR nor BF and is thus limited to a full scan of the KEY LC. It is used as a baseline. *Bloom Scan* has no PTR but can use the BF to avoid the full scan of KEY LC. Bloom Scan can be seen as a trivial extension of PBFilter [36] for secondary indexes. *Bloom Skip* has PTR, BUF and KEY but does not bound

[17] Note that giving a value of $p$ for simple Flash devices like SD cards is difficult since FTL code is proprietary. It is however necessarily rather large because of their reduced cache capabilities. This is confirmed by the ratio between sequential and random writes (between 130 and 5350! [30])

the insertion cost (i.e., the pointer chain is never interrupted). Finally, we name our proposed index *Bloom Bounded Skip*.

As expected, *Full Index Scan* has a very good insertion cost and a very bad lookup performance (full scan whatever the selectivity). *Bloom Scan* reaches pretty good insertion costs but lookups do not scale well with low selectivity predicates (the whole BF and KEY need to be scanned). Conversely, *Bloom Skip* performs better in terms of lookup but induces very high chaining costs when the inserted value is infrequent (BF needs to be scanned to find the previous occurrence of the inserted value). *Bloom Bounded Skip* appears as the best compromise, with a bounded insertion cost (with selectivity higher than 0.1 %, the pointer chain is broken), and very good lookup costs. With low selectivity predicates, it even outperforms any other index, including *Bloom Skip*, because pointers accessed in PTR are smaller in size.

## 7.4 Performance of the overall system

This section analyzes the behavior of the whole system considering the query cost of several types of queries, the impact of updates and deletes and the reorganization cost.

We first focus on the query cost and run a set of 18 queries (see Table 1): 12 queries, termed $Mono_i$, involve an exact match selection predicate on a single table on attribute ID, DUP10 or DUP100, joins this table up to $T_0$ and projects one attribute per table. 3 queries, termed $Multi_i$, involve 2 or 3 exact match predicates on MS1 or MS10. Finally, 3 queries, termed $Range_i$, involve a range predicate on $T_5$.DUP100. This attribute is therefore indexed using an adequate bitmap encoding as proposed in [13] enabling range predicate evaluation using a Full Index Scan. We measured the query response time with 3 settings: (1) the database has just been reorganized and thus $\downarrow$DB $= \emptyset$; (2) $\downarrow$DB $=$ 10 K; (3) $\downarrow$DB $=$ 100 K. Figure 9 presents the measurements for the 18 queries, ordered by the number of resulting tuples ($X$ axis). We split the graph in two in order to have different scales for response time (0–400 ms, 0–10 s).

For selective queries (1–799 results), selection cost is relatively important with large $\downarrow$DB (100 K) while with $\downarrow$DB $=$ 10 K the response time is very near the reorganized one. Considering several predicates (Multi1) increases this cost, as expected. Finally, the cost of Mono1 is almost zero because it retrieves a $T_0$ having a specific ID, which is, in our setting, the tuple's physical address.

For less selective queries (1 K–60 K results), the cost is dominated by the projection. Consequently, the $\downarrow$DB size has little influence.

Regarding updates and deletes, measurements on $\downarrow$DB $=$ 10 K and $\downarrow$DB $=$ 100 K have been done after having randomly deleted 3000 tuples (with cascade delete option) and having updated 3000 tuples (uniformly distributed on DUP10, DUP100, MS1, MS10 and A1 attributes of each table). We observed little influence of updates and deletes on performance, because they are evenly distributed. Considering more focused updates on the queried attribute value would lead to larger degradations, which stay however rather limited thanks to the indexation of $\downarrow$UPD.

**Table 1** Algebraic expression of the 18 queries

| Name | NbRes | Algebraic expression |
|---|---|---|
| Mono1 | 1 | $\pi_{T_0.A_1}(\sigma_{ID=500000}(T_0))$ |
| Mono2 | 8 | $\pi_{T_0.A_1,T_3.A_1}(T_0 \bowtie \sigma_{ID=10000}(T_3))$ |
| Mono3 | 10 | $\pi_{T_0.A_1}(\sigma_{DUP_{10}=\text{'VAL\_50000'}}(T_0))$ |
| Mono4 | 30 | $\pi_{T_0.A_1,T_3.A_1,T_5.A_1}(T_0 \bowtie T_3 \bowtie \sigma_{ID=5000}(T_5))$ |
| Mono5 | 80 | $\pi_{T_0.A_1,T_3.A_1}(T_0 \bowtie \sigma_{DUP_{10}=\text{'VAL\_1000'}}(T_3))$ |
| Mono6 | 100 | $\pi_{T_0.A_1}(\sigma_{DUP_{100}=\text{'VAL\_5000'}}(T_0))$ |
| Multi1 | 100 | $\pi_{T_0.A_1,T_1.A_1,T_2.A_1,T_3.A_1,T_4.A_1,T_5.A_1}(T_0 \bowtie T_1 \bowtie \sigma_{MS_1=\text{'VAL\_50'}}(T_2) \bowtie T_3 \bowtie T_4 \bowtie \sigma_{MS_1=\text{'VAL\_50'}}(T_5))$ |
| Mono7 | 300 | $\pi_{T_0.A_1,T_3.A_1,T_5.A_1}(T_0 \bowtie T_3 \bowtie \sigma_{DUP_{10}=\text{'VAL\_500'}}(T_5))$ |
| Mono8 | 599 | $\pi_{T_0.A_1,T_3.A_1,T_4.A_1}(T_0 \bowtie T_3 \bowtie \sigma_{ID=1000}(T_4))$ |
| Mono9 | 799 | $\pi_{T_0.A_1,T_3.A_1}(T_0 \bowtie \sigma_{DUP_{100}=\text{'VAL\_1000'}}(T_3))$ |
| Multi2 | 998 | $\pi_{T_0.A_1,T_1.A_1,T_2.A_1,T_3.A_1,T_4.A_1,T_5.A_1}(T_0 \bowtie T_1 \bowtie \sigma_{MS_{10}=\text{'VAL\_5'}}(T_2) \bowtie T_3 \bowtie \sigma_{MS_{10}=\text{'VAL\_5'}}(T_4) \bowtie \sigma_{MS_{10}=\text{'VAL\_5'}}(T_5))$ |
| Mono10 | 2997 | $\pi_{T_0.A_1,T_3.A_1,T_5.A_1}(T_0 \bowtie T_3 \bowtie \sigma_{DUP_{100}=\text{'VAL\_50'}}(T_5))$ |
| Range1 | 2997 | $\pi_{T_0.A_1,T_3.A_1,T_5.A_1}(T_0 \bowtie T_3 \bowtie \sigma_{DUP_{100}>\text{'VAL\_99'}}(T_5))$ |
| Mono11 | 5995 | $\pi_{T_0.A_1,T_3.A_1,T_4.A_1}(T_0 \bowtie T_3 \bowtie \sigma_{DUP_{10}=\text{'VAL\_100'}}(T_4))$ |
| Multi3 | 9971 | $\pi_{T_0.A_1,T_1.A_1,T_2.A_1,T_3.A_1,T_4.A_1,T_5.A_1}(T_0 \bowtie T_1 \bowtie \sigma_{MS_1=\text{'VAL\_50'}}(T_2) \bowtie T_3 \bowtie T_4 \bowtie \sigma_{MS_1=\text{'VAL\_50'}}(T_5))$ |
| Range2 | 14955 | $\pi_{T_0.A_1,T_3.A_1,T_5.A_1}(T_0 \bowtie T_3 \bowtie \sigma_{DUP_{100}>\text{'VAL\_95'}}(T_5))$ |
| Range3 | 29912 | $\pi_{T_0.A_1,T_3.A_1,T_5.A_1}(T_0 \bowtie T_3 \bowtie \sigma_{DUP_{100}>\text{'VAL\_90'}}(T_5))$ |
| Mono12 | 59881 | $\pi_{T_0.A_1,T_3.A_1,T_4.A_1}(T_0 \bowtie T_3 \bowtie \sigma_{DUP_{100}=\text{'VAL\_10'}}(T_4))$ |

## 7.5 Reorganization cost and scalability limit

Let us now consider the reorganization cost. We consider a fixed size for *DB (900 K tuples in $T_0$) and vary the size of ↓DB (varying the $T_0$ table from 10 K to 100 K tuples, other tables growing accordingly). The reorganization cost varies linearly from 6.2 min (for 10 K) to 7.7 min (for 100 K) (see Fig. 10a). It is worth noting that reorganization does not block queries (it can be efficiently stopped and resumed). The reorganization cost results from (1) reorganizing ↓IND, ↓DEL and ↓UPD and (2) reading $*DB_i$ and rewriting $*DB_{i+1}$. The Flash "consumption", i.e., the quantity of Flash memory written during one reorganization, varies between 0.5 GB (for 10 K) and 0.9 GB (for 100 K) (see Fig. 10b). However, 100 reorganizations are required to reach 1 M tuples in $T_0$ with ↓DB of 10 K, while only 10 are necessary with ↓DB of 100 K. In any case, the Flash lifetime does not appear to be a problem.

As a final remark, note that the FTL approach, already disqualified due to its write behavior, is not a good option in terms of query performance either. Indeed, except for highly selective (then cheap) queries where the FTL approach can perform slightly better than the log-only approach, the FTL overhead incurred by traversing translation tables makes the performance of less selective (therefore costly) queries much worse than with MILo-DB. Indeed, the high number of IOs generated at projection time dominates the cost of these queries.

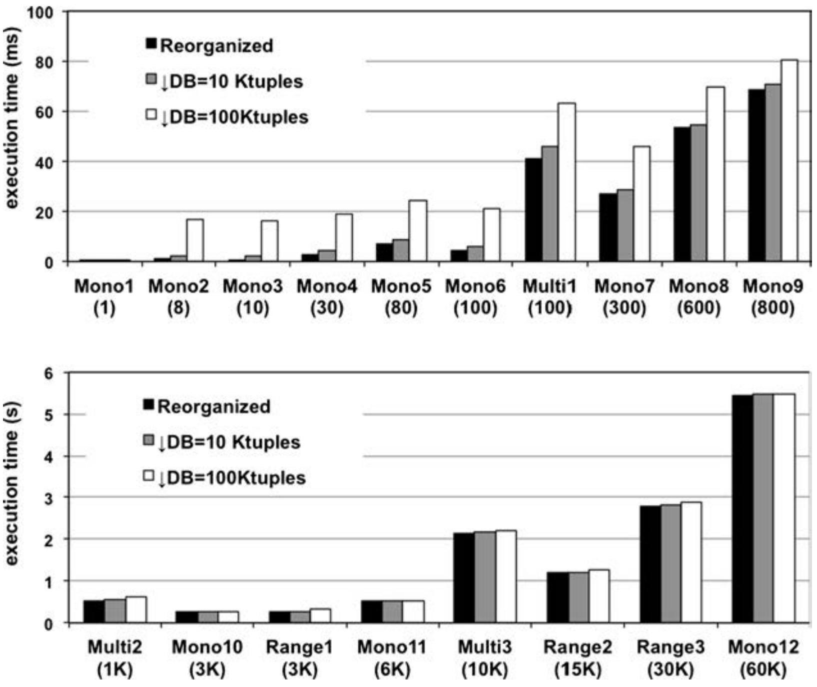


**Fig. 9** Performance of 18 queries with different settings

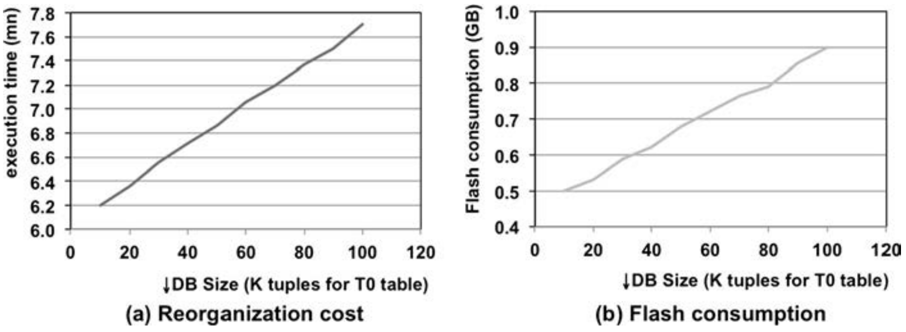


**Fig. 10** Reorganization cost and flash consumption

## 8 Conclusion

This paper proposed a new approach based on log-only data structures to tackle the conflicting NAND Flash/tiny RAM constraints inherent to Secure Tokens. It has shown the effectiveness of the approach to build a complete embedded DBMS engine called MILo-DB. MILo-DB is well adapted to data insertions, even at high rate, even with massive indexing, and supports gracefully a reasonable amount of updates and deletes. It offers an efficient evaluation of relational queries with a tiny RAM on gigabyte sized datasets and can protect these datasets against any form of confidentiality and integrity attacks. Hence, MILo-DB perfectly matches the requirements of a personal secure server where data (either stored locally or in the Cloud) needs to be managed under holder's control.

MILo-DB is however not adapted to all contexts. For instance, high updates/deletes rates would trigger too frequent reorganizations, long transactions and complex queries (e.g., involving ad-hoc joins or nested queries, etc.) are not yet supported. We currently investigate part of those issues but some of them are inherent to the approach. We are also adapting the design to cope with a very small number of Log Containers (typically less than 4) to efficiently address contexts where the FTL cannot be bypassed (e.g., SD cards).

This paper has shown that managing a crypto-protected gigabyte sized database within a secure token is not pure utopia. Beyond the interest for PIM applications, the major technical contribution is being able to manage a complete database without generating any random write. This result may have a wider applicability, in every context where random writes are detrimental in terms of I/O cost, energy consumption, space occupancy or memory lifetime.

**Acknowledgements** This work has been partially funded by the French ANR KISS project under grant No. ANR-11-INSE-0005. The authors also wish to thank Philippe Bonnet for his accurate comments on early versions of this paper.

## References

1. Agrawal, D., Abbadi, A.E., Wang, S.: Secure data management in the cloud. In: DNIS (2011)
2. Agrawal, D., Ganesan, D., Sitaraman, R., Diao, Y., Singh, S.: Lazy-adaptive tree: an optimized index structure for flash devices. In: PVLDB (2009)
3. Allard, T., Anciaux, N., Bouganim, L., Guo, Y., Le Folgoc, L., Nguyen, B., Pucheral, P., Ray, I., Ray, I., Yin, S.: Secure personal data servers: a vision paper. In: PVLDB (2010)
4. Allard, T., Anciaux, N., Bouganim, L., Pucheral, P., Thion, R.: Trustworthiness of pervasive healthcare folders. In: Pervasive and Smart Technologies for Healthcare, Information Science Reference (2009)
5. Anciaux, N., Benzine, M., Bouganim, L., Pucheral, P., Shasha, D.: Revelation on demand. In: DAPD (2009)
6. Anciaux, N., Bouganim, L., Guo, Y., Pucheral, P., Vandewalle, J.J., Yin, S.: Pluggable personal data servers. In: SIGMOD (2010)
7. Arge, L.: The buffer tree: a technique for designing batched external data structures. Algorithmica (2003)
8. Bernstein, P., Reid, C., Das, S.: Hyder—a transactional record manager for shared flash. In: CIDR (2011)
9. Bityutskiy, A.B.: JFFS3 design issues. Tech. report (2005)
10. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM (1970)
11. Bolchini, C., Salice, F., Schreiber, F., Tanca, L.: Logical and physical design issues for smart card databases. In: TOIS (2003)
12. Bursky, D.: Secure microcontrollers keep data safe. PRN engineering services (2012). http://tinyurl.com/secureMCU
13. Chan, C.Y., Ioannidis, Y.E.: An efficient bitmap encoding scheme for selection queries. In: SIGMOD (1999)
14. Debnath, B., Sengupta, S., Li, J.: SkimpyStash: RAM space skimpy key-value store on flash. In: SIGMOD (2011)
15. Elbaz, R., Champagne, D., Lee, R.B., Torres, L., Sassatelli, G., Guillemin, P.: TEC-tree: a low-cost, parallelizable tree for efficient defense against memory replay attacks. In: CHES (2007)
16. Eurosmart: Smart USB token. White paper (2008)

17. Gemmell, J., Bell, G., Lueder, R.: MyLifeBits: a personal database for everything. Commun. ACM **49**(1) (2006)
18. Giesecke devrient: portable security token. http://www.gd-sfs.com/portable-security-token
19. Haas, L.M., Carey, M.J., Livny, M., Shukla, A.: Seeking the truth about ad hoc join costs. VLDB J. (1997)
20. Bonnet, P., Bouganim, L., Koltsidas, I., Viglas, S.D.: System co-design and date management for flash devices. In: PVLDB (2011)
21. Li, Y., He, B., Yang, R.J., Luo, Q., Yi, K.: Tree indexing on solid state drives. In: PVLDB (2010)
22. Li, Z., Ross, K.A.: Fast joins using join indices. VLDB J. (1999)
23. Lim, H., Fan, B., Andersen, D., Kaminsky, M.: SILT: a memory-efficient, high-performance key-value store. In: SOSP (2011)
24. Menezes, A.J., Van Oorschot, P.C., Vanstone, S.A., Rivest, R.L.: Handbook of Applied Cryptography. CRC Press, Boca Raton (2001)
25. Moglen, E.: FreedomBox. http://freedomboxfoundation.org
26. Muth, P., O'Neil, P., Pick, A., Weikum, G.: The LHAM log-structured history data access method. VLDB J. (2000)
27. O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E.: The log-structured merge-tree (LSM-tree). Acta Inform. (1996)
28. Pucheral, P., Bouganim, L., Valduriez, P., Bobineau, C.: PicoDBMS: scaling down database techniques for the smart card. VLDB J. (2001)
29. Rosenblum, M., Ousterhout, J.: The design and implementation of a log-structured file system. ACM Trans. Comput. Sci. (1992)
30. Schmid, P., Roos, A.: SDXC/SDHC memory cards, rounded up and benchmarked. http://tinyurl.com/tom-sdxc
31. Severance, D., Lohman, G.: Differential files: their application to the maintenance of large databases. ACM Trans. Database Syst. (1976)
32. Sundaresan, P.: General key indexes. US Patent No. 5870747 (1999)
33. Vo, H.T., Wang, S., Agrawal, D., Chen, G., Ooi, B.C.: LogBase: scalable log-structured storage system for write-heavy environments. Technical report (2012)
34. Weininger, A.: Efficient execution of joins in a star schema. In: SIGMOD (2002)
35. Wu, C., Chang, L., Kuo, T.: An efficient b-tree layer for flash-memory storage systems. In: RTCSA (2003)
36. Yin, S., Pucheral, P., Meng, X.: A sequential indexing scheme for flash-based embedded systems. In: EDBT (2009)