Personal Data Server Engine Design and performance considerations

Lionel Le Folgoc

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

Département d'informatique
UFR de Sciences

École Doctorale "STV"

Personal Data Server Engine

Design and performance considerations

THÈSE

Présentée et soutenue publiquement le mercredi 12 décembre 2012

pour l'obtention du

Doctorat de l'université de Versailles Saint-Quentin
(spécialité informatique)

par

Lionel Le Folgoc

Composition du jury

Directeur :	Luc Bouganim	Directeur de recherche, INRIA Paris-Rocquencourt.
Rapporteurs :	Thierry Delot	Maître de conférences, Université de Valenciennes.
	Jean-Marc Petit	Professeur, INSA de Lyon.
Examinateurs :	Nicolas Anciaux	Chargé de recherche, INRIA Paris-Rocquencourt.
	Stéphane Gancarski	Maître de conférences, LIP6 – UPMC, Paris.
	Karine Zeitouni	Professeur, Université de Versailles Saint-Quentin.

Laboratoire PRISM

informatics mathematics Inria

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 the holder's control. As it develops and expands, 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 (for example the block-erase-before-page-rewrite constraint, or the limited number of erase cycles) and embedded system constraints (for example scarce RAM space available), disqualifying known state of the art solutions, as previous works overcome one constraint by relaxing the requirements of another (for instance, use a log in RAM to defer updates in NAND Flash).

In this thesis, with embedded constraints in mind, we propose an alternative database engine that relies on two key concepts to specifically address this challenge: serialization and stratification of the complete database. A database fully organized sequentially precludes random writes and their negative side effects on Flash write cost. Then, the global stratification allows us to solve the inherent scalability issue of a serialized design and to maintain acceptable performance when this limit is reached, without abandoning the benefits of serialization in terms of random writes. We show the effectiveness of this approach through a comprehensive performance study.

Résumé en français

L'émergence de dispositifs portables et sécurisés à large capacité de stockage promet une véritable avancée dans la gestion des données sensibles. Ces dispositifs peuvent héberger aussi bien des données personnelles que les métadonnées référençant des documents chiffrés stockés dans le nuage, et permettent d'en contrôler les droits d'accès. La mise à disposition de tels systèmes accroît le besoin en techniques efficaces de stockage et d'indexation pour les bases de données embarquées.

Ces techniques sont très difficiles à concevoir en raison des contraintes de la Flash NAND (par exemple, l'effacement d'un bloc avant réécriture et le nombre limité de cycles d'effacement), auxquelles s'ajoutent celles des systèmes embarqués (par exemple, une très faible quantité de RAM disponible). Cette combinaison de contraintes disqualifie les travaux antérieurs de l'état de l'art qui surmontent l'une des contraintes en abaissant les exigences d'une autre (tel que l'usage d'un tampon en RAM pour différer les écritures en NAND).

Dans cette thèse, nous proposons une nouvelle alternative spécifiquement conçue pour le monde embarqué et répondant aux contraintes correspondantes. Elle repose sur deux principes fondamentaux : la sérialisation et la stratification complète de la base de données. Une base de données entièrement séquentielle évite les écritures aléatoires et leurs effets déplorables sur les performances d'écriture de la Flash NAND. Ensuite, la stratification globale permet un passage à l'échelle en terme de performance des requêtes sans renoncer aux bénéfices de la sérialisation. Les résultats analytiques et expérimentaux montrent que cette nouvelle approche répond très bien aux exigences des systèmes embarqués.

Remerciements

Je tiens tout d'abord à exprimer ma profonde gratitude à Luc Bouganim, mon directeur de thèse, et Nicolas Anciaux, co-encadrant de cette thèse. Je les remercie pour l'aide et le soutien qu'ils m'ont apporté pendant ces trois années. Je leur suis très reconnaissant pour leurs conseils, leurs critiques, leurs qualités humaines et leurs encouragements qui ont contribué à l'aboutissement de cette thèse.

J'adresse mes plus vifs remerciements aux membres du jury qui ont bien voulu consacrer à ma thèse une partie de leur temps. Je cite en particulier Thierry Delot et Jean-Marc Petit qui m'ont fait l'honneur d'accepter d'être rapporteurs de ma thèse. Je remercie également Stéphane Gançarski et Karine Zeitouni pour avoir accepté de faire partie de mon jury de thèse.

Ma reconnaissance va aux membres de l'équipe SMIS, qui m'ont permis de réaliser cette thèse dans des conditions privilégiées, qui font de SMIS un environnement très agréable et motivant. Un grand merci à Alexei Troussov pour sa disponibilité et sa spontanéité à partager ses connaissances techniques lors de l'implémentation et des expérimentations.

Pour finir, je voudrais dédier cette thèse à mes parents pour leur soutien et encouragements, et à mon frère Loïc pour m'avoir donné des avis plein de bon sens.

Table of contents

List of figures

List of tables

Chapter 1
Introduction

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 the holder's control. As it develops and expands, 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 (for example the block-erase-before-page-rewrite constraint, or the limited number of erase cycles) and embedded system constraints (for example scarce RAM space available), disqualifying known state of the art solutions, as previous works overcome one constraint by relaxing the requirements of another (for instance, use a log in RAM to defer updates in NAND Flash). In this thesis, with embedded constraints in mind, we propose an alternative database engine that specifically addresses this challenge. In this chapter, we first position the relevant aspects of the Personal Data Server environment for our study; then we list the precise objectives of the thesis. Third, we present the main difficulties that must be faced in order to provide an efficient implementation of the Personal Data Server engine. Finally, we give the main contributions of this thesis and the outline of this manuscript.

1.1 Context

Today, many citizens receive salary forms, invoices, banking statements, etc., through the Internet. In their everyday life, they continuously interact with electronic devices (at home, at work, at hospital, while driving or shopping, taking photos, etc.) acquiring, producing and exchanging large volumes of personal data. Be it for backup purposes, for sharing them with friends, or to benefit from Personal Information Management applications (e.g., budget optimization, pay-per-use, health supervision, etc.), they would like to store them "somewhere". Storing this data in a well organized, structured and queryable *personal database* [34] 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, the French Digiposte, Project VRM[2]) are pursuing this same objective.

Technology effectively provides several solutions to attain this laudable objective. Needless to say, any solution where one's data is gathered on their 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 users are concerned that the price to pay is losing the control of their own data[3]. This uncertainty is encouraged by recurrent news on privacy violations resulting from

[1] Cf. `http://www.kuppingercole.com/` (retrieved on 2012-06-15).

[2] Cf. `http://cyber.law.harvard.edu/projectvrm/Main_Page` (retrieved on 2012-06-15).

[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*", cf. `http://news.cnet.com/8301-1009_3-10437844-83.html` (retrieved on 2012-06-15).

negligence, abusive use and internal or external attacks; for instance, according to DataLossDB.org, since 2009, more than 50 data breaches have been accounted for each month, in spite of industry self regulations and standards (e.g. Visa's Cardholder Information Security Program) and legislations such as Health Insurance Portability and Accountability Act (HIPAA).

On the other hand, the FreedomBox [58] initiative suggests the idea of a small and cheap data server that each individual can plug on her Internet gateway, so as to avoid any risk linked to the centralization of their data on remote servers. The Personal Data Server (PDS) vision [9] promotes an idea similar to FreedomBox, that is providing a fully decentralized infrastructure where personal data remains under the holder's control, but with stronger guarantees. It builds upon the emergence of new devices combining the tamper resistance of a secure microcontroller [22] 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. To go more in-depth, a secure token is organized around a slow-clocked processor, a tiny RAM space, a small internal stable storage, and connected via a bus to external NAND Flash chips. These devices have different form factors (e.g., SIM card, USB token, Secure MicroSD, etc.) and names (e.g., Personal Portable Security Device [41], Smart USB Token [32], Secure Portable Token, or SPT [9], etc.). Despite this diversity, they share similar characteristics (low-power, cheap, highly portable and highly secure).

This unprecedented conjunction of portability, secure processing and Gigabyte-sized storage holds the promise of a real breakthrough in the secure management of personal data. Moreover, a secure device capable of acquiring, storing, organizing, querying and sharing personal data under the holder's control would be a step forward in translating the personal database vision into reality.

1.2 Motivation

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 Secure Portable Token Flash memory) or remotely (e.g., in the Cloud). In the latter case, the secure tokens only manage 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 secure token holder's control, thereby providing ultimate security and privacy to cloud data [3]. Whatever the approach, the amount of data/metadata to be managed by the Secure Portable Tokens can be rather huge, embracing potentially the complete digital history of an individual.

Turning this vision into reality requires developing and embedding, in Secure Portable Tokens, software components capable of acquiring, storing and managing, securely, sensitive data, although the objective is beyond a simple secure data repository. The ambition, is, on the one hand, to allow the development of new, powerful, user-centric applications, while, on the other hand, to serve data requests in a "classical" mean from existing server-based applications. Around these modules, one must be able to control in a user-friendly way the sharing conditions concerning her data. These objectives, and the focus of this thesis, require solving a core technical challenge, that is designing a DBMS engine embedded in a Secure Portable Token providing at least basic storage, indexing and query facilities with acceptable performance. A more in-depth description of the entire Personal Data Server (PDS) vision will be given in Chapter 3.

1.3 Problem statement

Embedded devices in general are often subjected to several restrictions, be it on size, cost, energy consumption, shock resistance, etc. Indeed, Secure Tokens are no exceptions in this context, but are subject to stronger restrictions, due to their use of a secure microcontroller that must comply with security properties. These limitations force to adopt certain hardware technologies and designs, such as reducing the system capabilities with less RAM and processing power, or using storage systems like NAND Flash without any mechanical moving parts.

As such, the scarce RAM space on Secure Tokens presents a problem to handle join operations. "Last resort" join algorithms (block nested loop, sort-merge, Grace hash, hybrid hash) are known to rapidly decrease in performance when the smallest join argument exceeds the RAM size [37]. In addition, security incentives (only the secure micro controller is trusted) preclude swapping data to the terminal or the external storage, because the encryption cost would be prohibitive (considering the ratio between the RAM size and the size of the data to be joined, the amount of swapped data would be consequent). This implies that not only joins but all operators involved in the query must minimize their RAM consumption and intermediate results materialization. Since applications can be dynamically downloaded in the Secure Token at the user convenience, materialized views should not be considered, because we cannot assume all applications are known in advance. Therefore, massive indexing is usually required in the embedded context to tackle the RAM limit and compute database queries with acceptable performances.

Additionally, NAND Flash chips exhibit uncommon characteristics. First, reads and writes are done at the page granularity; writes, being more expensive than reads, must be performed sequentially within a block. Second, erase operations are done at the block granularity: a page previously written cannot be overwritten without erasing the

complete block containing it, which can be an order of magnitude slower than writes. On top of that may be added lifetime concerns, because a block wears out after about ten thousand write/erase cycles; beyond this threshold it is considered unable to hold data. Because a massively indexed database may generate numerous random writes on index updates, combining these constraints with the RAM shortage makes the design of a fully-fledged database engine very challenging, especially for storage and indexing aspects.

State of the art indexing methods that target Flash memory (see Section 2.3) were not designed with both NAND Flash characteristics and embedded context in mind and poorly adapt to these constraints. Although many existing works have been designed to support NAND Flash storage models, they do not address concerns with frequently updated data such as indexes. Other approaches suggested adapting traditional structures such as B^+-Trees and other tree-like structures to NAND Flash. While implementation decisions vary, these methods commonly rely on a *batch approach*: index updates are delayed using a log and grouped by frequency or locality, for example to write only once updates to the same node. This allows to mitigate one of the main concerns with using Flash on database, by decreasing the global write cost. Nevertheless, such policies require maintaining in-RAM data structures to provide acceptable lookup performance until the next delayed batch is written. Additionally, they are subject to unpredictable and low response times because they perform out-of-place updates, leading to sub-optimal Flash usage and overheads from address translation and garbage collection (see Section 2.4).

Since neither these designs nor policies are suitable for a RAM-constrained, Flash-based storage embedded context, the objective is to abandon the traditional structures and open a new direction of thinking, so as to try to design new data structures that adapt natively to Flash memory and embedded constraints at the same time.

1.4 Contributions

The objective of this thesis is to break the implication between massive indexing and fine-grained random writes and to design an efficient database engine under the embedded system constraints. The contributions of this thesis are summarized as follows:

1) We show that state of the art technologies cannot handle the combination of hardware constraints induced by a secure embedded system, such as a Secure Portable Token, and we isolate key points that will form necessary design principles.

2) We extend the concepts of *serialization* and *stratification* introduced for primary key indexes in [80] and apply them to the complete database (data, all indexes, logs and buffers). On a database fully organized sequentially, random writes and their negative side effects on Flash write cost are simply precluded. Then, the global stratification allows us to solve the inherent scalability issue of a serialized design and to maintain acceptable performance when this limit is reached. The objective is reorganizing the database without abandoning the benefits of serialization in terms of random writes.

3) We build a cost model simulator to evaluate the performance of the aforementioned solutions, and show that the proposed approaches are able to manage large amounts of data in spite of the strict hardware constraints of SPTs.

4) We implement a prototype of the Personal Data Server engine, experiment its behavior on several SPT's form factors, and illustrate the impact of different storage devices on the global performance of the system.

1.5 Outline

This thesis is composed of three main parts. First, in Chapter 2, we describe the common hardware characteristics of SPT-like devices, so as to summarize several constraints that are impacting a database system design in our embedded context. Then, we survey, on the one hand, "Commercial-Off-The-Shelf"-like embedded database engines, and state of the art storage and indexing techniques on the other hand, to evaluate their positioning in light of the constraints we extracted.

The second part contains Chapter 3 and introduces the Personal Data Server approach, whose objective is to manage (and regain control of) personal information in a secure, decentralized, privacy-preserved and user-controlled manner. We describe the global architecture of this concept, and define the features, capabilities and policies that must be proposed in order to provide a full-fledged personal data server engine in this context.

The third part is composed of Chapters 4 and 5. Chapter 4 concentrates on the design of an efficient personal data server engine that could be used as a building block of the PDS vision. We define the notion of Sequentially Written Structures (SWS), cornerstone of a serialized database storage model that we introduce as a new paradigm called database serialization, where all database structures are maintained in an append-only manner. However, the performance of this sequential layout being linearly linked with the database size, we introduce a second principle called database stratification, with the objective of further improving the performance without violating the SWS idea. Then, in Chapter 5, we build an analytical cost model to analyze the impact of serialization and stratification through the performance results obtained. We also further evaluate the validity of the personal data server engine by verifying experimentally the behavior of the serialized storage model on "state-of-the-art" real NAND Flash hardware.

Finally, Chapter 6 concludes and proposes some ideas for future work.

Chapter 2
State of the art

Thanks to its shock resistance, high density and read performance, NAND Flash is commonly used in embedded devices as the main data storage. However, combining Flash characteristics with resource constraints of the embedded world makes the design of storage and indexing models highly challenging. In this chapter, we will first evaluate and analyze thoroughly the hardware constraints of Secure Portable Tokens (SPTs) and by extension Flash-based embedded devices, and briefly review developments done in the embedded DBMS context. Then, we will concentrate on the literature related to the main dimensions of the problem addressed in our work, namely RAM conscious indexing schemes, Flash constraints management, Flash-based indexing schemes and log-structured indexing schemes. Finally, we will conclude this survey of the state of the art by outlining the contradictory objectives pursued by these different approaches and we will formulate the main challenges to be solved.

2.1 Hardware constraints

SPTs are emerging today with a wide variety of form factors, with smart cards and smart contactless passes being the most known to the public. They share many hardware commonalities with smaller devices such as embedded environmental and healthcare sensors, and handheld devices such as mobile phones or personal assistants (Figure 1). To a lesser extent, home routers, multimedia players and other set-top

boxes are based on the same hardware platform, although they lack the security of smart and SIM cards.

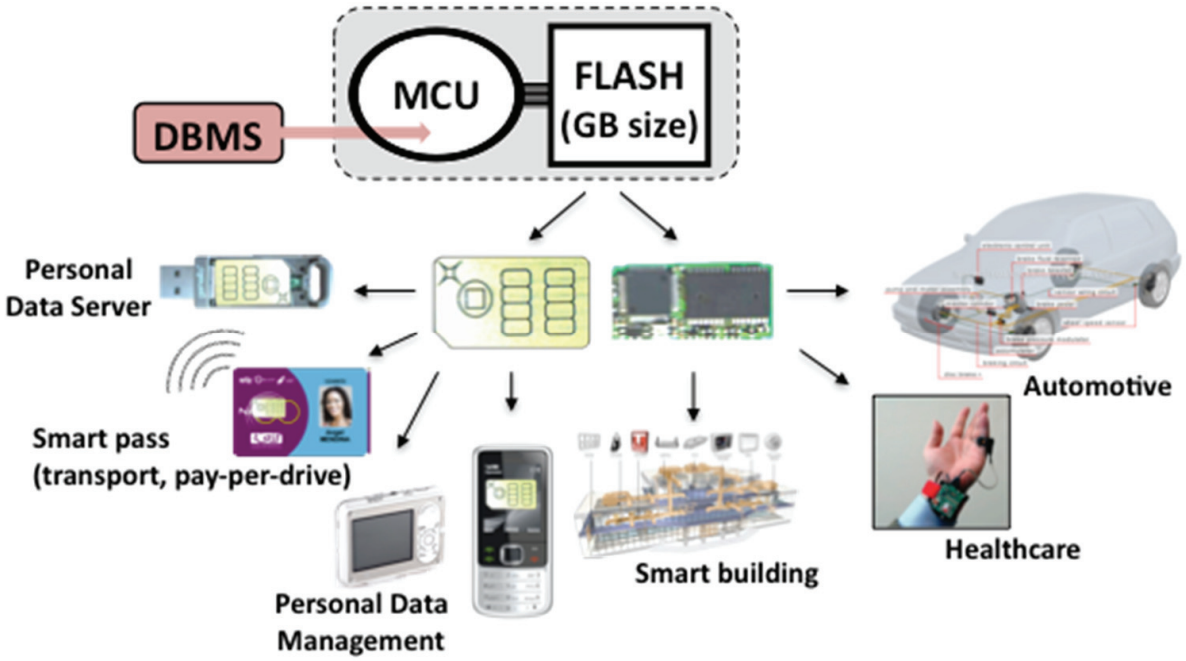

Figure 1: SPT form factors

SPT-like devices use a System on Chip (SoC) design, where an integrated circuit contains all components on a single chip. Their secure microcontroller unit (MCU) is typically equipped with a 32-bit RISC processor (clocked at about 150 MHz today[4]), ROM memory modules to host the operating system, static RAM chips (less than 128 KB [71]), internal stable storage (about 128 KB – 1 MB of NOR Flash), a cryptographic co-processor and security modules providing the tamper-resistance. Embedded code (in our context, the database engine and applications) is stored in the internal stable storage that features eXecute In Place (XIP) methods to reduce memory usage. Mass storage is provided by NAND Flash modules (several gigabytes of data), either through an external bus or an I/O port (e.g., SD cards). SPTs can include

[4] For instance, the frequency of ARM *SecurCore* products targeted at smart cards lies in the 50 – 275 MHz interval, cf. `http://www.arm.com/products/processors/securcore/index.php` (retrieved on 2012-06-15).

additional controllers to communicate with the outside world through various standards. For instance, sensors and smart passes use wireless communication (e.g., RFID, IrDA, 802.11, Bluetooth), devices embedded on cars use its Controller Area network (CAN), whereas bigger devices propose an extended connectivity through physical ports such as USB, Serial (RS232) and even Ethernet.

Hardware progress is fairly slow in the secure chip domain, because the size of the market (billions of units) and the requirements for high tamper-resistance lead to adopting cheap and proven technologies [31]. Nonetheless, microcontroller manufacturers forecast a regular increase of the CPU power, stable storage capacity and the support of high communication throughputs (up to 480 Mb/s, Hi Speed USB 2.0). The trend for external memory such as NAND Flash chips is to increase their density (TLC and MLC devices). However, due to its poor density, RAM will unfortunately remain a scarce resource in the foreseeable future. Indeed, the smaller the silicon die, the more difficult it is to snoop or tamper with its processing, and RAM competes for space with CPU, ROM, NOR and crypto-security modules on the same silicon die.

Therefore, hardware characteristics of SPTs can be translated into four main constraints, relevant to the design of an embedded database engine:

Constraint C1 – Small RAM/storage ratio

As mentioned above, MCUs have scarce RAM resources while the capacity of the external NAND Flash is only limited by its price. This leads to a ratio RAM/stable storage smaller than in traditional platforms running a DBMS by several orders of magnitude, and that is constantly decreasing.

Constraint C2 – Vulnerability of external storage

Due to its external connection (and removable nature in the case of SD cards), mass storage does not benefit from the MCU tamper-resistance. This implies that this memory must be protected against confidentiality and integrity attacks.

Constraint C3 – Scarce secure stable storage

On the other hand, internal XIP memory is protected by the MCU and can be used to store database metadata for confidentiality and integrity protection, as well as secret keys. Although its capacity regularly grows to match the needs of new embedded applications, it will remain limited in size for security purposes and thus a scarce resource in comparison with the volume of sensitive data stored in the database.

Constraint C4 – NAND Flash behavior

NAND Flash memory is badly adapted to fine-grain data (re)writes. The memory is divided in blocks, each block containing (e.g., 64) pages themselves divided in (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. As mentioned above, the technology trend is to increase the density of Flash, thereby ever worsening the constraints. To tackle these 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;
3. a *Wear Leveling mechanism* (WL) is required to guarantee that blocks are erased evenly.

A large body of work [48] 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]. In addition, FTL are black boxes firmware with behaviors difficult to predict and optimize, with the consequence that random writes can be orders of magnitude slower than sequential writes.

Constraints C2 and C3 concern the security aspect of the database engine, which is outside the scope of our work. Therefore, the following discussion will focus on constraints C1 and C4.

2.2 Embedded database systems

Popular and full-fledged DBMSs usually propose "lightweight" versions for embedding purposes. For example, Microsoft SQL Server features a Compact edition (5 MB RAM footprint) for use on Windows phones. Similarly, IBM and Oracle provide embedded variants of their database engines with such requirements, respectively called DB2 Everyplace [40] and Oracle Database Mobile Server [63]. On the other hand several existing database systems have been explicitly designed for embedded systems among other applications. For instance, SQLite [42] is a serverless transactional (ACID-compliant) database engine that supports a reduced subset of the SQL language; its typical size is about 300 KB of RAM. It is widely used in embedded operating systems for smartphones, such as Apple iOS, Google Android and RIM Blackberry. In contrast to SQLite, BerkeleyDB [64] is not a relational database engine, but proposes a key/value store; however, its minimal RAM footprint is about 1 MB.

[5] This is not the case in high-end or recent SSDs which can use relatively large onboard RAM to handle those constraints. For example, the Petrol SSD series from OCZ ships with a 512 MB DRAM cache, cf. `http://www.ocztechnology.com/res/manuals/OCZ_Petrol_Product_sheet.pdf` (retrieved on 2012-06-15).

Off-the-shelf software is targeted towards small - but comparatively - powerful devices (e.g., MP3 players, PDA, smartphones, set top boxes), whose storage and computation capabilities[6] are far beyond what constrained hardware such as smartcards allows. Typically, they address neither the limitations of secure MCUs (Constraint C1) nor the specificities of NAND Flash (Constraint C4).

Constraint C1 was partially addressed in some early work on database for secure MCUs. Bolchini et al. [18] analyzed algorithms, physical and logical data structures to ground design rules on databases for smart cards. The PicoDBMS prototype [66] implements a small database that requires less than 4kB of RAM and that is stored in the smart card's internal storage. However, the stable storage considered at that time was Electrically-erasable programmable read-only memory (EEPROM), whose physical characteristics are quite different from NAND Flash. Like RAM and NOR memory, EEPROM is directly addressable at the byte granularity, thus relies on main-memory data management techniques that cannot fulfill constraint C4. On the other hand GnatDB [74], whose footprint is only 11 KB, offers a storage model ensuring secrecy and tamper-detection for secure MCUs connected to external stable storage (e.g., Flash memory). Unfortunately it does not support query processing.

2.3 Massive indexing schemes

In our context, database queries must be executed on Gigabytes of data with only a few Kilobytes of RAM; therefore, the most RAM demanding operations, i.e. Joins and Aggregates, are the most problematic to handle. Traditional embedded low RAM devices (e.g., sensors) do not support joins, yet they are an essential feature for SPTs. Several studies about the impact of available RAM quantity on the behavior of join

[6] For example, the first iPhone release in 2007 featured a 600 MHz CPU and 128 MB of RAM, cf. `http://en.wikipedia.org/wiki/IPhone#Model_comparison` (retrieved on 2012-06-15).

operators have been conducted; they show that performance declines sharply when the ratio between the RAM size and the size of the data to be processed falls below a certain threshold. For instance, the performance of "Last resort" join algorithms (e.g., block nested loop, sort-merge, Grace hash, hybrid hash) quickly deteriorates when the smallest join argument exceeds the RAM size [37]. In contrast to the former, Jive join and Slam join use join indices [55] but both require that the RAM size be of the order of the square root of the size of the smaller argument. Moreover swapping data in the terminal or in the external NAND Flash is precluded due (1) to the dramatic amount of swapping required considering the ratio between the RAM size and the size of the data to be joined and (2) to the cost of encryption (only the microcontroller is trusted).

With the constraints of our embedded environment, the ratio between RAM and tables' size is so small that the only efficient solution is to resort to a highly indexed model where all (key) joins are precomputed. Similar approaches have been devised for the Data Warehouse context [72]. To deal with Star queries involving very large Fact tables (hundreds of GB), these 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 [76].

In a relational context, Anciaux et al. [8] proposed a multiway join index called Subtree Key Table (SKT) and a Climbing Index (CI) allowing to speed up selections at the leaves of a join tree. Combined together, these indexes allow selecting tuples in any table, reaching any other table in the join path in a single step. Queries can then be executed in a pure pipeline fashion without consuming RAM or producing intermediate results (see Section 4.3.2). This work may be considered as a first step towards the definition of indexing models and query execution techniques dedicated to systems matching RAM Constraint C1.

However, a massive indexation of the database has the effect of generating a huge amount of fine-grain random writes at insertion time to update the indexes, which is problematic with respect to Flash Constraint C4. In [29], Do et al. evaluated traditional (i.e., designed for magnetic drives) Join algorithms on Flash-based Solid State Disks (SSDs). Their experiments showed that random writes result in varying I/O and unpredictable performance. They also emphasized that similarly to HDDs, the buffer allocation strategy has a critical impact on the performance of join algorithms for SSDs. This yet again outlines the importance of Constraint C1.

2.4 FTLs and NAND Flash behavior

Traditional storage devices such as tapes and hard disk drives, feature interfaces that allow to read and write data by fixed-size chunks (typically, a disk sector); traditional software was thus implemented based on a block-device interface. However, raw NAND Flash chips provide a different interface: data can still be accessed and written by sectors (i.e. Flash pages), but erase operations are required prior to updating a page. Moreover, only blocks (multiple contiguous pages) can be erased, and blocks have a limited lifetime (they wear out – are unable to retain data – after a given number of erasures). FTLs (Flash Translation Layers) were introduced to hide Flash characteristics under a block-device abstraction, in order to prevent the expensive write-erase-rewrite on page updates from being handled at a software level.

The first paper about FTL concepts was written by Kawaguchi et al. in 1995 [44]. They present an UNIX driver for Flash memory, which manages a logical-to-physical address translation table, meta data related to the state of the blocks (allocated, written, invalid...), and a garbage collector that finds and erases invalid blocks to reclaim free space. Although algorithms were mainly targeted for NOR memory, which differs from NAND in terms of addressing, principles of FTLs were also described in a

patent [49] in 1992 (only issued on 2001), and the vocabulary and the proposed structures have been later extended to NAND memory.

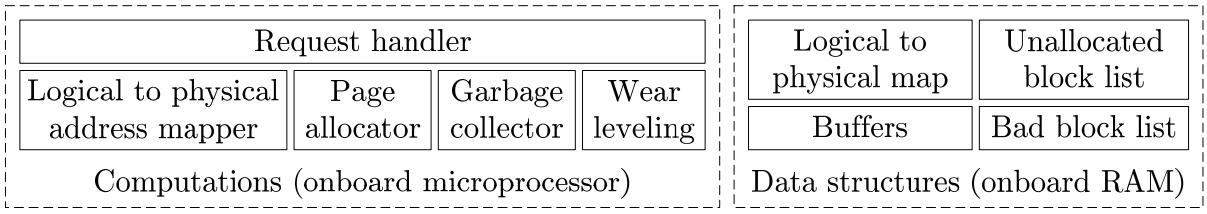

Figure 2: FTL features

One could see FTLs as a device firmware (Figure 2) that makes it possible to use Flash devices as drop-in replacements of traditional storage devices. Therefore an FTL must provide invariant logical addresses, hide the complexity of erase and update operations to high-level software and manage the lifetime of Flash blocks. Coherent logical addressing is achieved by maintaining a *logical-to-physical address translation table*. These mapping algorithms can be divided into three main categories (page-, block- and hybrid-mapping). They represent the FTL part that draws the most attention from researchers and are further detailed in the next paragraphs. To avoid the expensive in-place updates caused by page rewrites, the FTL performs *out-of-place updates* by allocating a new physical page, updating the mapping table to keep the logical address invariant, and finally marking the old physical page as invalid. Reclamation of invalid pages (i.e., management of erase operations) is handled by a *garbage collector*. This process should identify which pages are still valid in a candidate block, copy them to a new block, update the mapping table and erase the old block. Erase operations affect all pages of a block as a whole, which means that an inefficient reclaiming policy can lead to either a severe degradation of performance or, when some blocks are worn out more quickly than others, to a reduction of the drive reliability. This is the reason why FTLs include *wear-leveling mechanisms*, in order to distribute as much as possible block erases.

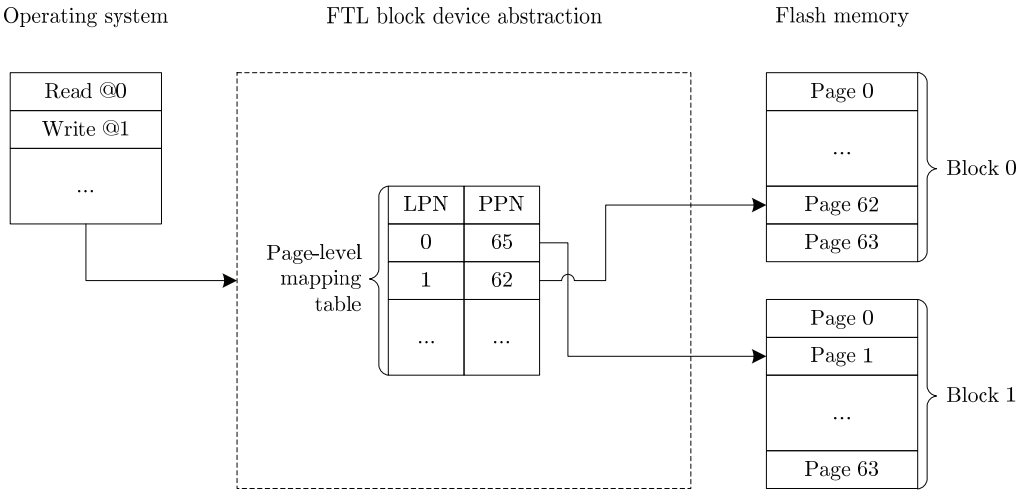

Figure 3: Page-level address mapping

Page-level mapping [12] (Figure 3) is the first algorithm designed and the most intuitive. The mapping table contains as many physical pages as logical ones, which means that if the Flash memory contains m physical pages, the size of the mapping table is also m. When the file system issues a request to write to a Logical Page Number (LPN), the translation into a Physical Page Number (PPN) is performed within the FTL by a lookup in the mapping table. If this page was previously written, the algorithm searches for a new free page, writes on it instead, and updates the mapping table to point to it. If no free page can be found, the garbage collector runs to reclaim free pages by erasing blocks that contain invalid pages.

The problem of this algorithm is the size of the mapping table that is directly proportional to the number of pages on the memory: for example, a 30 GB SSD contains about 60000000 pages, with 64 pages per block; 4 bytes are needed to represent all addressable pages, which means that the mapping table requires *4 bytes × 60000000 pages = 240 MB*! Therefore, this FTL algorithm was used in low capacity Flash memories only, and is not used for SSDs, as it would consume an

unreasonable amount of onboard RAM[7]. Besides, rebuilding such a table is very slow as it requires scanning all physical pages.

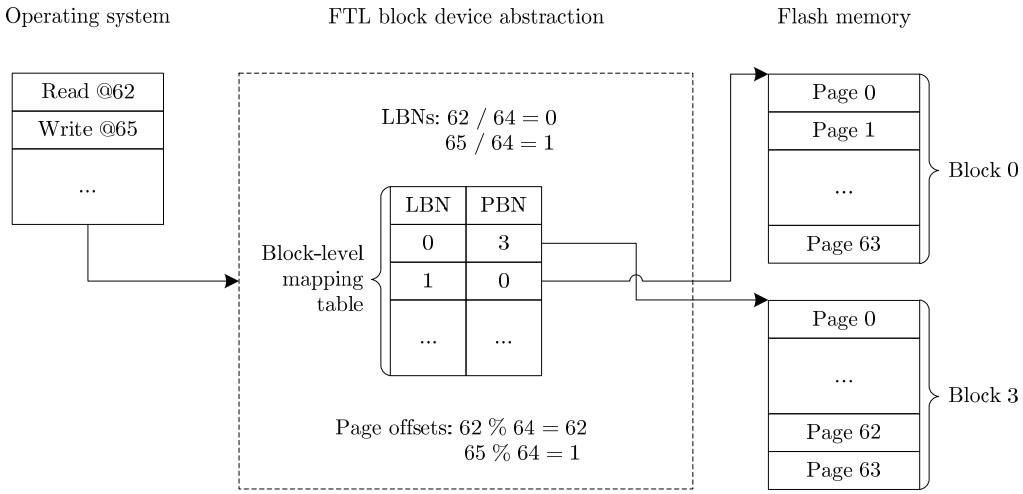

Figure 4: Block-level address mapping

Block mapping (Figure 4) is used to reduce RAM requirements to maintain the mapping table. Instead of referencing each logical page, the idea is to keep only Logical Block Number (LBN) references to Physical Block Numbers (PBNs); pages are accessed at a fixed offset, computed from the LPNs, for instance by a modulo on the pages per block count. The RAM consumption of this algorithm is significantly lower compared to page-level mapping: the 30 GB SSD illustrated above contains about 940000 blocks; 3 bytes are needed to represent whole blocks: the mapping table requires $3\ bytes \times 940000\ blocks = 4\ MB$. Although the required mapping information is very small, the block mapping algorithm does not offer good update performances. Indeed, because of the fixed page offset in a block, if the file system frequently issues a second write command to the same page, the FTL needs to switch to a free block and copy the valid pages from the old block prior to successfully completing the request. This behavior implies a very inefficient Flash page occupation,

[7] Cf. note 5 p. 15. For a typical 512 GB high-end SSD (including 512 MB of onboard RAM), its page-level mapping would require several Gigabytes of RAM space.

as many pages are still free when the switch occurs; the introduction of replacement blocks [13] was proposed to alleviate page copy costs, but these blocks suffer as well from low occupation ratios. Besides, current generation Flash chips must be programmed sequentially: inside a block, the second page can only be written after the first one, due to floating-gate interferences [46]. This renders the fixed page offset approach unusable on modern Flash devices.

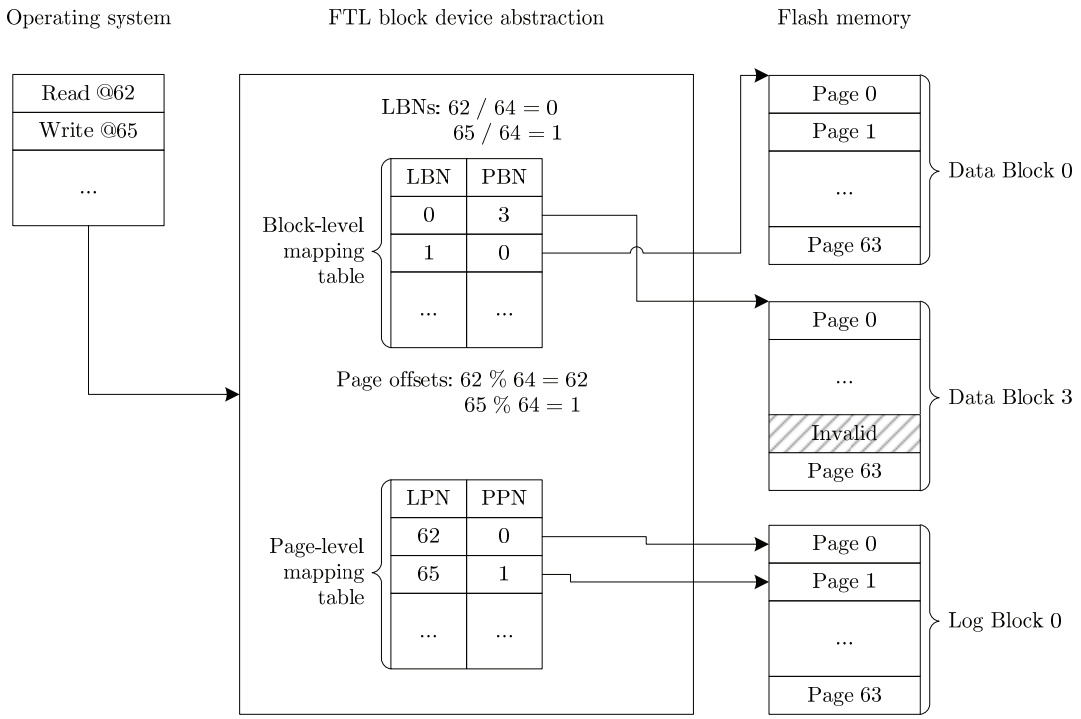

Figure 5: Hybrid address mapping

Hybrid mapping [21] (Figure 5) was proposed to overcome drawbacks of previous mapping techniques and deal with the trade-offs between RAM consumption, update performances and Flash usage. Its basic principle is to map m logical blocks to $m + n$ physical blocks, with m data blocks and n log blocks. Logical to physical mapping of data blocks is performed at a block level. However, updates are handled out-of-place and redirected into a free page of the log blocks, which must be mapped at the page level in order to track internally the latest version of repeatedly updated pages. In other words, log blocks act as a write cache that triggers the garbage

collector when consumed; the trade-off is that log blocks are subtracted from the actual capacity of the Flash device.

Several variants of this technique have been proposed, focusing on the associativity between data and log blocks. One of the first suggested hybrid techniques is called Block Associative Sector Translation (BAST) [45] and relies on *1:1* cardinality: each data block is associated to a single log block. Nonetheless, BAST performances drop dramatically with two special workloads. The first one is related to random writes: when all log blocks are allocated to data blocks, consequent random writes to new blocks trigger the garbage collector to reclaim a log block. The second one is inherent to the *1:1* policy: if the file system writes repeatedly to the same range of addresses (e.g., for a swap file), the corresponding data blocks will be repeatedly consumed and reclaimed as well, while other log blocks might be free to use.

Therefore multiple improvements to BAST have been proposed since 2007, which aim at reducing the amount of calls to the garbage collector. Lee et al. [52] designed a Fully Associative Sector Translation (FAST) that clusters all log blocks as a big buffer for all data blocks to compensate for low Flash usage. Park et al. [65] created a reconfigurable FTL from a *n:1* mapping in FAST to a $n{:}n + k$ one, to make it possible to associate more log blocks to a group of data blocks, in order to reduce the risk of a garbage collector run. Kang et al. [43] proposed a FTL algorithm that regroups several log and data blocks inside a structure called a superblock, so that hot and cold data could be dynamically separated. Later, Lee et al. [53] tried to further reduce garbage collection costs with Locality-Aware Sector Translation (LAST), by providing a *temporal locality* detection mechanism to separate more efficiently hot and cold data in log blocks.

Surprisingly, recent works such as DFTL [36] insist on using page-level mapping to avoid update slowdowns with hybrid mapping due to the limited number of log blocks. The key point is to maintain mapping tables in Flash, and selectively cache part of it

in RAM. On the one hand, it improves global performance thanks to the temporal locality in accesses that most enterprise-scale workloads exhibit; on the other hand, cache misses render the performance of a single operation less predictable. Another work [15] suggests that onboard RAM capacity should be increased to hold the complete page-level mapping table in RAM; although it may be suitable for high-end enterprise SSDs, it is not feasible in constrained embedded environments.

However, it is important to note that FTLs remain undocumented black boxes that are difficult to analyze due to their proprietary and patented nature; their exact behavior is constructor- and device-dependent[8]. Bonnet et al. proposed in [19] a bimodal FTL, with a first interface that allows unconstrained I/Os (like current FTLs) and a second for constrained I/Os. This provides near-optimal performances under the condition that the DBMS is aware of Flash limitations (for example, no update nor random write in this mode). This approach would help to reduce the cost of problematic access patterns on FTLs without jeopardizing DBMS design, but requires that both FTLs and database software be adapted (or even rewritten); thus it is premature to assume that upcoming devices will provide these features. Besides, it is unclear whether this approach would be efficient on high-end enterprise SSDs, where FTLs are highly optimized by manufacturers to take advantage of the physical layout of Flash chips in their drives (such as inter-chip parallelism by the request handler).

Accounting physical Flash characteristics without the block device abstraction layer, several approaches have been proposed to improve write performances on raw Flash. Lee et al. [51] designed In-Page-Logging (IPL), where update logs are co-located near original data pages (the last pages of full data blocks are reserved for the logs). This

[8] For instance, newer SSD products include FTLs featuring data compression and deduplication, which make performances dependent not solely on the workload access pattern, but also on the data content itself (i.e., existing benchmark suites writing constant data, e.g., zero-fill techniques, produce unrealistic optimal results).

method has the advantage that log pages can be found and merged back easily to recreate the logical data page, when the block is full and must be garbage collected. However, the frequency of log overflows with frequently updated pages containing "hot" data prevents IPL from being ideally-suited to storing indexes.

The Page-differential Logging approach (DPL) [47] suggests to only store in update logs the delta between the original physical page and its updated logical counterpart. This approach improves "merge" performances, because the size of the differential cannot exceed one page; therefore at most two page reads are needed to reconstruct the final page. Nonetheless, DPL requires maintaining a mapping table in RAM to track the physical page targeted by each differential, and where this differential is physically written in Flash; also, due to their small size, when several differential share the same log page, a more complex garbage collector is mandatory, so as to separate valid from invalid differentials.

All in all, although they do not rely on block device abstractions, the above techniques for raw Flash actually provide translation mappings, garbage collectors and similar FTL-like features. This makes them vulnerable to unpredictability among other issues such as a significant RAM consumption incompatible with secure MCU constraints.

2.5 Indexing techniques for NAND Flash

Conventional indexes like B$^+$-Trees [26] have been shown to perform poorly on top of FTLs [77], because each insertion triggers an out-of-place update of the Flash page that contains the leaf node of the tree in order to add the key. Many studies address the problem of storage and indexing in NAND Flash, mostly by adapting traditional structures to avoid such updates.

Journaling Flash File System version 3 (JFFS3) [16] keeps all file system objects (inodes, files, directory entries, extended attributes, etc.) in one large B$^+$-Tree, and logs

updates in a journal whose index is maintained in RAM. On read requests, the journal index is first looked up and determines whether the data to be read is in the "real" B$^+$-Tree or in the journal. Thanks to this journal, writes can be deferred at a more appropriate time, to merge node updates and reduce the number of NAND Flash writes. However, the journal index needs to be reconstructed at boot time, with a scan of all journal Flash pages. In other words, there is a trade-off between the performance improvements and the mount time of the file system, depending on the journal size.

Wu et al. [77] proposed an efficient B$^+$-Tree Layer for Flash (BFTL) that sits between the FTL and the database, and consists of a reservation buffer and a translation table. BFTL constructs an "index unit" with each inserted primary key, and manages index units as a log. A large buffer is allocated in RAM to hold the various insertions related to the same B$^+$-Tree node in the same log page. To maintain the logical view of the B$^+$-Tree, a node translation table built in RAM keeps, for each B$^+$-Tree node, the list of log pages that contain index units belonging to this node. To limit scanning costs due to an unexpected growth of the translation table, each list is compacted when a given threshold is reached (e.g. 3 log pages), and dirty records are flushed to update B$^+$-Tree Flash pages. Similarly to JFFS3, the efficiency of such a policy therefore depends on the allocated RAM size for tracking structures, on the commit frequency and on the size of the update batches. FlashDB [61], which is a framework based on B$^+$-Trees and BFTL, was proposed to obtain optimal performance by self-tuning these indexing parameters.

FD-Tree [54] is composed of a small B$^+$-Tree on the top and a few levels of sorted runs of increasing sizes below. As previous measurements showed [24], it is based on the fact that random writes limited to a small area (512 KB – 8 MB) perform similarly to sequential writes; thus updates are limited to the head tree. Later, they are merged into the lower-level sorted runs in batches, with the benefit that these mostly trigger sequential I/Os. However, the FD-Tree does not obey any RAM constraint and is

targeted to SSD products, which furthermore ship important quantities of onboard RAM to be used as buffer pool.

Agrawal et al. [4] suggested the Lazy-Adaptive Tree to minimize accesses to Flash memory. It uses a set of cascaded buffers: at flush time, elements are pushed to a lower-level buffer instead of being written directly to Flash. To overcome slowdowns in read operations due to buffer scanning, an online algorithm is proposed to resize optimally buffers depending on the workload. However, acceptable performance for buffer scans and memory reclamation is dependent on a low fragmentation of Flash buffers; therefore, LA-Trees also require more memory available in order to perform write coalescing techniques. The Buffer-Tree [11] also proposes a technique to design batched data structures for I/O efficiency and relies on a similar "lazy" approach, with buffers associated to internal nodes of the tree.

To summarize, recent B^+-Tree adaptations all rely 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, so as 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 mitigate 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. The RAM consumption and the random write cost are thus conflicting parameters. Under the RAM limitations of secure MCUs the commit frequency becomes de facto very high and the gain on random writes vanishes.

2.6 Log-Structured Indexes

Instead of adapting existing data structures to reduce Flash page updates, it has also been suggested in the literature to completely avoid them with sequential structures.

Many suggestions are based on principles from log-structured file systems [68] or older ones such as Differential File [70], which uses an overflow area to store new records and merges them later with the main data file, creating a transient log-organized structure. For instance, most existing Flash systems, such as JFFS/2 [16], YAFFS, and more recently UBIFS, rely on a log-structured approach, where the entire file system is organized as a log, with inodes written sequentially in segments and in-memory maps to maintain the current physical location of each inode.

B-FILE [60] is a structure designed to optimize specific workloads on NAND Flash. It is based on an analysis of a typical workload, which shows that writes are rarely completely random, and introduces the concept of "semi-random writes". It proposes to dispatch writes randomly across buckets (blocks), with each bucket filled sequentially. Therefore, the global write pattern becomes a semi-random pattern (which most of current FTLs support efficiently), and global write costs are further amortized as they are handled in bulks.

The log-structured history data access method (LHAM) [59] proposes to divide the time domain into intervals, and assign each interval a storage component. Each record is associated with a timestamp at transaction time and stored to a different component depending on this timestamp. New records are always inserted to main-memory components first; as time goes, "old" components are archived, i.e. flushed sequentially to the final storage. Following the idea of the Log-Structured Merge-Tree (LSM-Tree) [62], successive components can be merged together when archived in order to reduce lookup costs: a "rolling-merge" cursor walks through equal key values of the old and new components, emptying indexing data out from the old component to merge them with and fill the new component on disk. LogBase [75] also uses a log-only storage method ("log repository"). However, the log is abstracted as an infinite

sequential repository containing contiguous segments, with the underlying file system HDFS[9]. The log index – more traditional with only *<IdxKey, Ptr>* pairs – is entirely stored in RAM, although the authors state that LogBase can also employ a method similar to LSM-tree for merging out part of the in-memory indexes into the stable storage, so as to ensure persistence and avoids rescanning the log on recovery.

These proposals differ in the way indexes are built or maintained, but they always make use of relatively large buffers in RAM that are inherently incompatible with the constraints of our embedded environment. More recently, Bernstein et al. [14] proposed Hyder, a log-structured, multi-version key-value database stored in raw Flash memory and targeted at data centers (data shared over the network with multi-core nodes). 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, as binary trees are not adequate to index non unique keys.

To follow up in the key-value store context, SkimpyStash [27] aims at a very low memory footprint, by avoiding the overhead (4 B) of RAM-to-Flash key pointers. Key-value pairs are stored in an append-only log, alongside a pointer to the next record from the Hash Table (actually a previously written pair on Flash); the directory structure for these pairs is maintained in RAM in the form of a Hash Table, with each slot pointing to the head of a chain in Flash. Small Index Large Table (SILT) [56] organizes key-value pairs in three stores, either written sequentially or in a single batch, with data on Flash and indexes in RAM. Key-value pairs are first inserted in an append-only and write-friendly store called *LogStore*. It is then converted to a more lookup-friendly structure called *HashStore*, which is a hash table based on *Cuckoo*

[9] Hadoop Distributed File System, cf. `http://hadoop.apache.org/hdfs` (retrieved on 2012-06-15).

Hashing. Eventually, several hash tables are merged together with a sorted tree called *SortedStore*. In a nutshell, these proposals both rely on log structure to exploit sequential writes and maintain some form of in-memory (RAM) indexing, with a size proportional to the database size (between 0.5 and 1.5 B per record). These approaches unfortunately do not scale well in the secure MCU context.

Finally, PBFilter [81] 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 little RAM. However, PBFilter is designed for primary keys. With secondary keys the Bloom filter summary becomes non selective and hardly useful, as 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.

2.7 Conclusion

In this chapter, we first analyzed hardware characteristics of Flash-based embedded devices (using SPT as an example) and translated them into four main constraints: C1 "Small RAM/storage ratio", C2 "Vulnerability of external storage", C3 "Scarce secure stable storage" and C4 "NAND Flash behavior".

With these constraints in mind, we then surveyed the state of the art indexing and storage techniques, and found that none of them could meet all the requirements of our embedded environment. The challenge 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, that 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.

Chapter 3
The PDS approach

In this chapter, we illustrate the Personal Data Server (PDS) vision through different scenarios motivating our approach, and present the hypothesis related to the security of PDSs and of the infrastructure surrounding them. Next, we describe in detail the global PDS infrastructure and review all elements that are part of it, and conclude by showing how this vision can help enforcing user control rules.

3.1 Motivating examples

3.1.1 Healthcare scenario

Alice carries her electronic healthcare folder (along with other information) on a PDS. She has an account on *e-Store*, a Supporting Server provider. She downloaded in her PDS, from the Ministry of Health, a predefined healthcare database schema, an application to exploit it, and an access control policy defining the privileges attached to each role (physician, nurse, etc). Alice may manage the role assignment by herself or activate specific user policies predefined by e.g., a patient association. When she visits Bob, a new physician, she is free to provide her SPT or not, depending on her willingness to let Bob physically access it (this is a rough but effective way to control the sharing of her data, as with a paper-based folder). In the positive case, Bob plugs Alice's PDS on his terminal, authenticates to the PDS server with his physician

credentials, queries and updates Alice's folder through his local Web browser, according to the physician's privileges.

Bob prescribes a blood test to Alice. The test results is sent to Alice by the medical lab in an encrypted form, through e-Store acting here as a secure mailbox. The document is downloaded from e-Store and wrapped by Alice's PDS to feed the embedded database. If this document contains information Alice would like to keep secret, she simply masks this document so that it remains hidden from any user querying the database except her. The lab keeps track of this medical act for administrative purposes but does not need anymore to keep a copy of its medical content. If Alice loses her PDS, its tamper-resistance renders potential attacks harmless. She will then recover her folder from an encrypted archive stored by e-Store using, e.g., a pass-phrase.

Alice suffers from a long-term sickness and must receive care at home. Any practitioner can interact at home with Alice's PDS thanks to his netbook, tablet PC or PDA without need for an Internet connection. To improve care coordination, Bob convinces Alice to make part of her folder available 24/7, during a one month period, to him and to Mary, a specialist physician. A1ice uploads the requested part of her folder encrypted on e-Store. The secret key is exchanged with Bob's and Mary's PDSs in order for them to be able to download Alice's data on their own PDS and query it. While Alice's data is now replicated on Bob's and Mary's PDSs, Bob and Mary cannot perform actions on the replica exceeding their privileges and this replica will be destroyed after a one month period because their PDS will enforce these controls. Bob and Mary's actions are recorded by their own PDSs and sent back to Alice through e-Store for audit purpose. To make this sharing scenario possible, patients and practitioners are all assumed to be equipped with PDSs and these PDSs are assumed to share a compliant database schema.

Finally, if the Ministry of Health decides to compute statistics or to build an anonymized dataset from a cohort of patients, the targeted PDSs will perform the processing and deliver the final result while preventing any leakage of sensitive data or identifying information.

3.1.2 Vehicle tracking scenario

John, a traveling salesman, drives a car from his company during working hours and shares his personal car with Cathy, his daughter. Both have a PDS that they plug in the car to register all their personal trips. Several applications are interested in the registered GPS locations. John's insurance company adapts the insurance fee according to different criteria (e.g., the distance traveled, type of road used, and speed). Cathy will probably pay more than her father because she lacks enough driving experience. The Treasury is also interested by this information to compute John's carbon tax according to similar criteria, though the computation rules are different. Finally, John's company would also like to track John's moves to organize his rounds better. GPS raw data is obviously highly private. Fortunately, John's PDS externalizes only the relevant aggregated values to each application. In other words, each application is granted access to a particular view of the data registered in John's database.

3.1.3 BestLoan.com & BudgetOptim scenarios

Alice needs a loan to buy an apartment. She would like to find the best rates for her loan and, thus, relies on the service of *BestLoan.com*, a mortgage broker. To assess Alice's financial situation, BestLoan needs to get access to sensitive information from Alice's PDS such as salary, bank statements and tax information. Alice's data can be securely shared with Donald, a BestLoan employee, as follows:

1. Alice opts in for the BestLoan application and downloads the security policy associated to it in her PDS;

2. Donald authenticates to Alice's PDS with his credentials embedded in his own PDS and requests the required data;
3. Alice agrees to share this data with Donald for a specified duration (e.g., two weeks);
4. Finally Donald downloads the data in his PDS, all this by exchanging messages and data through the e-Store Supporting Servers.

Donald cannot perform actions on Alice's data exceeding their privileges or the retention period fixed by Alice because his PDS will preclude these actions. If Alice distrusts Donald, she can audit his activity and can at any moment opt out of the BestLoan application (with the effect of deleting Alice's data in Donald's PDS), all this again by exchanging messages through the e-Store.

Alice now wants to optimize her budget and thus opts in for the *BudgetOptim* application. BudgetOptim runs locally on Alice's PDS with a GUI running on the terminal. BudgetOptim accesses details of Alice's invoices, telecom bills, etc. in order to suggest more advantageous services according to her consuming profile. With BudgetOptim application, Alice does not share data with anybody. This last scenario is typical of many private applications that can process personal data (e.g., diet advices, tax minimization, pension simulation, vaccine reminders, etc.).

3.2 PDS Global architecture

PDS is not a simple secure repository of personal documents but rather provides a well organized, structured, consistent and queryable representation of these documents for serving applications requests. The difficulty to achieve this objective comes notably from the variety of data sources and applications targeted by PDS. This section presents an initial design of the PDS architecture.

3.2.1 Problem statement

As described in Section 2.1, a Secure Portable Token (SPT) can be seen as a low power but very cheap (a few dollars), highly portable, highly secure computer with reasonable storage capacity for personal use, and several form factors, ranging from SIM cards to various forms of pluggable secure tokens (Figure 1).

The level of trust which can be put in the PDS comes from the following factors:

1. The PDS software inherits the tamper resistance of the SPT making hardware and side-channel attacks highly difficult.
2. The basic software (operating system, database engine and PDS generic tools), called hereafter *PDS core*, can be certified according to the Common Criteria, making software attacks also highly difficult.
3. The PDS core can be made auto-administered thanks to its simplicity, in contrast to its traditional multi-user server counterpart. Hence, insider attacks (e.g., from the DBA in a traditional database server) are also precluded.
4. Compared to a traditional server, the ratio Cost/Benefit of an attack is increased by observations 1 and 2 and by the fact that a successful attack compromises only the data of a single individual.
5. Even the PDS holder cannot directly access the data stored locally. After authentication (e.g., by a pin code), she only gets the data according to her privileges.

Unfortunately, a PDS cannot provide all the required database functionalities (e.g., durability, if the PDS is lost or destroyed, availability when the PDS is disconnected, global queries involving data from several PDSs) without resorting to external servers, called hereafter *Supporting Servers*.

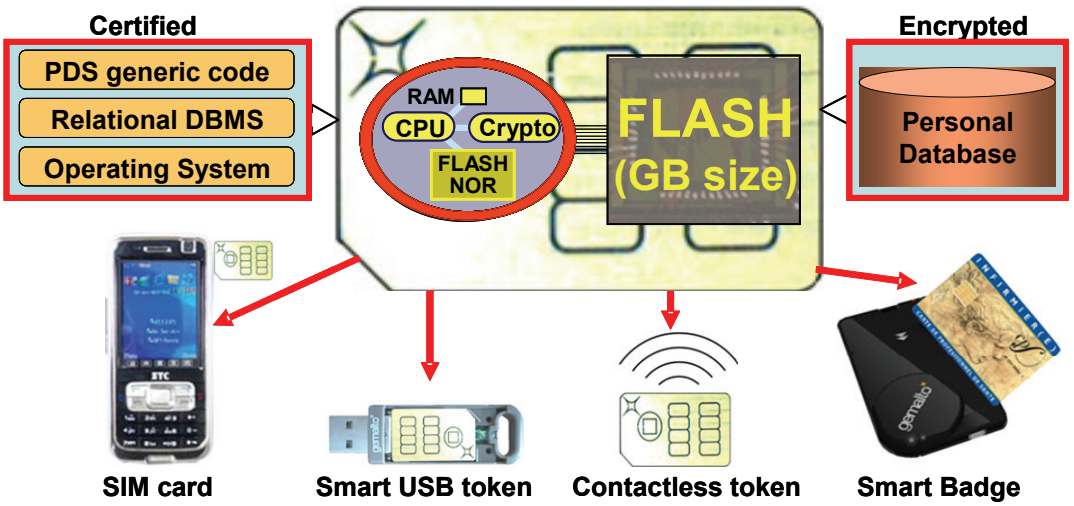

Figure 6: Possible SPTs in the embedded PDS context

We assume that Supporting Servers are *Honest* but *Curious*, a common security assumption regarding Storage Service Providers. This means that they correctly provide the services that are expected from them (typically serve store, retrieve, and delete data requests) but they may try to breach confidentiality of any data that is stored locally.

Therefore, implementing the PDS approach requires solving the problem stated below:

- To revisit the main database techniques to make the PDS core compliant with the SPT hardware constraints.
- To reestablish the traditional functions of a central server (durability, availability, global queries) in a secure way using Honest but Curious Supporting Servers.
- To provide the user with intuitive tools and underlying mechanisms helping her to control how her personal data is shared.

3.2.2 Positioning

Compared to an approach where all personal data is gathered on traditional servers, the benefit provided by PDS is fourfold. First, the PDS holder is his own Database Service Provider. Hence, abusive uses by the Database Service Provider are precluded.

Second, the PDS provides the holder with tangible elements of trust which cannot be provided by any traditional server (see factors 1 to 4 in Section 3.2.1). Third, privacy principles (e.g., limited retention, audit) can be enforced for the data externalized by the holder provided the recipient of this data is another PDS. Fourth, the holder's data remains available in disconnected mode.

However, alternatives to the traditional server exist. The Hippocratic database approach [5] has been precisely designed to protect personal data thanks to principles such as purpose specification, consent, limited collection, limited retention, audit, safety, etc. Part of PDS architectural ideas has been inspired by this work. But the Hippocratic database approach provides tangible guarantees only if the server can be fully trusted. In this respect, the PDS approach can be seen as a fully distributed implementation of a Hippocratic database where the founding Hippocratic principles can be definitely enforced.

The Database as a Service approach (DAS) [38] is another option. Here data is stored encrypted on the server and is decrypted at the client side, making server attacks harmless. This time, the DAS approach makes sense only if all clients can be trusted; the PDS provides a way to make the clients trusted.

Statistical databases [2] and data anonymization [33] are both motivated by the desire to compute statistics or to mine data without compromising sensitive information about individuals. Both require trusting the server, either to perform query restriction or data perturbation in the former case, or to produce the anonymized data set in the latter case. Though orthogonal to the PDS approach, these concerns still exist in the PDS context and must be addressed adequately.

3.2.3 Personal database

The personal database is assumed to be composed of a small set of database schemas, typically one per application domain. We make no assumption on the granularity of application domains but e-health and e-administration are illustrative examples of domains. Database schemas are defined by *DB Schema Providers*. Depending on the domain, a DB Schema Provider can be a government agency (e.g., Ministry of Health) or a private consortium (e.g., a group of bank and insurance companies).

Content Providers are external information systems that deliver personal data (e.g., blood test, salary form), encoded in XML. We make the simplifying assumption that each XML document conforms to one XML schema defined by a standardization organization (e.g., HL7) or by a DB Schema Provider (e.g., the Ministry of Health). To allow building a consistent and structured view of a set of related documents, an XML document (e.g., a prescription) is enriched with all referential data required to fill the embedded database accurately (e.g., detailed data related to the doctor who wrote the prescription and to the drug prescribed). Hence, the data contained in different documents related to a given doctor or drug can be easily queried and cross document processing becomes possible (e.g., get the list of current medical treatments or compute average blood pressure during the last month). Then the enriched document is pushed in an encrypted form to the recipient PDS through Supporting Servers (see Section 3.2.6 for a description of Supporting Servers). The recipient PDS downloads the XML document and wraps it into a set of records thanks to *mapping rules* provided by DB Schema Providers[10]. Mapping rules are declarative and interpreted by a generic wrapper, a certified component of the PDS core (see Section 3.2.7 for a deeper discussion on certification). The benefit of declarative mapping rules is not only

[10] The mapping rules are related to the transcription of XML documents into a structured database and are required even with an XML database.

that it simplifies the work of the DB Schema Provider but primarily that the safety of these rules can be controlled.

Figure 7 illustrates the wrapping of a prescription, enriched with doctor and drug referentials sent from a hospital. In this figure, we assume that the embedded database is relational but the choice of the database model (relational, XML, hybrid) has little impact in the global architecture. The document conforms to an XML schema for healthcare, and is wrapped into four tables (two of them being referentials) from the healthcare database schema. As shown in Figure 7, not all documents are wrapped and integrated in the database. Some documents (e.g., an X-ray image) can stay encrypted in the Supporting Servers and simply be referenced by the embedded database.

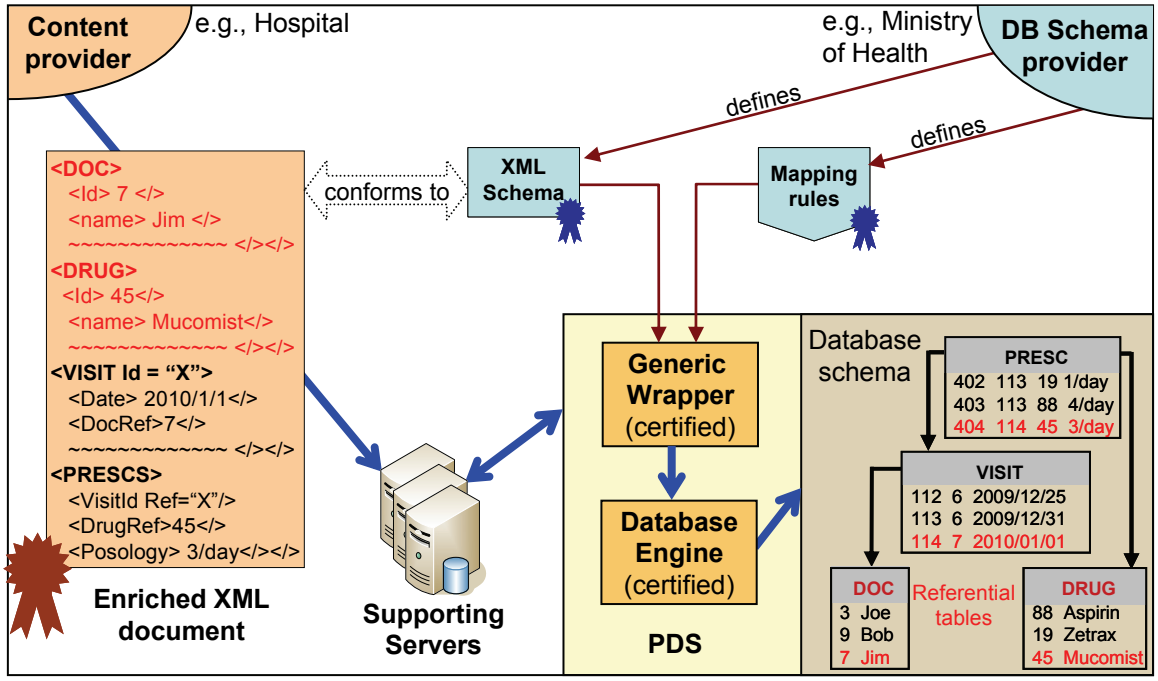

Figure 7: Wrapping a document into the PDS database

Note that problems incurred by the existence of several standards in a given application domain and the problems of data redundancy when database schemas overlap are orthogonal to the PDS approach and are not tackled in this thesis.

3.2.4 Applications

Applications are developed by *Application Providers* (e.g., BestLoan.com). They are defined on top of the published DB schema(s) of interest and can use all database functionalities provided by the embedded DBMS (i.e., DDL and DML statements). Each application defines a set of *collection rules* specifying the subset of documents required to accomplish its purpose (e.g., the five most recent salary forms are required by BestLoan.com). These rules are expressed at the document level to make them meaningful to the PDS holder (helping her to opt in or opt out of this application) and are mapped at the database level to be enforced similarly to access control rules. Applications can run locally (on the holder's PDS with a Graphical User Interface (GUI) on a terminal), on another user's PDS (e.g., on the doctor's one) or on an external server sending queries to the holder's PDS (e.g., the Treasury server computing the holder's carbon tax). While most applications are assumed to perform only selection queries, insertion of new documents is not precluded (e.g., a treatment prescribed at home by the doctor). An updating application will play the same role as a content provider and the insertion will follow the same process.

3.2.5 User Control

The prime way for the PDS holder to control the usage of her data is to opt-in/out of applications and to decide situations where she physically delivers her PDS to another individual (e.g., a doctor). Assuming that the PDS holder's consent has been given, the actions that any individual can perform are regulated by a predefined access control policy. This policy can either be defined by the DB schema provider (e.g., the Ministry of Health fixes a RBAC policy stating the privileges of each category of healthcare professionals according to current legislation) or be defined by the Application Provider and be ratified by a consumer protection association or the legislator.

Predefined access control policies are usually far too complex[11] to be understandable by the PDS holder. It is therefore mandatory to provide the PDS holder with simple tools to protect her sensitive data following her wish. A first way consists in managing the privileges through a simple GUI, as illustrated in the healthcare scenario. A second way is to give the user the ability to mask documents in the database. The records corresponding to a masked document are no longer considered at query execution time, except if the query is issued by the PDS holder herself (through an application). To make this process intuitive, the DB Schema Provider can predefine masking rules (e.g., hide documents by doctor, pathology, time period, etc.) exploiting the expressive power of the DBMS language and easily selectable by the user through a GUI.

The PDS holder (called hereafter the donor) can also impose privacy preserving rules whenever data leaves her PDS to enter another PDS. This sharing is required when a donor's data must be made available while her PDS is disconnected (see the healthcare scenario). This sharing must be ruled by the following principles:

- **Minimal exposure:** in a nominal use, only the results of authorized queries are externalized by a PDS and raw data always remains confined in the PDS. When the donor raw data is made available to others, this must be done in such a way that minimal data (limited collection principle) is exchanged during a minimal duration (limited retention principle) and with the minimum number of recipient PDS (need-to-know principle) to accomplish the purpose of this externalization.
- **Secure delete:** if the donor decides to delete a document before the retention period expires, all replicas of the corresponding raw data hosted by the recipient PDSs must be deleted.

[11] For instance, the RBAC matrix regulating the use of the French EHR contains more than 400 entries, cf. `http://www.dmp.gouv.fr/web/dmp/matrice-d-habilitations-des-professionnels-de-sante` (in French, retrieved on 2012-06-15).

- **Audit:** the donor must have the ability to audit the actions performed by all recipient PDSs on replicas.

Minimal exposure can be implemented by a Secure Publish/Subscribe mechanism working as follows. The raw data to be exchanged (published) is the records belonging to the database view computed over the data targeted by the purpose of the sharing, by intersecting the collection rules of the application, the predefined access control rules applied to the subscribers and the donor's masking rules. The donor publishes these records in an encrypted form on the Supporting Servers. The recipient PDSs subscribe to this data and receive the decryption key once the publisher has accepted the subscription. If the content of the view evolves in the publisher PDS (e.g., because new documents have been inserted), the update is pushed to the subscriber PDSs. We assume that publisher and subscriber PDSs have a compatible database schema (e.g., doctors and patients share a uniform healthcare DB schema).

In the following, we denote by *user control rules* all rules which can be fixed by the PDS holder herself to protect her privacy, namely masking rules, retention rules and audit rules. User control rules are enforced by all PDSs, both on the PDS holder's data and on the data downloaded after a subscription.

3.2.6 Supporting Servers

Supporting Servers Providers provide storage (for encrypted data) and timestamping services to implement the functions that PDSs cannot provide on their own, namely:

- **Asynchronous communication:** since PDSs are often disconnected, documents, shared data and messages must be exchanged asynchronously between Content Providers and PDSs and between PDSs themselves through a storage area.

- **Durability:** the embedded database must be recovered in case of a PDS loss or destruction. The PDS holder's personal data can be recovered from the documents sent by Content Providers through the Supporting Servers (assuming these documents are not destroyed). Data downloaded from other PDSs can be recovered from the data published in the Supporting Servers (assuming their retention limit has not been reached). Other data (user control rules definition, metadata built by applications, etc.) must be saved explicitly by the embedded DBMS on the Supporting Servers (e.g., by sending a message to itself).
- **Global processing:** a temporary storage area is required to implement processing combining data from multiple PDSs. Statistical queries and data anonymization are examples of such processing.
- **Timestamping:** the SPT hardware platform is not equipped with an internal clock since it takes electrical power from the terminal it is plugged in. Hence, a secure time server is required to implement auditing and limited retention.

3.2.7 Security

The security of the architecture lies in:

1. the tamper-resistance of the SPT platform;
2. the certification of the embedded code (and ratification of declarative rules);
3. the encryption of any data externalized in the Supporting Servers.

Regarding encryption, the security of data embedded in a given PDS is considered comparable to the security of the same data stored encrypted in the Supporting Servers as long as the key remains confined to this PDS. Even if any data stored in the Supporting Servers is encrypted, the identity of the users downloading and uploading this data must be obfuscated. Indeed, spying communications could lead to disclosure of sensitive information (e.g., the volume of data sent by a hospital may reveal a heavy pathology). The Supporting Servers provide the storage required to make the

communication asynchronous and the PDS themselves integrate a protocol making these communications anonymous.

The certification does not apply to all parts of the embedded code. Typically, assuming the certification of all embedded applications is unrealistic. Figure 8 shows the elements for which certification is mandatory, namely: (1) the core software (operating system, database engine), (2) the generic XML wrapper, (3) the communication manager, (4) the Publish/Subscribe manager and (5) the privacy manager enforcing the user control rules. Implementing these software pieces and certifying them is the responsibility of the *PDS Providers* (e.g., a SPT manufacturer like Gemalto). Declarative rules need also to be ratified to prove their conformance to a public specification. This data is: (1) the mapping rules consumed by the wrapper, (2) the predefined access control rules, the predefined masking rules and the collection rules enforced by the DBMS. The documents themselves are assumed to be signed to prove their authenticity.

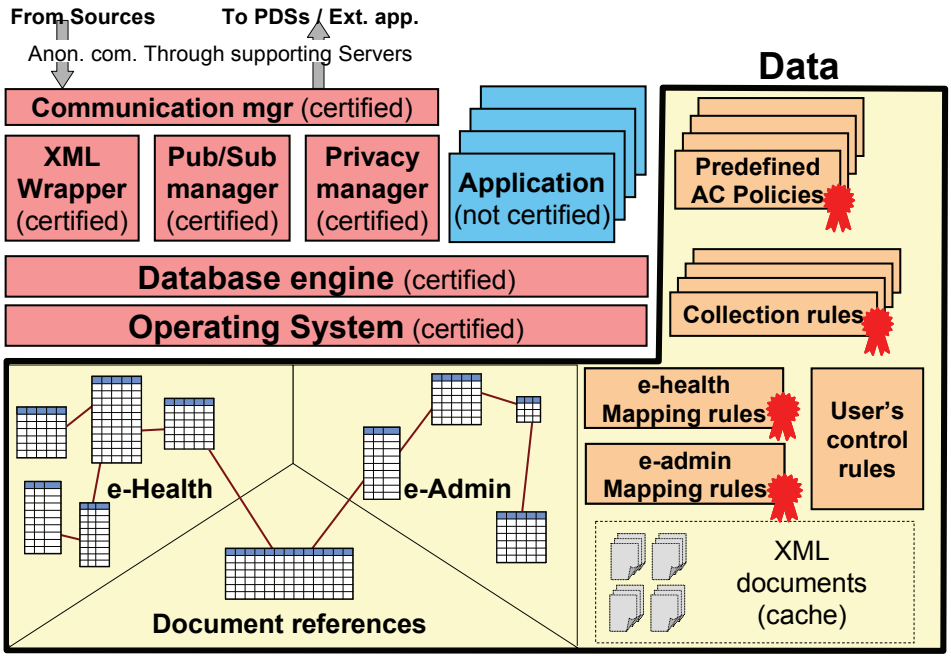

Figure 8: PDS generic software, application and database

Trusting the predefined access control policies requires being able to authenticate all users. Depending on the application domains, PKI infrastructures already serve this purpose. For example, in France, all healthcare professionals have a certificate embedded in a smart card containing their identity and role (a strong authentication is mandatory to access any server hosting healthcare data). In the same spirit, several countries are developing infrastructures based on smart cards or on software certificate to allow any citizen to authenticate electronically (e.g., IdéNum in France).

The NAND Flash remaining unprotected by the tamper-resistance of the microcontroller, cryptographic techniques must be used to protect the database footprint against confidentiality and integrity attacks. Indeed, integrity attacks make sense because the PDS holder herself can try to tamper the database (e.g., she could perform a replay attack to be refunded several times for the same drug or try to change an access control rule or the content of an administrative document, e.g., a diploma).

A primary concern in the PDS context is the granularity of the traditional encryption and hashing algorithms (e.g., 128 bits for AES and 512 bits for SHA). As explained above, the PDS query execution engine must rely on a highly indexed model, thereby generating very fine grain random accesses (in the order of the size of a pointer). Solutions to this problem can be: (1) designing encryption and hashing techniques for fine grain accesses [67] compatible with the SPT resources, (2) designing clustering techniques so that relevant data are contiguous, in the spirit of the PAX models [6] and (3) encrypting the data in such a way that lookups can be done without decrypting the data. The idea here is different from order-preserving encryption or privacy-homomorphism. Roughly speaking, the idea is to exploit the sequentiality of the database to encrypt the data according to their insertion order (hence data having the same clear text get a different cipher text) but equality tests on the cipher text remain possible if they take this order into account. Version management, required to

detect replay attacks, is another complex issue. Maintaining a version number for each page in secure storage (i.e., in the secure stable storage of the microcontroller) is unrealistic considering the small size of the NOR and the fact that it is primarily dedicated to the storage of application code. TEC-Tree [30] overcomes this problem by organizing secret information as a tree. However, it incurs update propagation in the tree, which badly adapts to NAND Flash. Again, our expectation is that the sequential organization of the database can lead to smarter version management techniques. Hence PDS introduces specific interesting challenges in terms of cryptographic techniques applied to database management.

3.3 Durability, availability and global processing

3.3.1 Durability and Availability

Honest but Curious Supporting Servers are assumed to correctly store, retrieve and delete data requests on an unbounded storage area in a durable and highly available way. PDSs capitalize on this to implement higher level secure functions.

Anonymous communications between Content Providers and PDSs and between PDSs themselves can be implemented through the Supporting Servers using an anonymizing network like Tor [28], based on the Onion-routing protocol [35]. The anonymizing network provides a virtual circuit C from the source to the Supporting Servers. Thus, the latter can send data back to the source without knowing its identity, following the *return circuit* C^{-1} encoded in the initial message (this is called *Reply Onions* [35]). An interesting challenge is to use the secure microcontrollers of SPTs to increase the security of anonymous protocols, having SPTs as entry or exit point for the anonymous route.

Recipient PDSs must be able to retrieve messages or data sent to them. Although communications are anonymous, the difficulty lies in selecting the relevant

message/data without disclosing any information that could help the Supporting Servers to infer the identity of the PDS. A protocol tagging messages with anonymous markers [9] can be used to this end. However, the delete request is trickier to implement. First, the physical image of the targeted data should be destroyed by the Supporting Servers (e.g., for cleaning purpose) only if the requesting PDS can exhibit an anonymous proof of legitimacy for this request. Second, the deletion must be effective even if an attacker spies all messages sent to the Supporting Servers and records them. Hence, there is no other solution than removing definitely the access to some data (i.e., by removing the way to decrypt it) even if its image has been stolen and cannot be physically destroyed. To tackle this problem, a protocol [9] based on the Diffie-Hellman key agreement can be used. Note that secure deletion is also a prerequisite to enforce masking and limited retention. Assuming that Supporting Servers guarantee the durability of all messages/data sent to them (except those legitimately destroyed), the log enabling PDSs to recover after a crash or a loss comes for free. Finally, enforcing audit requires a protocol guaranteeing that audit logs are produced and delivered despite unpredictable disconnections of the subscriber and the publisher PDSs.

3.3.2 Global processing

Executing global processing over a set of autonomous trusted PDSs, connecting to Honest but Curious Supporting Servers leads to unusual computations in order to:

1. tackle the unpredictable nature of PDS connections;
2. preserve PDS holders' privacy.

We illustrate this through examples on relational data.

The Ministry of Health would like to prevent a pandemic. It executes a continuous-like query on each PDS that connects to the Supporting Server in order to select individuals having a given set of symptoms. If more than p individuals living in the

same region are at risk, they are encouraged to go to a hospital. However, the patients consent to this form of dynamic queries only if their anonymity is guaranteed. The query can then be of the form '`SELECT pseudonym, city FROM any PDS WHERE symptom IN (x, y, z)`' where pseudonym and city are sent to the querier in clear text through the Supporting Servers. If threshold p is reached, the querier sends messages back tagged with the pseudonym of the individual at risk to the Supporting Servers. Thanks to anonymous communication, a PDS holder can get the outcome of the query for herself without revealing her identity. Interesting issues lie in the organization of the continuous querying protocol, in the classification of the queries which can be managed in this manner and in the conditions to preserve anonymity (i.e., anonymity could be breached if successive queries succeed in recomposing the association between quasi-identifiers and sensitive attributes).

Statistical databases [2] aim at answering aggregate queries (e.g., `SELECT AVG(IQ) FROM … WHERE Age=10 AND Diagnosis='Dyspraxia'`) without compromising sensitive information about individuals. Examples of disclosure control techniques include analyzing the query trail to prevent compromising overlaps between successive queries and/or perturbing the result without affecting the global distribution [78]. An interesting feature of the PDS context is that successive aggregates are computed over a fluctuating population of PDSs (due to the unpredictable nature of PDS connections), making inference among runs harder and influencing the design of disclosure control algorithms accordingly.

Privacy Preserving Data Publishing is another form of global processing aimed at publishing a set of micro-data while protecting the identity of individuals. The traditional process is composed of three phases: data collection, computation of sanitization rules based on the collected data and finally data sanitization. The challenge here is to design a distributed protocol that (1) allows the publisher (through the Supporting Servers) to collect enough data from the targeted PDSs to

compute the sanitization rule, and then (2) delegates the sanitization process itself to the PDSs (so that raw data is never exposed) while providing them a way to control the safety of the sanitization rules. Allard et al. suggested a preliminary solution [7] for a sanitization algorithm preventing record linkages through k-anonymity [73]. Much work remains to be done to prevent from other types of linkages (e.g., attribute linkage prevention through l-diversity [33]).

3.4 User control

Enforcing user control rules, namely masking, limited retention and audit and combining them with application collection rules introduce a set of interesting problems described below:

3.4.1 Impedance mismatch between documents and databases

While predefined access control rules (e.g., RBAC matrix published by an application or by the DB Schema Provider) and queries issued by applications are expressed at the database level (e.g., in SQL), user control rules as well as application collection rules are expressed over documents to be meaningful for the end-user. Conversely, for audit purposes, accesses are recorded at the database level but must be delivered to the end-user at document level in order to interpret them. Consequently, *translation structures* must be integrated in the PDS to store document-to-record and record-to-document links.

The query engine must integrate these links in the query evaluation in order to compute a result compliant with the application collection rules, the predefined access control rules and the user masking rules. The evaluation can be as follows. When a document D (e.g., a medical prescription) is inserted in the database, the records created at wrapping time reference D in the database (records related to referentials like doctors and drugs are not concerned). Let S_c be the set of documents targeted by

the collection rules of application A and S_m be the set of documents targeted by the user masking rules. When A queries the database, the query result includes the document references for each selected record r and this result is post-filtered to keep only the records satisfying $(r \in S_c \wedge r \notin S_m)$. Post-filtering can be implemented efficiently in RAM constrained environments using Bloom filters [17].

When a delete request is issued for D or when D reaches its retention limit, it must be removed from the database. The translation structures are used to identify all records related to this document. This includes the records referencing D either directly (e.g., prescription elements) or transitively (i.e., the referential data like the doctor who does the prescription and the drug prescribed). The presence of referential data in a personal database is sensitive and the related records must be removed as well. The difficulty lies in the fact that referential data may be shared by other documents. A garbage collector algorithm[12] must be designed to tackle this problem. The deletion of the targeted records can be logical (following the marking process evoked in Section 3.3.1) or physical, the latter case being more costly due to the Flash constraints.

3.4.2 Propagating user control rules to other PDSs

If data has been uploaded on the Supporting Servers by a publisher PDS and downloaded by a subscriber PDS, the user control rules defined by the publisher must be propagated to the subscriber. Being able to implement the mechanisms presented above on the subscriber PDS requires sending the user control rules and the translation structures along with the data and forwarding to the subscriber any masking and delete operation performed on the fly by the publisher. Hence, the effect of user control

[12] Storing reference counters is badly adapted to the Flash update constraints. An option can be to recompute counters dynamically.

rules will be the same independently of the location of the data and of the number of replica.

3.5 Conclusion

The vision proposed in this chapter of a secure and portable Personal Data Server is a first contribution in the way people think about management and protection of personal data, thanks to the emergence of new hardware devices combining portability, secure processing and Gigabytes-sized storage. We have presented an initial design for this vision and have identified important technical challenges related to it. Moreover, there already exist experiments in the healthcare field [10] that prefigure the PDS approach and give some confidence about the feasibility of converting the PDS vision into reality.

Chapter 4
Designing a purely sequential database engine

In this chapter, we describe the design of an embedded database engine that could be deployed on SPTs and used within the Personal Data Server context. We propose a new paradigm based on database serialization (managing all database structures in a pure sequential way) and stratification (restructuring them into strata when a scalability limit is reached). We show that a complete DBMS engine can be designed according to this paradigm and, in the next chapter we will demonstrate the effectiveness of the approach through a performance evaluation.

4.1 Design rules and problem statement

The C1 constraint "small RAM/storage ratio" we emphasized in Section 2.1 raises the problem of computing complex queries on gigabytes of data with kilobytes of RAM. This means that not only joins but all operators involved in the query must be evaluated with minimal RAM consumption and intermediate results materialization. This leads to the first design rule:

Rule R1: *Design a massive indexing scheme allowing computation of any combination of selections and joins with minimal RAM.*

Rule R1 leads to define data structures having usually a fine grain read/write/rewrite pattern (massive indexing scheme), thereby generating a huge amount of random

writes. As illustrated in Section 2.4, FTLs usually incur a severe degradation of random writes' performance; this is the reason why several recent works (see Section 2.5 and 2.6) have focused on the impact of NAND Flash constraints on DBMS design and more specifically on index design. Most proposals that address Constraint C4 "NAND Flash behavior" bypass the FTL or minimize random writes, but none of them have been designed to cope with the tiny RAM constraint (C1) of an SPT. For all these reasons, we argue that delegating the optimization of the Flash usage to a FTL does not make sense in our context. Our design considers that the secure MCU has direct access to the NAND Flash[13] but can accommodate Flash access through FTL with minimal extension. This leads us to propose a second design rule:

Rule R2: *Design a database engine matching natively the NAND Flash constraints, i.e., proscribing random writes, despite scarce RAM.*

Rules R1 and R2 are thus conflicting by nature making the problem particularly challenging.

4.2 Proposed approach

4.2.1 Database Serialization

To reconcile rule R1 with R2, we propose a new paradigm called *database serialization*, based on the notion of *Sequentially Written Structures*, defined as follows.

Sequentially Written Structure: *A SWS is a data container satisfying three conditions: (1) its content is written sequentially within the (set of) Flash block(s)*

[13] 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.

allocated to it (i.e., pages already written are never updated nor moved); (2) blocks can be dynamically added to a SWS to expand it at insertion time; (3) allocated blocks are fully reclaimed when obsolete and no partial garbage collection ever occurs within a block.

If all database structures can be organized as SWS, including base data, logs, buffers and all forms of indexes required by rule R1, the net effect of database serialization will be to satisfy rule R2 by construction. Indeed, the SWS definition proscribes random (re)writes. Hence, the dramatic overhead of random writes in Flash is avoided and problematic FTL features are hopefully bypassed: the Translation Layer becomes useless, saving translation costs, the Garbage Collection is done at minimal cost on a block or SWS basis, and Wear Leveling mechanisms are simplified.

The database serialization objective is easy to express but difficult to achieve. It requires solving the following problems:

Base data organization

Natural solutions can be devised to organize the base data as SWSs. 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. Updating or deleting base data is more complex and we address this point separately.

Join Indexes

Join and multiway join indexes can be directly mapped into SWSs as well. Indeed, adding new base data incurs simply appending sequentially new entries in these indexes (sorted join indexes are not considered). Since they are managed is the same way, we call ↓DATA (↓ stands for serialized) the SWSs dedicated to both base data and join indexes.

Selection indexes

Classical indexes (e.g., tree-based or hash-based) are proscribed since inserting new base data would generate random node/bucket updates. We say that an index is *serialized* if the addition of new base data leads to append data to the index. Bitmap index is a basic representative of serialized index. Section 4.3 discusses smarter forms of serialized indexes. While less efficient than classical indexes, they provide a significant gain compared to scanning ↓DATA. We call ↓IND the SWSs dedicated to serialized indexes.

Flash Buffers

↓DATA and particularly ↓IND being made of fine grain elements (tuples, attribute values or index entries), inserting without buffering would lead to waste a lot of space. Indeed, NAND Flash constraints impose writing a new page in a SWS for each elementary insertion. Moreover, the density of a SWS determines the efficiency of scanning it. The objective of buffering is to transform fine-grain to coarse-grain writes in Flash. Elements are gathered into a buffer until they can fill a complete SWS page, which is then flushed. But buffers cannot reside in RAM, partly because of its tiny size and because we cannot assume electrical autonomy and no failure. 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 from different SWSs are buffered together if they can be filled and flushed synchronously. For example, let us assume n ↓INDs indexing different attributes of a same table. They can be buffered together because they are filled at the same rate. When enough data items have been buffered to fill a complete page of each ↓IND, the buffer pages containing them are flushed together. Buffers must be organized themselves as SWSs to comply with the serialization objective. A buffer is actually managed as a sliding

window within its SWS, a *Start* and an *End* markers identifying its active part (i.e., the part not yet flushed). We call ↓BUF the SWSs implementing buffers.

Updates/deletes

Applying updates and deletes directly in a target SWS (↓DATA, ↓IND or ↓BUF) would violate the SWS definition. Instead, updates and deletes are logged in dedicated SWSs, respectively named ↓UPD and ↓DEL. To manage updates, the old and new attribute values of each updated tuple are logged in ↓UPD. At query execution time, ↓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, ↓UPD and ↓DEL are also checked at projection time, to project up-to-date values and to remove deleted tuples from the query result. Overheads are minimized by indexing ↓UPD and ↓DEL on Flash, and building in RAM dedicated data structures to avoid accessing Flash for each result tuple (see Section 4.3.3).

Transaction atomicity

Rolling back a transaction (we restrict ourselves to a single transaction at a time), whatever the reason, imposes undoing all dirty insertions to the SWSs. To avoid the presence of dirty data in ↓DATA and ↓IND, only committed elements of ↓BUF are flushed in their target SWSs as soon as a full 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.

At first glance, database serialization is a powerful paradigm to build a robust and simple design for a DBMS engine complying with rules R1 and R2. However, such a design scales badly. A serialized index cannot compete with classical indexes (e.g., B^+-Tree) and the accumulation over time of elements in ↓UPD and ↓DEL will unavoidably degrade query performance. There is a scalability limit (in terms of ↓IND, ↓UPD and ↓DEL size) after which performance expectation will be violated, this limit being application dependent.

4.2.2 Database Stratification

To tackle this scalability issue, we propose a *stratification principle* transforming a serialized database organization into a more efficient SWS database organization. Let us introduce the notion of *Stratified SWS*:

Stratified SWS: *A stratified SWS denoted by *S, is an optimal reorganization of a SWS S given as input to the stratification process. Optimal reorganization means that the properties of *S are at least as good as if *S was built by a state of the art method ignoring Flash constraints*[14].

Before any stratification occurs, the initial serialized database denoted by ↓DB is composed of serialized SWSs of buffers, base data, indexes, update and delete logs. Thus, $Stratum_0 = \downarrow DB_0 = (\downarrow BUF_0, \downarrow DATA_0, \downarrow IND_0, \downarrow UPD_0, \downarrow DEL_0)$.

When the scalability limit of $Stratum_0$ is reached, $Stratum_1$ needs to be built. The stratification process then starts and triggers three actions.

[14] E.g., as explained in Section 4.4, the stratification of a serialized index can result in an efficient B^+-Tree like structure and can even outperform traditional B^+-Tree.

1. $\downarrow BUF_0$ elements are flushed into their target SWSs (i.e., $\downarrow DATA_0$, $\downarrow IND_0$, $\downarrow UPD_0$, $\downarrow DEL_0$).
2. All new insertions, updates and deletes in the database are directed to $\downarrow DB_1 = (\downarrow BUF_1, \downarrow DATA_1, \downarrow IND_1, \downarrow UPD_1, \downarrow DEL_1)$ until the next stratification phase.
3. The serialized database $\downarrow DB_0$ (which is then frozen) is reorganized in background into a stratified database denoted $*DB_1$, composed of $*DATA_1$ and $*IND_1$. $*DATA_1$ is built by merging $\downarrow DATA_0$ with all updates and deletes registered in $\downarrow UPD_0$ and $\downarrow DEL_0$. $*IND_1$ is the optimal reorganization of $\downarrow IND_0$ (including modifications stored in $\downarrow UPD_0$ and $\downarrow DEL_0$). Hence, $*DB_1$ has the same properties as a regular indexed database updated by random writes.

$Stratum_1$ of the database is thus composed of $\downarrow DB_1$ (receiving new insertions, updates and deletes) and of $*DB_1$, itself composed of $*DATA_1$ and $*IND_1$. When the stratification process terminates (i.e., $*DATA_1$ and $*IND_1$ are completely built), all SWSs from $Stratum_0$ can be reclaimed. $\downarrow DB_1$ keeps growing until the next stratification phase. The next time the scalability limit is reached, a new stratification step occurs. Stratification is then an iterative mechanism summarized as follows (see also Figure 9):

Stratify($Stratum_i$) → $Stratum_{i+1}$

1 $\downarrow DB_{i+1}=\emptyset$ //note that $*DATA_0=\emptyset$, $*IND_0=\emptyset$

2 Flush($\downarrow BUF_i$)in $\downarrow DATA_i$,$\downarrow IND_i$,$\downarrow UPD_i$,$\downarrow DEL_i$

3 Reclaim($\downarrow BUF_i$)

4 $*DATA_{i+1}$=Merge($*DATA_i$,$\downarrow DATA_i$,$\downarrow UPD_i$,$\downarrow DEL_i$)

5 $*IND_{i+1}$=StratifyIndex($*IND_i$,$\downarrow IND_i$,$\downarrow UPD_i$,$\downarrow DEL_i$)

6 Reclaim($*DATA_i$,$*IND_i$,$\downarrow DATA_i$,$\downarrow IND_i$,$\downarrow UPD_i$,$\downarrow DEL_i$)

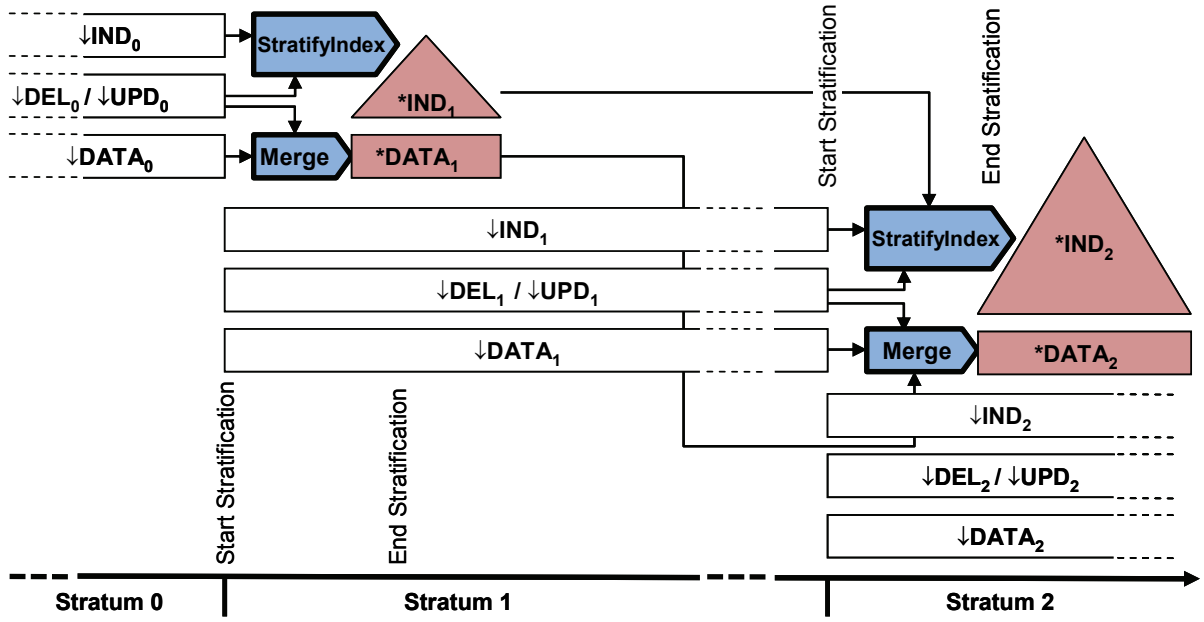

Figure 9: The stratification process

Rule R1 is preserved since $*IND_{i+1}$ is assumed to provide an optimal indexing scheme and Rule R2 is preserved since random writes are never produced despite the reorganization (new SWSs are written sequentially and old SWSs are fully reclaimed). As such, stratification is very different in spirit from batch approaches deferring updates thanks to a log since such deferred updates produce random rewrites. The price to pay is however a complete reconstruction of $*DB_i$ at each step[15]. This raises three important remarks:

Maximize the size of $\downarrow DB_i$

According to Yao's formula [79], little locality can be expected when reporting $\downarrow DB_i$ elements into $*DB_{i+1}$. Hence, the stratification cost is determined by the size of $*DB_i$, considering that, after several stratifications, $size(*DB_i) >> size(\downarrow DB_i)$. There is thus

[15] Note that other solutions could be envisioned, e.g., several Strata could coexist (i.e., $\downarrow DB_i$ is restructured but not merged with $*DB_i$) reducing stratification costs but increasing query costs.

a high benefit to maximize the size of $\downarrow DB_i$ to reduce the number of stratification steps, thereby decreasing the global cost of stratification.

Allow stopping / resuming the stratification process

The stratification is a long process that should preferably be done during idle times. However, the database may be queried while a stratification process is in progress. It is thus necessary to be able to stop and resume it with minimal overhead. The scarce RAM constraint actually provides a simple and efficient way to achieve this. A Brute-force strategy consists in flushing the RAM content to Flash memory when stopping[16]. This requires a couple of milliseconds (e.g., about 20 ms for 128 KB RAM). To resume the process, the RAM content is read back from Flash and the Flash block used is erased, again at low cost (7 ms for 128 KB RAM).

Take advantage of partial stratification

As a consequence of the previous point, queries can be run while stratification is incomplete. This is not a problem because SWS from the previous stratum are reclaimed only at the end of stratification and can then serve to answer the query. Assuming that each index is stratified independently, the query can even take advantage of the indexes already stratified.

4.3 Serialization techniques

Section 4.3.1 shows how serialized indexes answer the requirement expressed by design rule R2. Section 4.3.2 illustrates how to build the massive indexing scheme required by rule R1 on this basis. Finally, Section 4.3.3 shows how updates and deletes can be handled without violating rule R2.

[16] This is similar to the Suspend-to-disk, commonly named "Hibernation" in most operating systems.

4.3.1 Serialized Indexes

With no index, finding matching tuples leads to a sequential scan of the whole table (see Figure 10.a). Brute-force serialized indexes can be created by replicating the indexed attribute (called *key*) into a SWS called *Key Area* (*KA*) (see Figure 10.b). The row identifiers of matching tuples are found by a sequential scan of KA, thus the name *Full Index Scan*. *Full Index Scan* is trivial and works for any exact match or range predicate. For variable or large size keys, a collision resistant hash of the key can be used in place of the key itself (restricting the index use to exact match predicates). For keys varying on a small cardinality domain, the key can also be advantageously replaced by a bitmap encoding. Adequate bitmap encoding can be selected to support range predicates [23].

Smarter forms of serialized indexes can be devised to support exact-match predicates using *Bloom Filters* (*BF*) [17]. A Bloom Filter can represent a set of values of arbitrary length in a compact way and allows probabilistic membership queries with no false negatives and a very low rate of false positive. For example, the false positive rate produced by a Bloom Filter built using 3 hash functions and allocating 12 bits per value is 0.1, it decreases to 0.05 with 16 bits per value and to only 0.006 with 4 hash functions [17]. Hence, they provide a very flexible way to trade space with performance. In our context, Bloom Filters are used to summarize KA, by building one Bloom Filter for each flash page of KA. Finding matching tuples can then be achieved by a full scan of the KA summary (denoted *SKA* in Figure 10.c), followed by a direct access to the KA pages containing a result (or a false positive with a tiny probability). Only those KA pages are finally scanned, thereby saving many useless I/Os in case of selective predicates. For low selectivity predicates, this index, called *Summary Scan*, becomes less efficient since it qualifies a large number of KA pages.

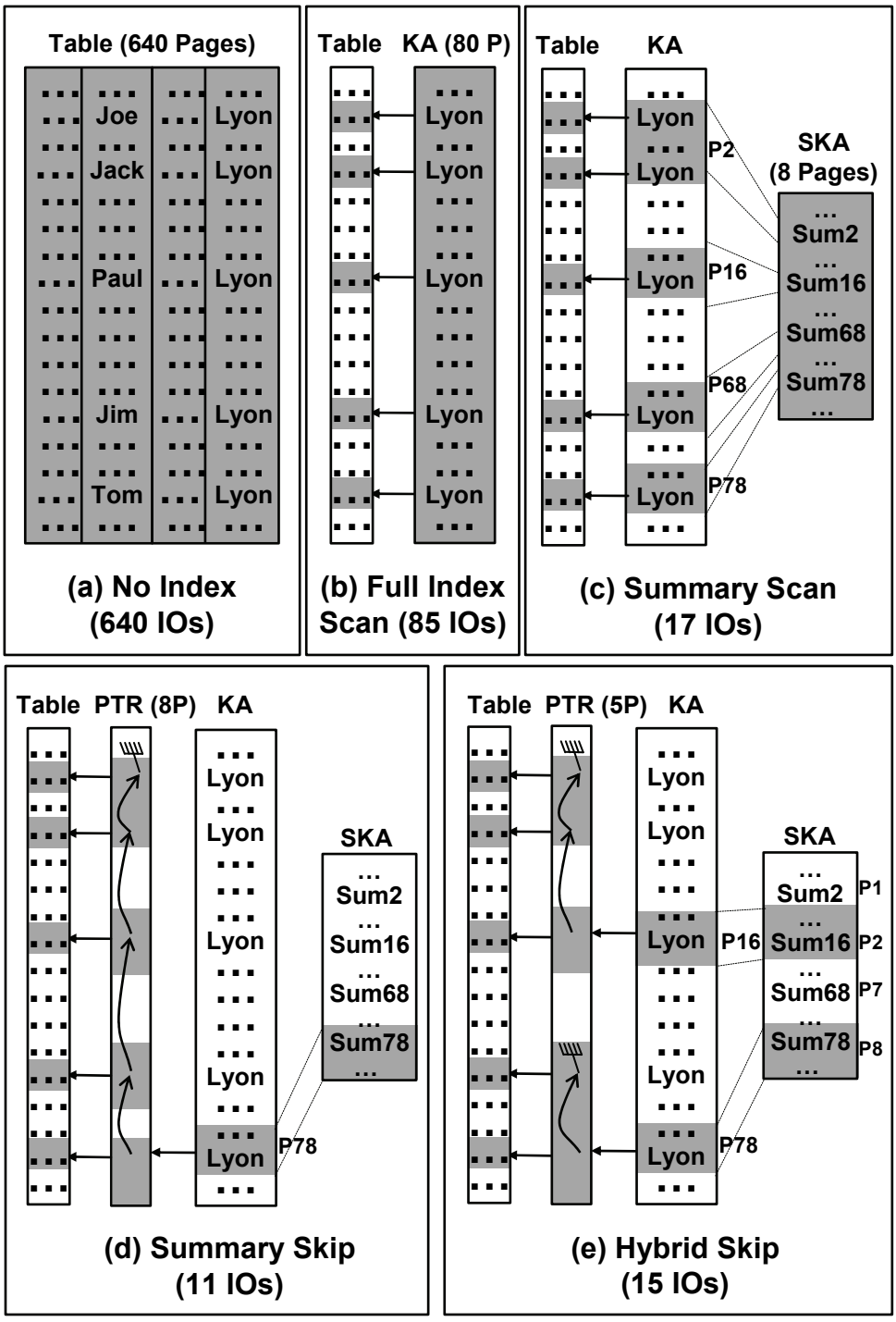

Figure 10: Serialized Indexes (grey areas are effectively accessed)

Given the sequential structure of a serialized index, Summary Scan can be further optimized by chaining index entries sharing the same key value. The chains are stored in a SWS called *PTR* (see Figure 10.d). To cope with the SWS constraints, the

chaining must be backward, i.e., the occurrence *n+1* of a given key value references occurrence *n*. SKA and KA are then used only for searching the last occurrence of the searched key (see Figure 10.d). If a key value appears *n* times in the index, in average, *1/n* of SKA and 1 page of KA will be scanned. Then, the pointer chain is followed to find all matching tuples' row identifiers. While this index, called *Summary Skip* is efficient for lookups, maintaining the pointer chain can however be costly when keys have few occurrences. It leads, in the worst case to a full scan of SKA to find the previous occurrence of the inserted key.

To bound the insertion cost, a solution is to specify the maximal number of SKA pages, say *n*, that should be scanned before stopping. If the previous occurrence of the key is not found once this limit is reached, the new key is inserted with a preceding pointer set to *NULL*, thereby breaking the pointer chain. At query time, when a *NULL* pointer is encountered, the algorithm switches back to SKA, skipping the *n* next pages of SKA, and continues searching the preceding key occurrence according to the Summary Scan strategy. Indeed, by construction, the KA pages summarized by these *n* pages of SKA cannot contain the searched value (otherwise, they would have been chained). We call this strategy *Hybrid Skip* since it mixes Summary Scan and Summary Skip (see Figure 10.e). This strategy has two positive impacts: (1) at insertion time, and contrary to Summary Skip, the search cost in SKA is bounded to *n* page accesses independently of the index size; (2) the pointer can be encoded on a smaller number of bits (linked to the number of keys summarized by *n* pages of SKA) thereby reducing the access cost of PTR.

Thanks to their sequential structure, all serialized indexes have the property of producing an ordered list of matching tuples' row identifiers (corresponding to the insertion order of these tuples). This property is essential to combine partial results of several selections because it allows efficient merges with no RAM consumption, as we will explain in the next section.

4.3.2 Querying Serialized DB with Small RAM

Organizing the whole database with SWSs (Rule R2), and using serialized indexes as basic constructs for a massive indexing scheme (Rule R1) are the cornerstones of the embedded engine design. This section describes a candidate massive indexing scheme and its associated query execution model. This model fits well our requirements in terms of query performance but other indexing schemes could be considered and be incorporated in the database serialization/stratification paradigm. For clarity, we illustrate the proposed model on the same database schema as the one used in the performance chapter.

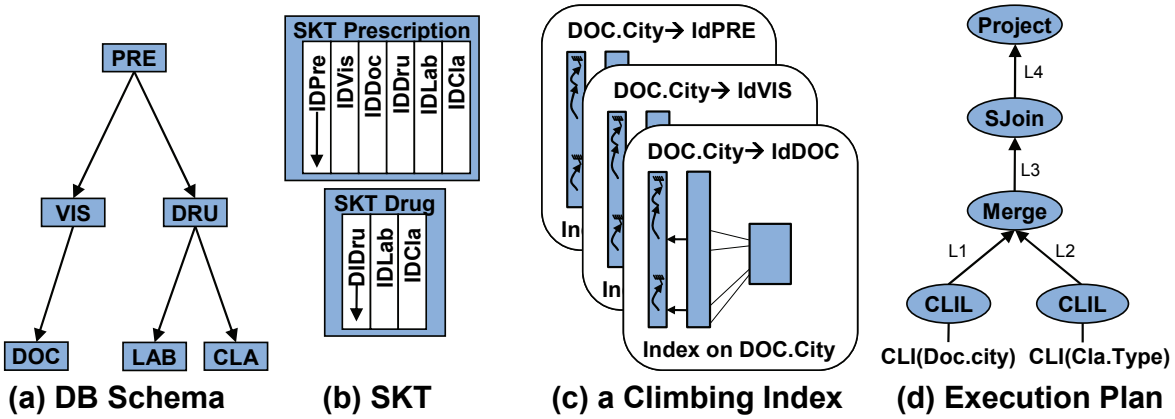

Figure 11: Query processing with SKTs and climbing indexes

As shown in Figure 11.a, we consider a tree-based medical database schema where table *Prescription* (*Pre*), the schema root, references tables *Visit* (*Vis*) and *Drug* (*Dru*), table *Vis* references *Doctor* (*Doc*) and table *Dru* references *Laboratory* (*Lab*) and *ClassOfDrug* (*Cla*).

Multi-way join indexes called *Subtree Key Tables* (SKTs) are created for each node tables of the tree-based schema. Each SKT joins all tables in a subtree to the subtree root and stores the result sorted on the identifiers of the root table (SKT entries contain the ids of the matching tuples). For example, the SKT_{Pre} rooted at *Pre* is composed of the joining sets of ids $\{id_{Pre}, id_{Vis}, id_{Dru}, id_{Doc}, id_{Lab}, id_{Cla}\}$ sorted on id_{Pre}.

This enables a query to directly associate a prescription with, e.g., the doctor who prescribed it. Selection indexes called *Climbing Indexes* (CLI) are created on all attributes involved in selection predicates. A CLI created on attribute A of table T maps A values to lists of identifiers of T, as well as lists of identifiers of each table T' ancestor of T in the tree-based schema (Figure 11.a). For example, in $CLI_{Doc.City}$, the value *'Lyon'* is mapped to lists of id_{Doc}, id_{Vis}, and id_{Pre} identifiers. Combined together, SKTs and CLIs allow selecting tuples in any table, reaching any other table in the path from this table to the root table in a single step and projecting attributes from any other table of the tree.

Yin et al. proposed this indexing scheme in a previous work to deal with the RAM constraint in a read-only embedded database context [81]. Hence, the contribution here is not on the indexing model itself but on the way indexing models (including this one) can be adapted to support inserts, updates and deletes while tackling the NAND Flash constraints. The solution is provided by the serialization/stratification paradigm. The primary step is to map each concept of the considered indexing scheme into SWSs. Once this achieved, the model will inherit all properties of SWSs in terms of update management (inserts/updates/deletes) in Flash, transaction atomicity and scalability (stratification).

The massive indexing scheme introduced above is mapped into SWSs as follows. Base tables and SKTs are directly mapped into SWSs since both kind of structures are sequential by nature. The climbing index structures are trickier to map. The initial design of the CLI structure presented in [81] relied on a B^+-Tree where each key value was associated to a set of inverted lists of tuple identifiers (one list for each level of the database schema, from the table the indexed attribute belongs to up to the root table). The SWS mapping of CLI consists of building one serialized index for each level of CLI, as pictured in Figure 11.c. For example, $CLI_{Doc.City}$ is made of three serialized indexes delivering respectively the lists of id_{Doc}, id_{Vis}, and id_{Pre} identifiers.

This mapping being defined, the query execution is as follows. Figure 11.d shows the Query Execution Plan (QEP) of a query which joins all the tables, evaluates the selection predicates on two indexed attributes (*Doc.City='Lyon' AND Cla.Type='AZT'*), and projects some attributes. The operators required to execute this query are:

1. *CLIL(CLI, P, π)→{id_T}* looks up in the climbing index CLI and delivers the list of sorted IDs referencing the table selected by π and satisfying a predicate P of the form (*attribute θ value*) or (*attribute $\in$ {value}*);
2. *Merge($\cap i\{\cup j\{id_T\}\}$)→{id_T}* performs the unions and intersections of a collection of sorted lists of identifiers of the same table T translating a logical expression over T expressed in conjunctive normal form;
3. *Sjoin({id_T}, SKT_T, π)→{<id_T, id_{Ti}, id_{Tj} …>}* performs a key semi-join between a list of identifiers of a table T and SKT_T, and projects the result on the subset of SKT_T attributes selected by π; this result is sorted on id_T; Conceptually, this operation implements a Join but its cost sums up to read the right SKT entries.
4. *Project({<id_T, id_{Ti}, id_{Tj} … >}, π) → {< Att_i, Att_j, Att_k … >}* follows tuples identifiers and retrieves attributes selected by π; the attribute values are buffered in RAM in a hierarchical cache keeping most frequent values to avoid Flash I/Os.

The query can be executed in a pipeline fashion as follows:

```
1   Project (L4,<Doc.Name,Dru.Name, Pre.Qty> → Result
2   SJoin (L3, SKTPre, <idPre, idDru, iddoc>) → L4
3     Merge (L1∩L2}) → L3
4       CLIL (Cla.Type, ='AZT', Pre) → L2
5       CLIL (Doc.City, ='Lyon', Pre) → L1
```

The RAM consumption for this query is limited to one Flash page per CLI, the rest of the RAM being used for projection (cache).

4.3.3 Processing Updates and Deletes

As explained in Section 4.2, updates and deletes are logged in dedicated SWSs, named ↓UPD and ↓DEL, rather than being executed in place. This virtualization implies compensating queries at execution time, by combining ↓DATA, ↓IND and ↓BUF with ↓UPD and ↓DEL to compute the correct result.

Regarding deletes, each result tuple which has been recorded in ↓DEL must be withdrawn from the query result. To perform this check efficiently, a dedicated structure *DEL_RAM* is built in RAM to avoid accessing ↓DEL on Flash for each result tuple. To limit RAM occupancy, only the identifiers of the deleted tuples, excluding cascading deletes, are stored in DEL_RAM. In a tree-based DB schema, deleting a tuple in a leaf table (e.g., one tuple d in *Doctor*) may incur cascading the deletes up to the root table (e.g., all *Visit* and *Prescription* tuples linked to d). Only d identifier is recorded in this case. At query execution end, the *SJoin* operator accesses the SKT of the root table of the query, to get the identifiers of all (node table) tuples used to form the tuples of the query result. At that time, each of these identifiers is probed with DEL_RAM and the tuple is discarded in the positive case, without inducing any additional IO. Coming back to our example, if a query applies to *Prescription* and selects a prescription p performed by the deleted doctor d, *SJoin* will return d identifier from the *Prescription* SKT, d identifier will be positively probed with DEL_RAM and p will be discarded.

Regarding updates, old and new attribute values are logged in ↓UPD for each updated tuple. To compensate the query, the query processor must (i) for each projected attribute, check its presence in ↓UPD and get its up-to-date value in the positive case; (ii) compensate index accesses to eliminate false positives, i.e., tuples returned by the index based on their old value (in ↓BUF, ↓IND) but which should be discarded based on their new value (in ↓UPD); and (iii) compensate index accesses to integrate false

negatives, i.e., tuples matching the query based on their new value but not returned by the indexes based on their old value. Those three steps are detailed below:

Projections

Similarly to the delete case, a structure *UPD_RAM* is maintained in RAM to speedup the membership test in ↓UPD. UPD_RAM stores the addresses of modified values in ↓DATA and is rebuilt at each session by scanning ↓UPD. Each attribute (address) to be projected is probed with UPD_RAM, and only when the probe is positive, ↓UPD is accessed on Flash. In addition, ↓UPD is indexed on the attribute addresses in a way similar to a ↓DATA SWS, thereby drastically reducing the overhead caused by update processing at project time.

Selections – removing false positives

The set of false positive tuples (identifiers) is extracted from ↓UPD and stored in a RAM structure called *FP_RAM*. FP_RAM is used to probe each output of the index scan. For example, to get *the doctors living in 'Paris'* using the index *Doc.City* excluding false positives, we (1) retrieve from ↓UPD *the doctors who left 'Paris'* (since the last stratification), (2) store their IDs in FP_RAM, (3) probe each result of the index with FP_RAM and (4) discard positive ones. Tackling climbing index accesses is trickier. For example, for a climbing index access on *Doc.City* at level *Pre* (i.e., *the prescriptions of doctors living in 'Paris'*) the result is a list of prescriptions, while ↓UPD provides doctors identifiers. Two solutions can be envisioned: (1) for each index result (prescription) the corresponding doctor is retrieved, e.g., by placing a *SJoin* after the index access, and is probed with FP_RAM; (2) for each element of FP_RAM (doctors) the corresponding prescriptions are found, e.g., by a subquery into the climbing index on doctors' IDs. In practice, we favor option (2) because the number of updated values is likely to be much lower than the number of results returned by the index. To avoid executing these same subqueries at each index lookup, we execute

them once at update time and materialize their result in ↓UPD (hence ↓UPD contains the identifiers of all tuples referencing an updated tuple, by a – path of – foreign key). At query time, FP_RAM is built from this basis, using a serialized index on ↓UPD defined on the old attribute value. This drastically reduces the cost of false positives removal for an acceptable space overhead (only identifiers are stored).

Selections – Integrating false negatives

The set of false negative tuples (identifiers) is extracted from ↓UPD and must be merged with the index output. For example, to get *the doctors living in 'Paris'* using the index on *Doc.City*, including false negatives, we (1) retrieve from ↓UPD *the doctors who moved to 'Paris'* (since the last stratification) and (2) merge the results. For a climbing index access, we use the same technique as presented above. For example, for a climbing index access on *Doc.City* at level *Pre* (i.e., *the prescriptions of doctors living in 'Paris'*), each doctor tuple in ↓UPD is linked to the (sorted) set of its prescriptions which have simply to be merged with the index output. Using this technique, the integration of false negatives into the index output generates a very small overhead, given that ↓UPD is also indexed on the new attribute value (speeding up the retrieval of false negative identifiers).

A check of ↓UPD may return several entries when an attribute from the same record has been updated successively between two stratifications, and the proposed technique needs to be adapted to inspect all entries that match the queried attribute value. With our example query, a doctor who moved from 'Paris' and came back later will have two entries in the update log, a first with 'Paris' as old value and a second with 'Paris' as new value. In this case, the doctor should not appear in FP_RAM, because the last entry contains a new value identical to the queried value ('Paris'); she should not be considered as a false negative either, because there's an entry present with the queried value as the old value. It is also possible to resort to a simpler brute-force approach: in

the event of an ID being both a false positive and false negative, checking the new value of the last entry in ↓UPD for this attribute will resolve the ambiguity.

The generalization of these techniques to multi-predicate queries is straightforward. Regarding the query processing techniques presented in Section 4.3.2, only operators *CLIL* and *Project* are impacted by the integration of updates and deletes, as described above.

Regarding RAM consumption, three types of structures are needed: DEL_RAM stores the deleted tuples identifiers, UPD_RAM stores the addresses of updated attributes, and FP_RAM stores the identifiers of modified tuples for an index access (its size is negligible compared with DEL_RAM and UPD_RAM). For instance, in the experiments conducted in Chapter 5, the RAM consumption corresponding to 3000 deletes and 3000 updates was about 18 KB.

4.4 Stratification techniques

4.4.1 Stratified Indexes

The construction of stratified indexes at stratification time can take advantage of three properties: (1) stratified indexes are never updated (SWS) and can then rely on more space and time efficient data structures than traditional B$^+$-Tree (100% of space occupancy can be reached compared to about 75%); (2) all the data items to be indexed are known in advance; (3) the complete RAM can be dedicated to the index construction since the stratification can be stopped/resumed with low overhead (see Section 4.2.2).

The constraint however is to organize the stratified index in such a way that the lists of row 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 or from a range search in one index. Moreover, this ordering allows combining query results in $\downarrow IND_i$ (serialized indexes) and in $*IND_i$ (stratified indexes) by a simple concatenation of the lists of matching tuples (see Figure 12). Indeed, $*IND_i$ tuple always precede $\downarrow IND_i$ tuple in their insertion order and so are the value of their identifiers.

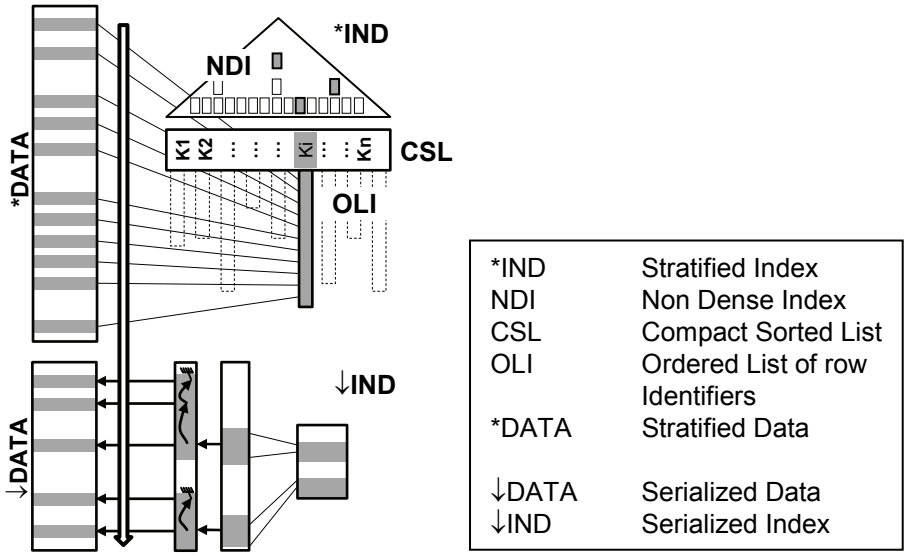

Figure 12: Stratified index and combination of $\downarrow DB_i$ and $*DB_i$

This way of combining $\downarrow IND_i$ and $*IND_i$ leads to a straightforward extension of the query processing and update/delete management techniques to tackle a combination of serialized and stratified databases. Actually, all operators described in Section 4.3.2 must be extended to consider both elements of $*IND_i$ and of $\downarrow IND_i$ using simple concatenations.

The resulting stratified index structure is as follows. A compact ordered list of row identifiers (OLI in Figure 12), is built for each index entry K_i. The set of index entries is represented by a compact sorted list, named CSL. Finally, a compact non-dense index, named NDI, stores the highest key of each page of CSL 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 NDI. Stratified indexes are efficient and

highly compact (100% of space occupancy) because they are built statically and are never updated.

4.4.2 Stratification Algorithms

In our approach, stratifying a database sums up to process the *Merge* and *StratifyIndex* operations introduced in Section 4.2.2. For the sake of clarity, indices are removed from the notations and refer to Stratum$_i$ (unless specified). We also remove the buffers structure (↓BUF) considering that they are flushed into ↓DB just before stratification starts. The stratification strategy is presented assuming the size of *DB is much greater than ↓DB, which is the normal case after some initial stratification steps. We present successively the algorithms used to stratify the index part and the data part.

4.4.3 StratifyIndex(*IND, ↓IND, ↓UPD, ↓DEL)

Different structures are involved in index stratification (see Figure 9). Let us first analyze their organization to deduce an adequate strategy given the constraints of the microcontroller:

- *IND contains a set of climbing indexes organized by index key. For simplicity, we consider a single climbing index CLI made of a set of sorted keys, each key being linked to a list of sorted tuple IDs sharing that key (see *IND on Figure 12). The extension to a set of climbing indexes each made of several lists of tuple IDs is straightforward.
- ↓UPD and ↓DEL are not sorted since elements are added sequentially when updates and deletes are performed.
- Finally, ↓IND contains a set of serialized indexes organized by tuple IDs (built sequentially).

These structures being organized differently, the stratification process appears to be complex, especially with large data structures and scarce RAM. We explain the strategy for each structure in isolation, and then consider the whole picture:

Stratifying ↓IND

For a single stratified index *I and a serialized index ↓I, StratifyIndex works as follows. Since the size of *I is much greater than ↓I, the best solution is reorganizing ↓I such that it complies with *I organization. This allows merging both structures to obtain a new stratified index *I' without RAM consumption. Conversely, reorganizing *I to make it compliant with ↓I or querying ↓I for each *I element is not an option in terms of performance.

Applying ↓DEL

How logged deletes (↓DEL) can be applied on *I? We could have considered storing in RAM tuple IDs of deleted tuples, as it is done for queries. This would however impose to access the SKT for all IDs of the index since ↓DEL does not contain tuple IDs of cascaded deletes. Computing the cascaded version of ↓DEL is not a solution because it may not fit in RAM, and because accessing it in flash for each ID would be too costly. The only practical solution is again to reorganize ↓DEL such that it complies with *I organization. This means (1) 'cascading' ↓DEL; (2) projecting ↓DEL on all indexed attributes; (3) sorting the result on (TabID, AttID, Key, TupId). While this reorganization is costly, it is done only once for the whole stratification.

Applying ↓UPD

Applying logged updates on *I leads to a similar strategy as applying ↓DEL for similar reasons. Cascading ↓UPD is also necessary because of the structure of climbing indexes. Consider for instance a doctor who moved from *Nice* to *Paris*. When the index $CLI_{Doc.City}$ is stratified, all prescription IDs in the *Nice* entry must be removed and

added to the *Paris* entry. Thus, we build two structures, ↓UPD_FP for false positives, and ↓UPD_FN for false negatives. Both are organized on (TabID, AttID, Key, TupId) to enable an efficient merge with *I. Thus, reorganizing ↓UPD leads to (1) 'cascading' ↓UPD, (2) splitting it in ↓UPD_FP (resp. ↓UPD_FN) having as key the old value (resp. the new value); (3) sort ↓UPD_FP and ↓UPD_FN on (TabID, AttID, Key, TupId). Note that this reorganization is less costly than for ↓DEL since it only considers 2 keys per entry in ↓UPD (while ↓DEL considers n keys, n being the number of indexed attributes). As for ↓DEL, this reorganization is done once per stratification.

The following pseudo code summarizes the StratifyIndex operation considering *IND, ↓IND, ↓DEL and ↓UPD:

```
StratifyIndex(*IND,↓IND,↓UPD,↓DEL)
1    D+= Cascade(↓DEL)
2    D+π=Project_on_all_indexed_attributes(D*)
3    D= Sort(D+π) on (TabID,AttID, Key, TupId)
4    U+= Cascade(↓UPD)
5    U_FP=Sort(U+)on(TabID,AttID,OldVal,TupID)
6    U_FN=Sort(U+)on(TabID,AttID,NewVal,TupID)
7    For T iterating on all tables
8      For A iterating of all Attributes of T
9        For O iterating on {T} ∪ {T.ancestors}
10         S = Sort(↓IND(T,A,O)) on Key, TupID
11   *IND_{i+1}(T,A,O)=PStratify(*IND(T,A,O), ↓IND(T,A,O))- D - U_FP + U_FN
```

The cascade operation (lines 1 and 4) can be done thanks to the climbing indexes defined on TupIDs. In line 2, the projected value is easy to retrieve thanks to the TupID. The sort operations (line 3, 5, 6 and 10) are done by sort merge using the whole. Finally, the PStratify operation (line 11) means pipelined stratification. Indeed, PStratify can be performed in pipeline, key by key, since all operands are sorted on (TabID, AttID, Key, TupID).

4.4.4 Merge(*DATA, ↓DATA, ↓UPD, ↓DEL)

*DATA and ↓DATA contain a set of tables and SKTs, each ordered by tuple ID. The merge of *DATA with ↓DATA, ↓DEL and ↓UPD is thus rather simple:

1. ↓DATA is simply concatenated to *DATA.
2. Since SKT is accessible, ↓DEL does not need to be cascaded and can be kept in RAM during ↓DATA stratification. The structure (DEL_RAM) and mode of operation to identify the deleted tuples is the same as for queries (see Section 4.3.3).
3. Similarly, the updated tuple IDs are kept in RAM (with no cascading) during ↓DATA stratification in the (UPD_RAM) structure and ↓UPD is only queried for tuples that have been actually updated, as for queries (see Section 4.3.3).

The resulting algorithm is as follows:

```
Merge(*DATA, ↓DATA, ↓UPD, ↓DEL)
1    Load DEL_RAM in RAM
2    Load UPD_RAM in RAM
3    For T iterating on all tables
4      For each tuple r of *DATA(T)||↓DATA(T)
5        Access r's SKT attributes, check DEL_RAM, if positive, replace r by
         empty space
6        Check UPD_RAM, if positive retrieve new values from ↓UPD and update r
7        Add r to *DATA_{i+1}(T)
```

4.5 Conclusion

This chapter proposed a comprehensive design for a purely sequential database engine with the objective to disseminate databases everywhere, up to the lightest smart objects. Inspired by low cost economic models, we considered the simplest and cheapest form of computer available today, that is a microcontroller equipped with

external Flash storage. We have presented a new paradigm, named database serialization and stratification, tackling the conflicting NAND Flash and tiny RAM constraints inherent to these devices and have shown its effectiveness to build a complete embedded DBMS engine.

The stratification cost is related to the size of the serialized database (↓DB), to the number of deletes and updates performed since last stratification, and to the size of the stratified database (*DB). The essential features of the stratification proposed in this chapter are (1) that the stratified part of the database is read and written only once; (2) that the process can run in background, be interrupted then resumed, without hindering queries on the database; and (3) that the stratification process is by nature failure-resistant (e.g., sudden power loss) since no element of ↓DB is reclaimed before the completion of the stratification.

Chapter 5
Performance evaluation

In this chapter, we show experimentally the effectiveness of the proposed solutions designed for the purely sequential database engine. In the first section, we describe the simulation platform we implemented to evaluate the performance of the components from our embedded engine. Then, we measure the response time of a basic SPT implementation featuring our techniques. Finally, we discuss the impact of the serialized storage model through a performance simulation of its access pattern on different Flash-based devices.

5.1 Platform Simulation

Developing embedded software for MCU is a complex process, done in two phases: (1) development and validation on simulators, (2) adaptation and flashing on MCU. We first developed a software simulator of the target hardware platform: a MCU equipped with a 50 MHz CPU, 64 KB of RAM and 1 MB of NOR-Flash, connected to a 4GB external NAND Flash. This simulator is IO-accurate, i.e., it computes exactly the number of page read, write and block erase operations. In order to provide

performance results in seconds, we calibrated[17] (see Table 1) the output of this simulation with performance measurements done on a previous prototype named PlugDB. PlugDB has already reached phase 2 (i.e., runs on a real hardware platform), has been demonstrated [10] and is being experimented in the field to manage personal healthcare folders. While PlugDB is simpler than our database engine (simpler indexes and no stratification), it shares enough commonalities with our design to allow this calibration.

Table 1: Performance parameters defined for the SPT simulator

Category	Description	Value
Flash	Flash sector size	512 B
	Flash page size	2 KB (4 sectors)
	Read one Flash sectors (transfer time included)	40 µs/IO
	Write one Flash page (transfer time included)	300 µs/IO
	Transfer one byte from NAND register to RAM	0.025 µs/B
CPU	Frequency	50 MHz
Bloom Filter	Number of hash functions	3
	Encoding size for each key	2 B
	False positive rate	0.005 %

5.1.1 Insertion cost and tuple lifecycle

This section assesses the benefit of serialization and stratification in terms of write I/Os to the NAND Flash.

We consider a synthetic medical database (see Figure 11 and Table 2) with cardinalities similar to TPC-H (with a scale factor $SF = 0.5$, leading to 3M tuples for

[17] 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.

the largest table *Prescription*). Each of the 6 tables has 5 indexed attributes (indicated by an asterisk appended to the attribute name): 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. The serialized part of all climbing indexes is implemented as *Hybrid Skip*. Including the required foreign keys (in italic font) and other non-indexed attributes, the tuple size reaches 160 bytes. Crossed cells represent non existing instances. We also built *Subtree Key Tables* on tables *Prescription*, *Visit* and *Drug*. The tables are populated uniformly. This massively indexed schema holds 30 climbing indexes, translated into 64 serialized indexes, from which 29 for the *Prescription* table (5 from each sub-table + 4 defined at *Prescription* level).

Table 2: Synthetic medical database schema and cardinalities

Attribute \ **Table**		Pre	Vis	Doc	Dru	Lab	Cla
ID*	INTEGER	3M	75K	7500	400K	5K	10K
Dup10*	CHAR(10)	300K	7500	750	40K	500	1K
Dup100*	CHAR(10)	30K	750	75	4K	50	100
MS1*	CHAR(10)	100	100	100	100	100	100
MS10*	CHAR(10)	10	10	10	10	10	10
A1	CHAR(10)	3M	75K	7500	400K	5K	10K
A2	INTEGER	3M	75K	7500	400K	5K	10K
A3	DATE	3M	75K	7500	400K	5K	10K
Comment	CHAR(98)	3M			400K		
Comment	CHAR(94)		75K				
Comment	CHAR(90)			7500		5K	10K
IDVIS	INTEGER	75K					
IDDOC	INTEGER		7500				
IDDRU	INTEGER	400K					
IDLAB	INTEGER				5K		
IDCLA	INTEGER				10K		

Let us first study the insertion cost of a single tuple. The number of serialized indexes depends on the table, and so is the insertion cost (note that *Hybrid Skip* insertion cost is rather stable with respect to the selectivity – see Section 4.3.1 and Figure 10). Insertion cost varies between 1.3 ms for the *Doctor* table to about 7 ms for the *Prescription* table, and then does not appear as 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 [69] 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 (*Doctor)* to 4.2 s (*Prescription*). The minimal insertion cost (best SD card, table *Doctor*) is 42 ms while the maximal cost (worst SD card, table *Prescription*) is 15 s.

However, these numbers must be put in perspective with the cost incurred by future stratifications. Thus, we compare the write I/Os induced by the insertion of a single tuple during the complete lifecycle of the database (i.e., through all stratification steps) with the same operation over a FTL without stratification.

Considering the final size of the database with $SF = 0.5$ (3 M tuples in *Prescription*) and a scalability limit such that $\downarrow$DB = 300 K (i.e., 300 K tuples in *Prescription*), the total number of I/Os induced by a single tuple is between 4 and 5, depending on the table. This number grows between 8 and 12 with $\downarrow$DB = 50K (the smaller the scalability limit, the higher the number of stratifications). The reason for these surprisingly small numbers is twofold: (i) 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 Section 4.2.1); (ii) each stratification incurs the complete rewriting of the database, the part of this cost attributed to each tuple being proportional to its size (including indexes). In the synthetic medical database, this size is at maximum 300 bytes, leading to about 1/7 I/O per tuple for each stratification. In contrast, an insertion through a FTL would produce between 12 (*Doctor* table) and

31 (*Prescription* table with 29 index + 1 table + SKT) random I/Os, under the favorable hypothesis that each insertion in a B$^+$-Tree index generates a single I/O. Each of these I/O in turn generates p physical writes, where p is the write amplification factor of the considered FTL[18]. Thus, serialization and stratification 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.

5.1.2 Performance of serialized indexes

We now focus on the cost of exact match lookups and insertions for the serialized indexes proposed in Section 4.3.1. They are composed of a combination of different structures: *Key Areas* (KAs) replicate the indexed attributes; *Summarized KAs* (SKAs) are made up of Bloom Filters (one per KA page) to speed up exact match predicates; finally, Pointer Lists (PTRs) chain index entries that share the same value. Differently from 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 13.a reports the results for exact match index lookups and Figure 13.b 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). As expected, *Full Index Scan* has a very good insertion cost and a very bad lookup performance, because it performs a full KA scan whatever the selectivity.

[18] Typical values of p were around 4 in 2009 for SSDs [39] and are even higher on simpler flash devices like SD cards or flash drives given their reduced cache capabilities. For reference, recent (2012) high-end SSDs whose FTL features deduplication and compression have an estimated factor of 0.15 [50].

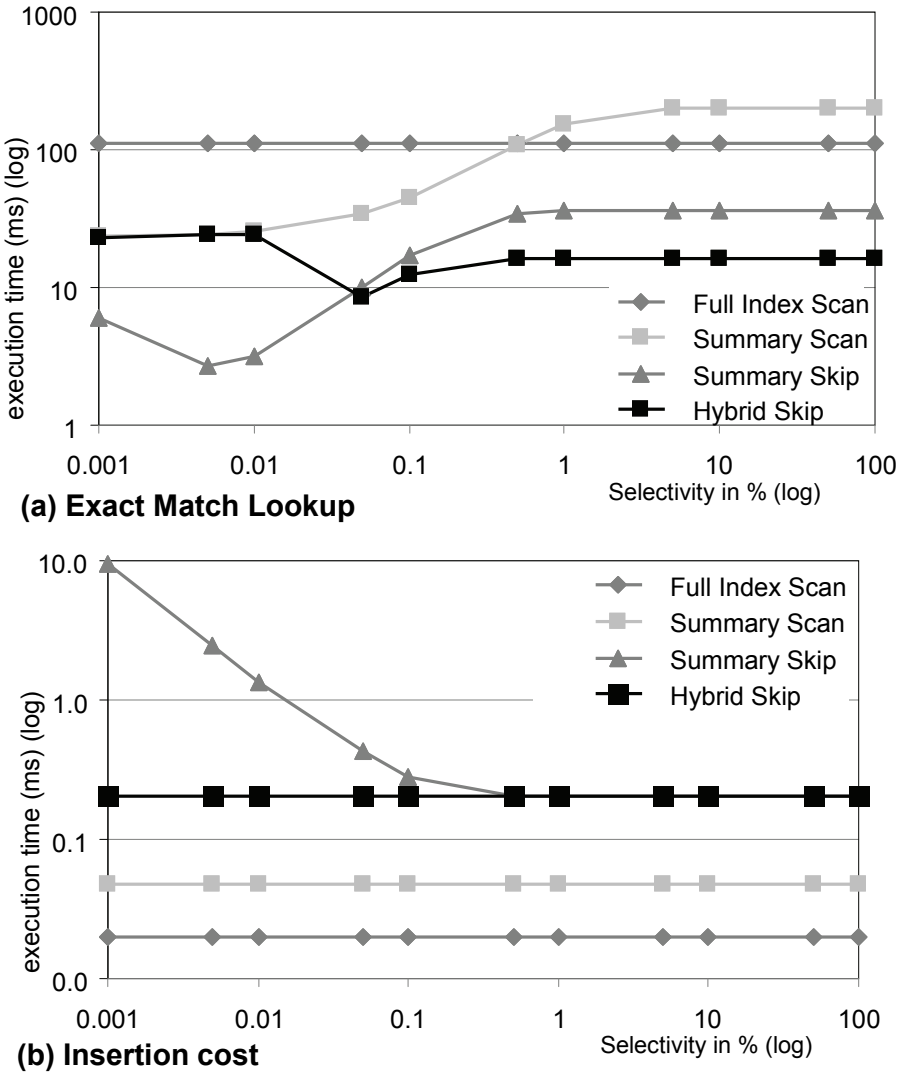

Figure 13: Performance of Serialized Indexes (simulation)

Summary Scan reaches pretty good insertion costs but lookups do not scale well with low selectivity predicates. Indeed, since almost all KA pages contain the searched value, each Bloom Filter of the SKA will have a match, leading to the scanning of all KA pages; thus, *Summary Scan* is equivalent to a SKA scan followed by a *Full Index Scan* for low selectivity predicates. This explains the extra cost compared to *Full Index Scan* with a selectivity lower than 1 %. Conversely, *Summary Skip* performs better in terms of lookup but induces very high chaining costs when the inserted value is infrequent. In order to maintain the chained list, the engine must perform a simplified *Summary Scan* (the scan is interrupted when the result is found) to get the previous occurrence of the

attribute. *Hybrid Skip* appears as the best compromise, with a bounded insertion cost, and very good lookup costs. With a selectivity higher than 0.1 %, the SKA n pages' scan limit is reached without finding a previous occurrence and the pointer chain is broken, which avoids the problematic behavior of *Summary Skip*. With low selectivity predicates, it even outperforms any other index, including *Summary Skip,* because pointers accessed in PTR are smaller in size. Indeed, the pointer size is only dependent on the number of keys summarized by n SKA pages rather than the cardinality of the whole SKA (*Summary Skip*). Thus, the PTR structure is more compact, which leads to less I/Os. Finally, we verified that *Hybrid Skip* scales up to rather large (e.g., 0.5 M) number of tuples (not shown on figures).

5.1.3 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 stratification cost.

We first focus on the query cost and run a set of 18 queries: 12 queries, termed $Mono_i$, involve an exact match selection predicate on a single table on attribute ID, DUP10 or DUP100, join this table up to *Prescription* and project 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 ClassOfDrug.DUP100. Table 3 presents each query used in Section 4.3.2 (in the same order) with its name, its number of results (given that 3000 deletes have been performed before), and, for a selection of representative queries, the corresponding execution plan. We measured the query response time with 3 settings: (1) the database has just been reorganized and thus $\downarrow DB = \emptyset$; (2) $\downarrow DB = 50\ K$; (3) $\downarrow DB = 300\ K$. Figure 14 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).

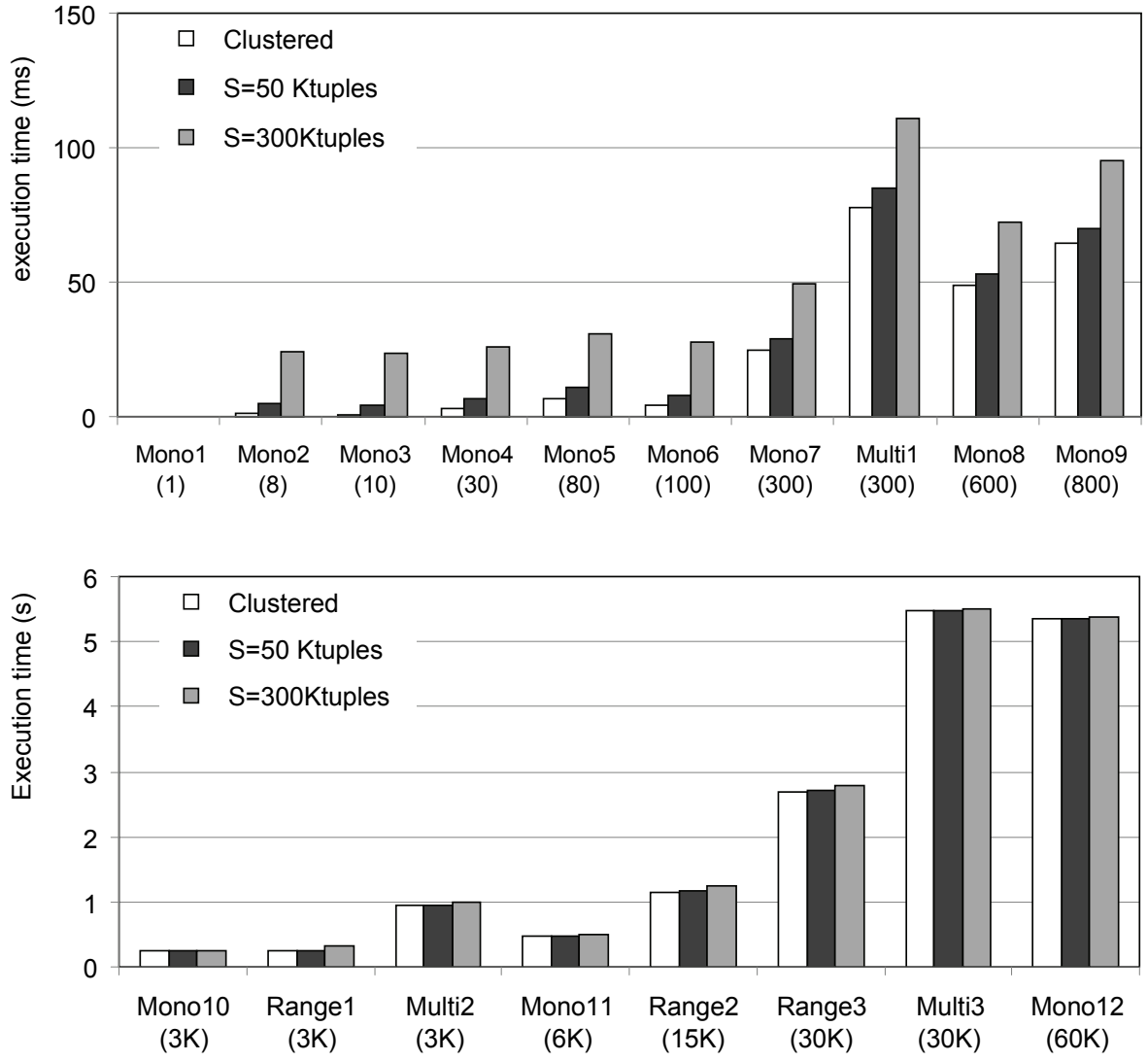

Figure 14: Performance of 18 queries with different settings

For selective queries (1-799 results), selection cost is relatively important with large ↓DB (300 K) while with $\downarrow DB = 50\ K$ the response time is very near the stratified one. Considering several predicates (Multi1) increases this cost, as expected. Finally, the cost of Mono1 is almost zero because it retrieves a prescription having a specific ID, which is, in our setting, the tuple physical address.

For less selective queries (3-60 K results), the cost is dominated by the projection step. Consequently, the ↓DB size has little influence.

Regarding updates and deletes, measurements on $\downarrow DB = 50\ K$ and $\downarrow DB = 300\ K$ have been done after having deleted randomly 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 ↓UPD (see Section 4.4.3).

Table 3: Queries and execution plans

Name	Results	Expression
Mono1	1	SELECT Pre.A1 FROM Pre WHERE ID = 1500000;
Mono2	8	SELECT Pre.A1, Dru.A1 FROM Pre, Dru WHERE Pre.idDru=Dru.ID AND Dru.ID = 20000; --- Project (L_2, <Pre.A1, Dru.A1>) SJoin (L_1, SKT_{Pre}, <idPre, idDru >) → L_2 CLIL ($CLI_{Dru.ID}$, Dru.ID=20000, Pre) → L_1
Mono3	10	SELECT Pre.A1 FROM Pre WHERE Pre.DUP10 = 'VAL_150000';
Mono4	30	SELECT Pre.A1, Dru.A1, Cla.A1 FROM Pre, Dru, Cla WHERE Pre.idDru = Dru.ID AND Dru.idCla = Cla.ID AND Cla.ID = 5000;
Mono5	80	SELECT Pre.A1, Dru.A1 FROM Pre, Dru WHERE Pre.idDru = Dru.ID AND Dru.DUP10 = 'VAL_2000';
Mono6	100	SELECT Pre.A1 FROM Prescription Pre WHERE Pre.Dup100 = 'VAL_15000';
Mono7	300	SELECT Pre.A1, Dru.A1, Cla.A1 FROM Pre, Dru, Cla WHERE Pre.idDru = Dru.ID AND Dru.idCla = Cla.ID AND Cla.DUP10 = 'VAL_500';
Multi1	300	SELECT Pre.A1, Vis.A1, Doc.A1, Dru.A1, Lab.A1, Cla.A1 FROM Pre, Vis, Doc, Dru, Lab, Cla WHERE Pre.idVis = Vis.ID AND Vis.idDoc = Doc.ID AND Pre.idDru = Dru.ID AND Dru.idLab = Lab.ID AND Dru.idClaID = Cla.ID AND Doc.MS1 = 'VAL_50' AND Cla.MS1 = 'VAL_50'; --- Project (L_4, <Pre.A1, Vis.A1, Doc.A1, Dru.A1, Lab.A1, Cla.A1>) SJoin (L_3, SKT_{Pre}, <idPre, idDoc, idVis, idDru, idLab, idCla>) → L_4 Merge ($L_1 \cap L_2$}) → L_3 CLIL ($CLI_{Doc.MS1}$, Doc.MS1='VAL_50', Pre) → L_2 CLIL ($CLI_{Cla.MS1}$, Cla.MS1='VAL_50', Pre) → L_1

Mono8	599	SELECT Pre.A1, Dru.A1, Lab.A1 FROM Pre, Dru, Lab WHERE Pre.idDru = Dru.ID AND Dru.idLab = Lab.ID AND Lab.ID = 2500;
Mono9	799	SELECT Pre.A1, Dru.A1 FROM Pre, Dru WHERE Pre.idDru = Dru.ID AND Dru.DUP100 = 'VAL_200';
Mono10	2997	SELECT Pre.A1, Dru.A1, Cla.A1 FROM Pre, Dru, Cla WHERE Pre.idDru = Dru.ID AND Dru.idCla = Cla.ID AND Cla.DUP100='VAL_50';
Range1	2997	SELECT Pre.A1, Dru.A1, Cla.A1 FROM Pre, Dru, Cla WHERE Pre.idDru = Dru.ID AND Dru.idCla = Cla.ID AND Cla.DUP100 > 'VAL_99'; --- Project (L_2, <Pre.A1, Dru.A1, Cla.A1>) SJoin (L_1, SKT_{Pre}, <idPre, idDru, idCla>) → L_2 CLIL ($CLI_{Cla.DUP100}$, Cla.DUP100>'VAL_99', Pre) → L_1
Multi2	2996	SELECT Pre.A1, Vis.A1, Doc.A1, Dru.A1, Lab.A1, Cla.A1 FROM Pre, Vis, Doc, Dru, Lab, Cla WHERE Pre.idVis = Vis.ID AND Vis.idDoc = Doc.ID AND Pre.idDru = Dru.ID AND Dru.idLab = Lab.ID AND Dru.idCla = Cla.ID AND Doc.MS10 = 'VAL_5' AND Cla.MS10 = 'VAL_5' AND Lab.MS10 = 'VAL_5';
Mono11	5995	SELECT Pre.A1, Dru.A1, Lab.A1 FROM Pre, Dru, Lab WHERE Pre.idDru = Dru.ID AND Dru.idLab = Lab.ID AND Lab.DUP10 = 'VAL_250';
Range2	14955	SELECT Pre.A1, Dru.A1, Cla.A1 FROM Pre, Dru, Cla WHERE Pre.idDru = Dru.ID AND Dru.idCla = Cla.ID AND Cla.DUP100 > 'VAL_95';
Range3	29912	SELECT Pre.A1, Dru.A1, Cla.A1 FROM Pre, Dru, Cla WHERE Pre.idDru = Dru.ID AND Dru.idCla = Cla.ID AND Cla.DUP100 > 'VAL_90';
Multi3	29913	SELECT Pre.A1, Vis.A1, Doc.A1, Dru.A1, Lab.A1, Cla.A1 FROM Pre, Vis, Doc, Dru, Lab, Cla WHERE Pre.idVis = Vis.ID AND Vis.idDoc = Doc.ID AND Pre.idDru = Dru.ID AND Dru.idLab = Lab.ID AND Dru.idCla = Cla.ID AND Cla.MS10 = 'VAL_5' AND Lab.MS10 = 'VAL_5';
Mono12	59881	SELECT Pre.A1, Dru.A1 Lab.A1 FROM Pre, Dru, Lab WHERE Pre.idDru = Dru.ID AND Dru.idLab = Lab.ID AND Lab.DUP100 = 'VAL_25';

Let us now consider the stratification cost. We consider a fixed size for *DB (2.7 M tuples in *Prescription*) and vary the size of ↓DB (varying the *Prescription* table from 50 K to 300 K tuples, other tables growing accordingly). The reorganization cost (see Section 4.4.4 for the reorganization algorithm) varies linearly from 7 min (for 50 K) to

9.6 min (for 300 K). It is worth noting that stratification runs in background, and does not block queries. The stratification 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 the stratification, varies between 1.5 GB (for 50 K) and 2,1 GB (for 300 K). However, 60 stratification steps are required to reach 3 M tuples in Prescription with a ↓DB of 50 K, while only 10 stratification steps are necessary with a ↓DB of 300 K. In any case, the flash lifetime does not appear to be a problem.

As a final remark, the FTL approach, which was already disqualified due to its write behavior, does not appear to be 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 serialization/stratification, the FTL overhead incurred by traversing translation tables makes the performance of less selective (then costly) queries much worse than serialization/stratification. Indeed, the high number of I/Os generated at projection time dominates the cost of these queries.

5.2 Hardware prototype

In a second phase, we developed a partial implementation of our database engine on a real hardware platform, i.e. on a secure USB device (Figure 15) provided by Gemalto, our industrial partner. This platform is equipped with a smart card like secure MCU, powered by a 20 MHz 32-bits RISC CPU and containing 64 KB of RAM (with half of it reserved to the operating system). Its secure storage is composed of 1 MB of NOR Flash memory and its external storage is a NAND Flash memory module, connected by a Serial Peripheral Interface (SPI) bus (for our experiments the token used a 128 MB Samsung K9F1G08X0A module, but in the near future it should use a Gigabyte-sized one). Unlike most commercial products, it offers a direct access to

Flash memory, i.e., the FTL can be bypassed. This Flash chip contains 512 B sectors, each page contains 4 sectors, and each block is formed by 64 pages.

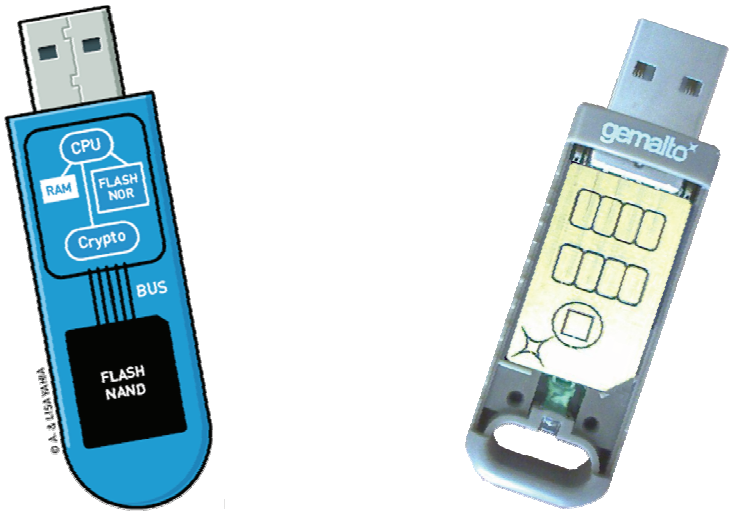

Figure 15: Logical and physical layouts of the secure USB device

Due to the limited storage space available on our test platform, we focused on evaluating the performance of our serialized indexes. As such, elements from the clustered database layout were not considered, as we estimated in Section 5.1.3 that the stratification process would require at least one free gigabyte of storage. Our implementation comprised the sequential database storage model, the different operators required to implement the three main index variants (*Full Index Scan*, *Summary Scan* and *Hybrid Skip* only, as the simulation results suggested that *Summary Skip* brings no advantage over *Hybrid Skip*), the support for delete and update operations and the I/O glue to handle direct Flash accesses.

It was not possible, however, to reproduce the simulation with the exact same parameters (Section 5.1.2). Instead of 300 K records, the maximum number of tuples that could be stored in ↓DB was 90 K, with 10 attributes, populated with a varying number of distinct values uniformly distributed. The execution time was measured by calculating the number of CPU ticks elapsed while inside the index operators

(additionally, I/Os were performed synchronously, no multithreading involved). The selection results are summarized in Figure 16.

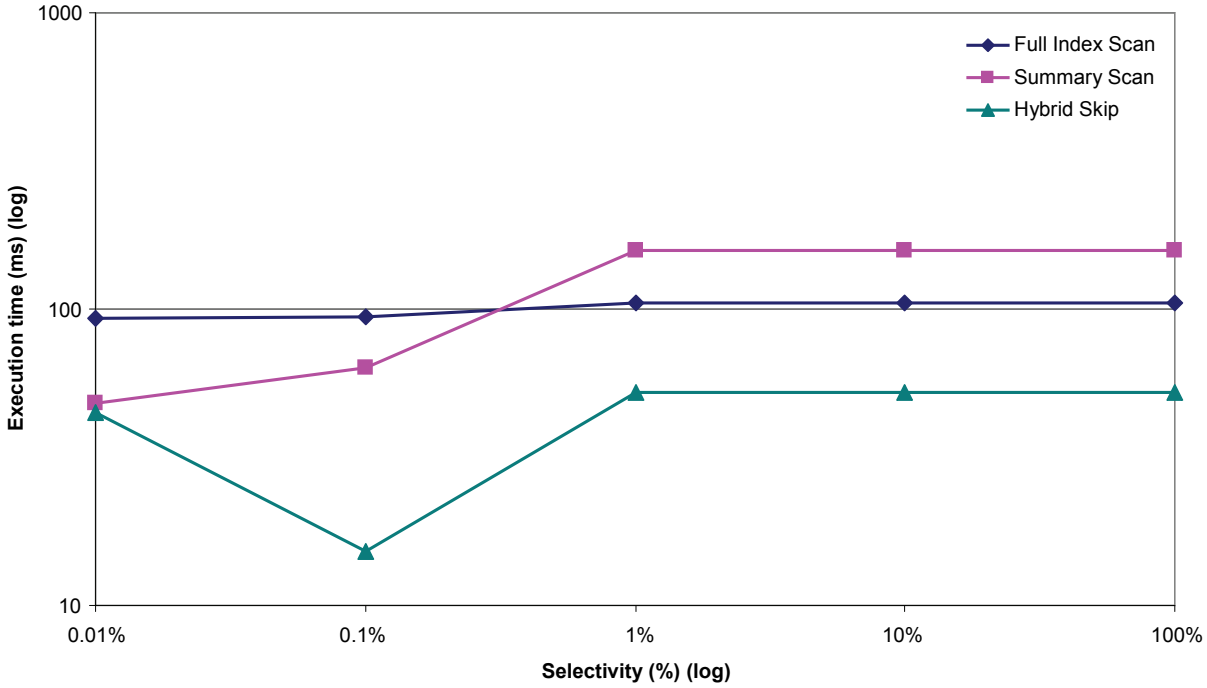

Figure 16: Selection performance of serialized indexes (secure USB device)

A comparison with the simulation results shows that the indexes perform globally in accordance with the behavior predicted. *Summary Scan* performs well for high selectivity predicates, but do not scale well when the selectivity decreases, because of the extra cost incurred to scan the whole KA and SKA; it is already outperformed by a greedy *Full Index Scan* when the selectivity reaches 1%. Again, we verified that *Hybrid Skip* appears as the best compromise, due to the small size of PTR. It is worth noting that the database size was about three times smaller in these experiments (90 K instead of 300 K), although the execution times were mostly similar, either for selections or insertions. This implies that hardware-specific problems[19] we estimated

[19] Cf. note 17 p. 82. Due to clock constraints for bus synchronization between the MCU and the NAND Flash chip, the CPU had to be down-clocked from 50 to 20 MHz.

during the calibration of the simulator to degrade the engine performance by a factor of x1.4 have a greater impact, because the synchronization issues and the consequently slower CPU clock increased the execution time by a factor of x3-4 instead. Although we are not able to confirm with a fully working platform, we hypothesize that the synchronization issues lead to use the SPI interface at a lower frequency (CLK signal at 10 MHz instead of 25 MHz by default). This basically means that SPI commands can be sent twice less often, and the time spent by the controller waiting for the acknowledgment after each byte transfer (no Direct Memory Access support) is also increased. As I/Os are synchronous, the SPI bus on this prototype is a bottleneck with respect to the Flash performance, at least by a factor of x2.

5.3 Storage layout

Many embedded platforms do not provide a direct interface to raw Flash chips anymore. For example, embedded boards available to system developers often feature a MicroSD connector for Flash memory cards only. Similarly to SSDs, Secure Digital (SD) cards use an abstraction layer to hide the peculiarities of Flash memory; however, due to their different form factor and much lower price, one can assume that their FTLs have less available "resources" (less powerful controller and reduced onboard RAM space because of the smaller form factor). Our serialized database was designed to work around problematic I/O patterns of raw Flash memory, which may already be hidden behind a basic SD card's FTL. In this section, we evaluate the impact of changing the storage interface on the database layout.

The second test platform selected was an ARM-based development board (STM32F217ZGT6[20]), whose System on Chip design was similar to the secure USB

[20] Datasheet available at `http://www.st.com/internet/mcu/product/250172.jsp` (retrieved on 2012-06-15).

device we previously evaluated. The STM32F217 series of MCUs feature an ARM Cortex M3 CPU (120 MHz), 128 KB of RAM, 1 MB of NOR, and a crypto co-processor. Although it uses a different form factor (10 x 10 cm board), it shares the hardware commonalities of SPTs and can be considered as such. However, this MCU does not provide direct NAND access, but a SPI bus with a MicroSD port, requiring the use of MicroSD cards as external storage devices. The exact physical layout of Flash-based SD cards is unknown: it might be identical to the secure USB device's chip, but the page size varies with the card capacity (large block Flash chips feature 4 KB pages and 512 KB blocks [21]). Therefore, any existing assumption of the Flash organization beyond the logical view the FTL provides is not safe to apply to several SD cards and should be discarded.

Figure 17: STM32F217 test platform

Preliminary performance tests with the MicroSD card on the platform produced the following results: a read operation takes 1 ms, and a write operation less than 10 ms. As such, I/Os on this platform are about x25 times slower than our calibrated

[21] For instance, the following white paper related to bad block management, available at `http://www.nxp.com/documents/application_note/AN10860.pdf` (retrieved on 2012-06-15), considers three different block sizes.

simulator, and x10 times slower than the secure USB device experimented previously. These results were corroborated by the fact that the complete experiment used to evaluate the selection performance of the serialized indexes took more than 24 hours to complete (instead of 2 hours on the secure USB device). To determine whether SD cards in general exhibit low performances, whether our implementation was sub-optimized, or whether the serialized design is badly adapted to these storage devices, we evaluated the performance of several SD cards in isolation by reproducing the serialized data access pattern with the uFLIP benchmark tool.

UFLIP [20] is a benchmark designed to measure the response time of Flash-based devices to different I/O patterns (distributions of reads and writes over time and device space). Although it targets complex Flash devices such as SSDs, we considered a lightweight version of its test plan and only selected a few tests to reproduce the behaviour of our serialized layout. First, uFLIP performs by default a *random format* of the device, covering its entire surface with random writes of varying sizes, so as to place all devices to be tested in a "neutral" state[22].

However, an SD card used in the context of our database engine would only receive sequential writes. Therefore, we assumed that a "neutral" state in our context would be when a card has already been filled completely with data from SWSs, which we simulated by an initial *sequential format* operation. Then a classic uFLIP run measures the impact of several I/O patterns, such as: granularity, alignment, locality, circularity, partitioning, ordering and parallelism – most of them being excluded by our serialized design. An insert implies appending data sequentially in SWSs, for example writing a tuple in ↓DATA and ↓IND for all its associated indexes' structures (e.g. for Summary Skip: KA, SKA and PTR). If we consider several SWSs, then a typical serialized

[22] All SSDs are not sold in identical states: for instance, they can be preformatted for Windows, or may have already experienced several write-erase cycles as part of the manufacturer's quality assurance process.

workload can be simulated by the partitioning pattern of uFLIP, where the device space is divided into partitions, accessed sequentially in a round-robin fashion (similarly to, for instance, a merge operation of several buckets during external sort). As illustrated in Figure 18, if we assume that W_{ij} represents the j^{th} write operation to the i^{th} partition, then m sequential writes to a single partition are represented with the sequence $\{W_{11}, W_{12}, W_{13},\ldots, W_{1m}\}$. Thus, with m sequential writes to n partitions, uFLIP will submit the following pattern: $\{W_{11}, W_{21}, \ldots, W_{n1}, W_{12}, W_{22}, \ldots, W_{n2}, \ldots, W_{1m}, W_{2m}, \ldots, W_{nm}\}$.

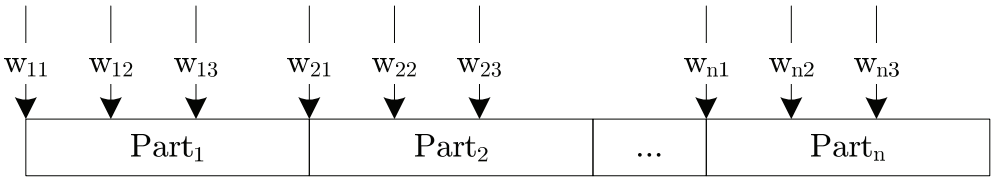

Figure 18: uFLIP's partitioning I/O pattern

In our context, the first partition can represent a table and the next partitions its associated indexes. Since each tuple insert is followed by an index insert, the round-robin partitioning pattern of uFLIP actually simulates the write operations that occur into SWSs on insert queries. As such, this benchmark allows us to verify whether a pseudo-sequential pattern that provides satisfying performances on raw Flash chips is well supported on SD cards. Nonetheless, if the cards exhibit performance issues, we hope this experiment will enable us to isolate reproducible behaviors among the tested cards, in order to refine the design rules with their peculiarities.

We ran our benchmark on a set of 10 SD cards, of various sizes (from 1 to 32 GB), manufacturers and form factors (3 of them are MicroSD cards), both for sequential partitioned reads and writes, from 1 up to 256 partitions. Basic read and write costs are included for reference in Table 4; random writes were performed after the sequential and read-only experiment to avoid disturbing the "neutral state" of each card.

Table 4: SD cards performance results for basic patterns

	Reads (512 B, in ms)		**Writes (512 B, in ms)**	
	sequential	random	sequential	random
KINSTON 4GB	0.9	2.1	5.9	300
SILICON POWER 4GB	0.7	0.8	1.8	100
SILICON POWER 4GB	0.7	1.3	14	600
SONY 4GB	1.2	1.5	3	700
KINSTON 8GB	0.7	1.1	9	400
SAMSUNG 8GB	0.9	1.2	2.4	400
SAMSUNG 32GB	0.7	0.9	1	40
KINGSTON 4GB (MicroSD)	1.1	1.4	6	500
SAMSUNG 8GB (MicroSD)	1	1.1	4	400
Noname 1GB (MicroSD)	0.6	0.8	3	100

As it could be expected by the absence of mechanical parts in NAND Flash, the access pattern (sequential, random) has little to no influence on read performance, which is only dependent on the data size; although the factor varies between cards. Also, these results confirm the better efficiency of a storage design avoiding random writes, because they are several orders of magnitude slower than sequential ones. Unfortunately, I/Os are indeed much more costly with SD cards than with a direct NAND Flash access, be it the poor quality of Flash chips used inside SD cards, the included FTL, or overall that heavy I/Os are not the intended market usage for them; for instance, a single sequential write is x3-35 times faster on raw NAND Flash, depending on the SD card considered.

As we can observe on Figure 19 for one MicroSD card, the duration of read operations is not, once more, affected by the number of partitions read from; whichever card was tested, the response time remained mostly constant. This suggests that successive and sequential reads to different structures, e.g. SKA then KA then PTR do not degrade significantly the performances of the database engine.

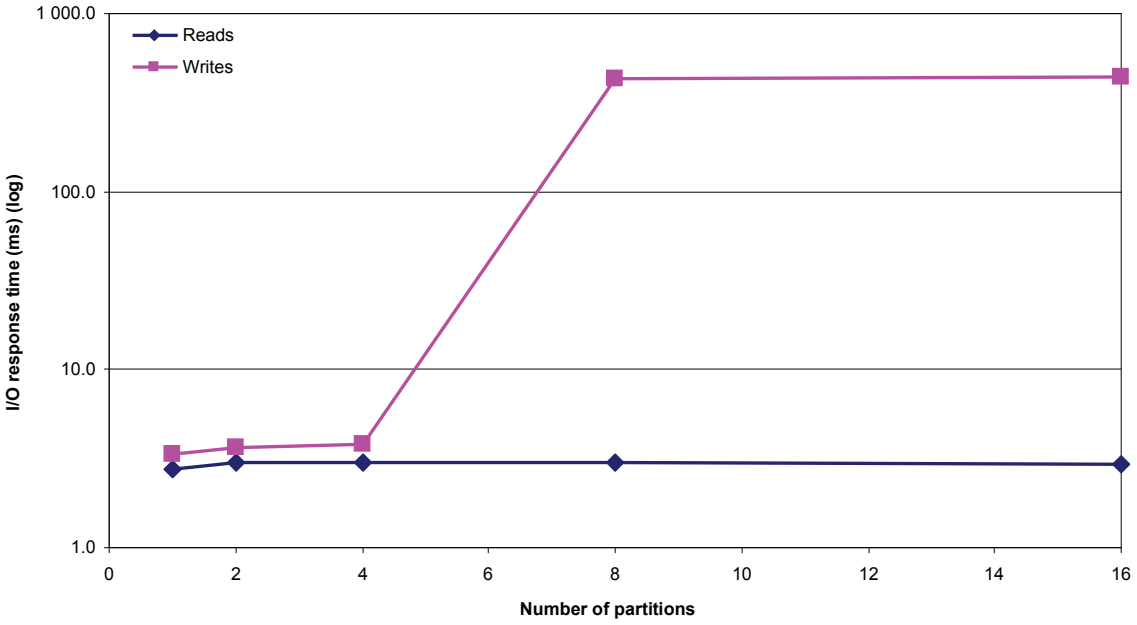

Figure 19: Response time of partitioned sequential I/Os (Samsung microSD 8GB)

Partitioned sequential writes appear, however, to be more problematic to handle by all tested cards. A common pattern is illustrated on Figure 19: partitioned sequential writes perform as quickly as sequential writes until a certain threshold (number of partitions) is reached; beyond this point their performances degrade to random writes (several orders of magnitude slower depending on the card). The actual FTL algorithms implemented on our tested samples being unknown, we can only hypothesize that the request allocator fails to "detect" that writes are sequential within each partition, and therefore each subsequent write triggers a remapping and possibly a garbage collection of the affected blocks, which would explain the sudden performance degradation, and why raw Flash chips are not affected. If we consider the design described previously (with one SWS for the table and a second SWS for its associated indexes), these results suggest that our serialized design becomes inefficient on SD cards for database schemas containing more than two tables.

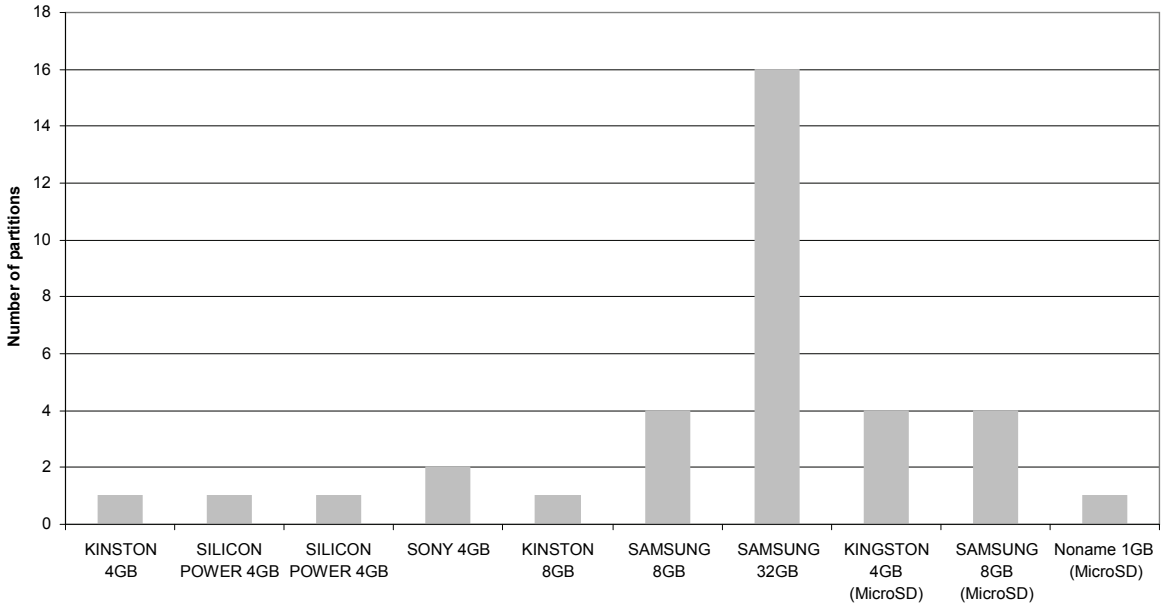

Figure 20: Support of partitioned writes for 10 SD cards

The thresholds for all tested cards are summarized in Figure 20, which shows an important heterogeneity of the results, similarly to what was observed by uFLIP authors on SSDs. Neither the form factor nor the capacity has a significant impact on the device's performance (with the exception of the 32 GB card that performs exceptionally well); worse, cards from a single manufacturer do not exhibit the same limit. This behavior makes it difficult to build a "catch-all" solution for FTL-over-Flash devices, by opposition to raw NAND Flash chips, although it is still possible to check whether a given card will perform best in our SWS context (for example, these results would suggest to use SAMSUNG cards only with the serialized database).

Furthermore, half of the cards tested do not support any partitioned write at all without performance degradation. This implies that our design is not adapted to SD cards, because its efficiency with a random card bought from a classic retailer would be unknown, or at best tied to the database schema the user intends to deploy. However, the design can be made compatible with minor adjustments to more cards by reducing the number of concurrent SWSs. For instance, a single SWS for the current serialized

database instead of one per table can be achieved by adding structures to track the position of tuples and sharing buffers among them. Then, it is possible to use at most 4 SWSs for the whole database: $\downarrow DB_{i+1}$, $*DB_{i+1}$, $\downarrow DB_i$, and $*DB_i$. $*DB_i$ is the reorganized database resulting from the previous stratification, and $\downarrow DB_i$ is the serialized database that reached its scalability limit; as such, both of them are invariant, and accessed in read-only mode. Conversely, $\downarrow DB_{i+1}$ is the current serialized database where insertions occur, and $*DB_{i+1}$ is the new reorganized database which is being produced by merging $\downarrow DB_i$, and $*DB_i$. Therefore, at any time, there are 4 SWS read (partitioned reads are well supported), but only 2 SWSs written concurrently (hence 2 RW partitions), which fits most SD cards. However, low-end cards with a single write partition remain problematic to handle efficiently with respect to the stratification process without a loss of functionality. For example, the stratification can still be performed in a degraded mode that cannot be interrupted, during which normal database interactions are suspended (no concurrent insertions to $\downarrow DB_i$, hence only one SWS is written at any time).

5.4 Conclusion

In this chapter, we first built a simulator for our platform using analytical cost models and calibrations from an existing similar prototype. The performance results showed that the purely sequential database design was fully adapted to the hardware constraints of the Secure Portable Token context. The behavior of the engine was further confirmed with an experiment on real hardware of our reference implementation, whose results suggested that the performances of the storage system, rather than the CPU clock speed, are the main bottleneck on SPT platforms. Finally, our experiment to evaluate the database performance on other form factors underlined the importance of avoiding unsupported I/O patterns and isolated new design issues that were not present with raw NAND Flash.

Chapter 6
Conclusion and future work

The number of information systems gathering personal data on servers is escalating at a tremendous pace. Citizens have no way to opt-out because governments or companies that regulate our daily life require them. Administrations and companies deliver an increasing amount of personal data in electronic form, which often ends up as well in servers at the user convenience. Indeed, the user expects her data to be available, resilient to failure and easily manageable – a service that many internet companies provide. All these situations result in the accumulation of one's complete digital history in servers. Although data centralization has unquestionable benefits in terms of resiliency, availability and even consistency of security policies, they must be weighted carefully against the privacy risks.

The approach promoted in this thesis is part of the global *Personal Data Server* vision. As described in [9], it outlines an individual-centric architecture whose aim is to enable powerful personal data applications and at the same time provide user-friendly control over one's data with tangible enforcement guarantees. This vision is based on a *Secure Portable Token* that embeds a software suite, allowing it to provide a full-fledged database engine, which acts as an interoperable data hub (e.g., acquire personal data from servers, share them with other personal data servers) while enforcing the Hippocratic privacy principles [5] (e.g., limited collection, limited retention, audit). These features must take into consideration the token security constraints and not

disrupt its portability. Although its scope is not limited to the above vision, the embedded data server engine proposed in this thesis fits in the personal data server architecture and shows that managing a gigabyte-sized database within a secure token is not pure utopia.

This chapter concludes the thesis. We synthesize the work conducted, and close the manuscript by opening exciting research perspectives.

6.1 Synthesis

Be it on size, cost, energy consumption or shock resistance, embedded devices are subjected to many restrictions. Due to their use of a secure microcontroller that must comply with security properties, Secure Portable Tokens are even more constrained. These limitations force to adopt certain hardware technologies and designs, such as reducing the system capabilities with less RAM and processing power, or using storage systems without any mechanical moving parts. NAND Flash has become the most popular stable storage medium for embedded systems. Therefore, efficient storage models and indexing techniques are needed to cope with the ever-growing storage capacity. Designing these methods is complex because embedded systems combine their own design limitations with the peculiarities of NAND Flash, whereas state of the art work focuses on addressing each of them independently. In this thesis, we addressed specifically the combination of both aspects.

We analyzed hardware peculiarities of Flash-based embedded devices and identified a comprehensive set of key constraints applying, in our context, to Secure Portable Tokens. We found that existing Flash-based storage models and batch indexing methods are inadequate to answer all requirements at once. Then we proposed a new approach based on serialization and stratification to adapt naturally the defined constraints.

In the serialized data storage model, all structures of the database (data, indexes, buffers and update and delete logs) are organized by Sequentially Written Structures (SWSs). In order to solve the scalability issues inherent to the sequential layout, we introduced the concept of database stratification. We illustrated both concepts by applying them to the whole database, and built a cost model simulator to evaluate the performance of each component and of the overall system. Experiments of our prototype on several embedded devices have showed the predictability and adaptability of the proposed solutions. Considering that our embedded engine is able to manage a complete database without generating any random write, it seems to be applicable to a wider context, as long as random writes are detrimental in terms of I/O cost, energy consumption, space occupancy or memory lifetime.

6.2 Perspectives

The work conducted in this thesis can be pursued in various directions. We identify below some challenging issues and outline possible lines of thought to tackle them.

Tolerance limits of NAND Flash memory

Current Flash trends [1] predict an increase of the bits stored per Flash cell, compacter dies (their size already shrinks faster than Moore's law) and a decrease of the block endurance[23]. This means that next-generation SD cards (which often use low quality and multi-level cells) are unlikely to change the poor performance observed in Chapter 5 with respect to partitioned sequential writes (only one partition is well supported, writes to two partitions degrade to random writes). Although our design can be adapted to generate less SWSs, the stratification process remains suboptimal on these SD cards, because it requires two SWSs to proceed (↓DB for new insertions and

[23] New generation TLC (Triple-Level Cells) Flash supports less than 5000 erase cycles per block.

*DB for the stratum being reorganized). A merge of these two distinct SWSs into one can be achieved by maintaining tracking structures to identify whether a tuple comes from ↓DB or *DB and pointer lists to link elements from e.g. the same table in order to speed up lookups. Although this increases the number of writes performed on insertions and reads on selections, the overall performance loss is minor because random writes still remain two or three orders of magnitude more costly. Note that maintaining tracking structures will increase RAM and Flash occupancy, and a single SWS for both ↓DB and *DB implies that their lifetimes are tied, because the SWS principle forbids any partial reclamation (whereas previously a small ↓DB could be reclaimed as early as possible).

Document retrieval in Personal Data Servers

In the PDS vision, the user is supposed to be able to store permanently files or documents on the Secure Tokens or on the Supporting Servers. She may need to access them later, without remembering their file names or in which directory they were saved. Several approaches can be envisaged to tackle this use case, such as full text indexing. However, maintaining these indexes with the embedded constraints seems to be challenging, additionally to being very space consuming. An alternate solution is to rely on the file system implementation to support more file metadata. For example, on Linux, the Unsorted Block Image File System (UBIFS), targeted at Flash memory, is able to store 4 KB user metadata (*user_xattr* mount option) alongside the file node. A further study could assess the relevancy of these approaches, how to enable efficient file and tag search with our low RAM constraint, and finally, how to handle file moves, deletions or tags updates with the Flash constraints.

Emerging memory and storage technologies

As written above, Flash memory evolves towards more density and more capacity but less endurance [1]. These trends imply that more and more error correction

mechanisms (ECC) will be required to ensure correct data retention. Several new non-volatile memories, such as SONOS (Silicon-Oxide-Nitride-Oxide-Silicon), aim at providing higher quality replacements by using more efficient components to store data in their cells. Conversely, other technologies are emerging, that do not inherit Flash constraints (see Chapter 2), such as MEMRISTOR, Magnetoresistive random-access memory (MRAM) or Ferroelectric RAM (FeRAM).

Phase Change Memory (PCM), whose high-volume production has been announced in July 2012, seems to be the technology preferred by the industry to replace Flash in embedded devices [57]. Unlike Flash memory, PCM is directly bit-alterable, which means that no block erase operation is required; furthermore, it does not suffer from the extra addressing cost on the first block access, enabling performance greater than Flash by several orders of magnitude. As such, in many aspects, it can be compared to volatile memory technologies such as DRAM, although PCM still shows an asymmetry between reads and writes, in terms of speed and energy consumption. Several usages of PCM are being envisioned with respect to main memory organization (replace DRAM with PCM, include PCM alongside DRAM, or demote DRAM as a simple cache for PCM), and database researchers are already suggesting PCM-friendly algorithms [25].

In the embedded context, PCM may be a good candidate for two usages. First, it may replace NAND Flash, due to the lack of moving parts and the great shock resistance. Although implementing PCM as the main storage device is unlikely for now (it offers less capacity than Flash as of today), its use would be interesting to free embedded devices from the burden of Flash constraints. Second, it may replace DRAM as the main memory unit thanks to their similarities. Its larger density over DRAM makes it particularly relevant for devices where the die size is a deciding factor, for instance Secure Portable Tokens. Furthermore, as PCM is a non-volatile memory (not cleared on restart), its use provides promising insights in term of transaction management (atomicity and durability) and recovery (automatic post-crash resuming of running

queries). However, these aspects should be weighted with its higher power consumption during write operations, especially on autonomous devices (e.g. sensors). A further study could assess the need for both PCM-friendly and energy-efficient database algorithms.

Bibliography

[1] Abraham, M. (Micron Technology, Inc), "NAND Flash trends for SSD/Enterprise", http://ftp.bswd.com/FMS10/FMS10-Abraham.pdf, Flash Memory Summit, 2010, retrieved on 2012-06-15.

[2] Adam, N. R. and Worthmann, J. C. Security-control methods for statistical databases: a comparative study. ACM Comput. Surv., 1989.

[3] Agrawal D., Abbadi A. E., Wang S., "Secure Data Management in the Cloud", DNIS, 2011.

[4] Agrawal, D., Ganesan, D., Sitaraman R., Diao Y. and Singh S. Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices. VLDB, 2009.

[5] Agrawal, R., Kiernan, J., Srikant, R. and Xu, Y. Hippocratic Databases. VLDB, 2002.

[6] Ailamaki, A., DeWitt, D.J. and Hill, M. D. Data page layouts for relational databases on deep memory hierarchies. The VLDB Journal, 2002.

[7] Allard, T., Nguyen, B. and Pucheral, P. Safe Anonymization of Data Hosted in Smart Tokens, PRiSM Technical Report n° 526, 2010.

[8] Anciaux, N., Benzine, M., Bouganim, L., Pucheral, P. and Shasha, D. GhostDB: Querying Visible and Hidden data without leaks. ACM SIGMOD, 2007.

[9] 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", PVLDB, 2010.

[10] Anciaux, N., Bouganim, L., Guo, Y., Pucheral, P., Vandewalle J-J. and Yin, S. Pluggable Personal Data Servers. ACM SIGMOD, 2010.

[11] Arge L., "The Buffer Tree: A Technique for Designing Batched External Data Structures", Algorithmica, 2003.

[12] Ban, A.: Flash file system. United States Patent, no. 5,404,485 (1995).

[13] Ban, A.: Flash file system optimized for page-mode flash technologies. United States Patent, no. 5,937,425 (1999).

[14] Bernstein P., Reid C., Das S., "Hyder - A Transactional Record Manager for Shared Flash", CIDR, 2011.

[15] Birrell, A., Isard, M., Thacker, C., and Wobber T.: A design for high-performance flash disks. Operating Systems Review 41(2): 88-93 (2007).

[16] Bityutskiy A. B., "JFFS3 Design Issues", Tech. report, 2005.

[17] Bloom, B. H. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 1970.

[18] Bolchini C., Salice F., Schreiber F., Tanca L., "Logical and Physical Design Issues for Smart Card Databases", TOIS, 2003.

[19] Bonnet, P., Bouganim, L.: Flash Device Support for Database Management. CIDR 2011: 1-8.

[20] Bouganim, L., Jónsson, B. Þ. and Bonnet P. uFLIP: Understanding Flash IO Patterns. CIDR, 2009.

[21] Bum-soo Kim, G.y.L.: Method of driving remapping in flash memory and flash memory architecture suitable therefor. United States Patent, no. 6,381,176 (2002).

[22] Bursky D., “Secure Microcontrollers Keep Data Safe”, PRN Engineering Services, http://tinyurl.com/secureMCU, 2012, retrieved on 2012-06-15.

[23] Chan C. Y., Ioannidis Y. E., “An Efficient Bitmap Encoding Scheme for Selection Queries”, SIGMOD, 1999.

[24] Chen, F., Koufaty, D., and Zhang, X., Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In SIGMETRICS, 2009.

[25] Chen, S., Gibbons, P. B., and Nath, S., “Rethinking Database Algorithms for Phase Change Memory”, CIDR, 2011.

[26] Comer, D., “The Ubiquitous B-Tree”, ACM Computing Surveys 11(2): 121-137(1979).

[27] Debnath B., Sengupta S., Li J., “SkimpyStash: RAM Space Skimpy Key-Value Store on Flash”, SIGMOD, 2011.

[28] Dingledine, R., N. Mathewson, and Syverson P. Tor: The Second-Generation Onion Router. USENIX, 2004.

[29] Do, J., Patel, J. M.: Join processing for flash SSDs: remembering past lessons. In Proceedings of the Fifth International Workshop on Data Management on New Hardware (DaMoN '09). ACM, New York, NY, USA, 1-8.

[30] Elbaz, R., Champagne, D., Lee, R. B., Torres, L., Sassatelli G. and Guillemin P. TEC-Tree: A Low-Cost, Parallelizable Tree for Efficient Defense Against Memory Replay Attacks. CHES, 2007.

[31] Emmit Solutions, “Microcontroller Market and Technology Analysis Report”, 2008.

[32] Eurosmart. Smart USB token. White paper, Eurosmart, 2008.

[33] Fung, B. C. M., Wang K., Chen R. and Yu P. S. Privacy-preserving data publishing: A survey on recent developments. ACM Computing Surveys, 2010. To appear.

[34] Gemmell J., Bell G., Lueder R., “MyLifeBits: a personal database for everything”, Commun. ACM 49(1), 2006.

[35] Goldschlag, D., M. Reed, and Syverson P. Onion Routing for Anonymous and Private Internet Connections. Communications of the ACM, 1999.

[36] Gupta, A., Kim, Y. and Urgaonkar B.: DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. ACM ASPLOS, Washington, 2009.

[37] Haas, L. M., Carey, M. J., Livny, M. and Shukla, A. Seeking the truth about ad hoc join costs. VLDB Journal, 1997.

[38] Hacıgümüş, H., Iyer, B., and Mehrotra, S. Providing Database as a Service. ICDE, 2002.

[39] Hu X., Eleftheriou E., Haas R., Iliadis I., Pletka R.: "Write amplification analysis in flash-based solid state drives", SYSTOR 2009.

[40] IBM Corporation, “IBM DB2 Everyplace Version 9 Release 1", 2009.

[41] IDC, “IDC Defines the Personal Portable Security Device Market”, `http://tinyurl.com/IDC-PPSD`, 2007, retrieved on 2012-06-15.

[42] J. A. Kreibich, “Using SQLite”, O'Reilly Media, 2011.

[43] Kang, J., Jo, H., Kim, J., Lee, J.: A superblock-based flash translation layer for NAND flash memory. In: Proceedings of the 6th ACM & IEEE International conference on Embedded software, ACM New York, NY, USA (2006) 161–170.

[44] Kawaguchi, A., Nishioka, S., Motoda, H.: A flash-memory based file system. In: TCON’95: Proceedings of the USENIX 1995 Technical Conference Proceedings on USENIX 1995 Technical Conference Proceedings, Berkeley, CA, USA, USENIX Association (1995) 13–13.

[45] Kim, J., Kim, J., Noh, S., Min, S., Cho, Y.: A space-efficient flash translation layer for compactflash systems. IEEE Transactions on Consumer Electronics 48(2) (2002) 366–375.

[46] Kim, J., Pyeon, H., Oh, H., Schuetz, R., Gillingham, P.: Low Stress Program and Single Wordline Erase Schemes for NAND Flash Memory. In: Non-Volatile Semiconductor Memory Workshop, 2007 22nd IEEE. (2007) 19–20.

[47] Kim, Y. R., Whang, K. Y., and Song, I. Y., Page-differential logging: an efficient and DBMS-independent approach for storing data into flash memory. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD '10). ACM, New York, NY, USA, 363-374.

[48] Koltsidas I., Viglas S. D., "Data management over flash memory", SIGMOD, 2011.

[49] Krueger, W., Rajagopalan, S.: Method and system for file system management using a flash-erasable, programmable, read-only memory. United States Patent, no. 6,256,642 (2001).

[50] Ku, A., "Intel SSD 520 Review: Taking Back The High-End With SandForce", `http://www.tomshardware.com/reviews/ssd-520-sandforce-review-benchmark,3124-11.html`, retrieved on 2012-06-15.

[51] Lee, S. and Moon, B. Design of flash-based DBMS: an in-page logging approach. ACM SIGMOD, 2007.

[52] Lee, S., Park, D., Chung, T., Lee, D., Park, S., Song, H.: A log buffer-based flash translation layer using fully-associative sector translation. ACM Transactions on Embedded Computing Systems 6(3) (2007).

[53] Lee, S., Shin, D., Kim, Y., Kim, J.: LAST: locality-aware sector translation for NAND flash memory-based storage systems. (2008).

[54] Li Y., He B., Yang R. J., Luo Q., Yi K., “Tree Indexing on Solid State Drives,” PVLDB, 2010.

[55] Li, Z. and Ross, K. A. Fast joins using join indices. VLDB Journal, 1999.

[56] Lim H., Fan B., Andersen D., Kaminsky M., “SILT: a memory-efficient, high-performance key-value store”, SOSP, 2011.

[57] Micron Technology, Inc., "Micron Announces Availability of Phase Change Memory for Mobile Devices", http://investors.micron.com/releasedetail.cfm?ReleaseID=692563, 2012, retrieved on 2012-07-30.

[58] Moglen E., “FreedomBox”, http://freedomboxfoundation.org, retrieved on 2012-06-15.

[59] Muth P., O'Neil P., Pick A., Weikum G., “The LHAM log-structured history data access method”, VLDB Journal, 2000.

[60] Nath, S., Gibbons, P. B.: Online maintenance of very large random samples on flash storage. Proc. VLDB Endow. 1, 1 (August 2008), 970-983.

[61] Nath, S., Kansal, A. FlashDB: dynamic self-tuning database for NAND flash. In Proceedings of the 6th international conference on Information processing in sensor networks (IPSN '07). ACM, New York, NY, USA, 410-419.

[62] O’Neil P., Cheng E., Gawlick D., O’Neil E., “The log-structured merge-tree (LSM-tree)”, Acta Informatica, 1996.

[63] Oracle Corporation, “Database Lite 10gR3,” 2008.

[64] Oracle Corporation, “Oracle Berkeley DB,” June 2011.

[65] Park, C., Cheon, W., Lee, Y., Jung, M., Cho, W., Yoon, H.: A Re-configurable FTL (Flash Translation Layer) Architecture for NAND Flash based Applications. In:

Proceedings of the 18th IEEE/IFIP International Workshop on Rapid System Prototyping, IEEE Computer Society Washington, DC, USA (2007) 202–208.

[66] Pucheral P., Bouganim L., Valduriez P., Bobineau C. "PicoDBMS: Scaling down Database Techniques for the Smart card", VLDB Journal, 2001.

[67] Robshaw, M., Billet, O. New Stream Cipher Designs - The eSTREAM Finalists, LNCS 4986, 2008.

[68] Rosenblum M., Ousterhout J., "The Design and Implementation of a Log-Structured File System", ACM TOCS, 1992.

[69] Schmid P., Roos A., "SDXC/SDHC Memory Cards, Rounded Up And Benchmarked", http://www.tomshardware.com/reviews/sdxc-sdhc-uhs-i,2940-10.html, retrieved on 2012-06-15.

[70] Severance D., Lohman G., "Differential files: their application to the maintenance of large databases". ACM TODS, 1976.

[71] STMicroelectronics, "STM32 F-2 Series 32-bits Microcontrollers", http://www.st.com/internet/mcu/product/245085.jsp, retrieved on 2012-06-15.

[72] Sundaresan P., "General Key Indexes", US. Patent n° 5870747, 1999.

[73] Sweeney, L. k-anonymity: a model for protecting privacy. Int. J. Uncertain. Fuzziness Knowl.-Based Syst, 2002.

[74] Vingralek, R.: GnatDb: a small-footprint, secure database system, In VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases (2002), pp. 884-893.

[75] 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.

[76] Weininger, A., “Efficient execution of joins in a star schema”, SIGMOD, 2002.

[77] Wu, C., Chang, L., and Kuo, T. An Efficient B-Tree Layer for Flash-Memory Storage Systems. RTCSA, 2003.

[78] Xiao, X. and Tao, Y. Output perturbation with query relaxation. VLDB, 2008.

[79] Yao S., “Approximating the Number of Accesses in Database Organizations”, Communication of the ACM, 1977.

[80] Yin, S., “Un modèle de stockage et d’indexation pour des données embarquées en mémoire Flash”. PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, 2011.

[81] Yin, S., Pucheral, P. and Meng, X. A Sequential Indexing Scheme for flash-based embedded systems. EDBT, 2009.