# Systèmes de gestion de base de données embarqués dans une puce électronique(Database Systems on Chip)

Nicolas Anciaux

*N° d'ordre : _____*

*Université de Versailles Saint-Quentin-en-Yvelines*

# THÈSE / PhD THESIS

Pour obtenir le grade de

**docteur**

*discipline :* **Informatique**

présentée et soutenue publiquement

par

Nicolas ANCIAUX

Le 17 décembre 2004

Sur le sujet

# Systèmes de gestion de base de données embarqués dans une puce électronique

(Database Systems on Chip)

_____

**JURY**

Professeur Philippe PUCHERAL, *Directeur de thèse*

Professeur Michael J. FRANKLIN, *Rapporteur*

Professeur Patrick VALDURIEZ, *Rapporteur*

Docteur Philippe BONNET, *Examinateur*

Docteur Luc BOUGANIM, *Examinateur*

Monsieur Jean-Claude MARCHETAUX, *Examinateur*

*à Ani,*

*à M'An, Sèv, Angel et Garooj…*

# Remerciements – Acknoledgements

Merci d'abord à Benoît sans qui cette thèse n'aurait pas vu le jour sans, vu que je n'aurais jamais découvert l'informatique. J'y pense chaque matin que fait le calendrier. Merci aussi énormément à Miflo sans qui non plus je n'aurais pu vivre ces quelques mois qui m'ont procuré ces années de plaisir.

Et merci aussi à M'An et Vincent, qui m'ont poussé et aidé dans mon parcours lorsqu'il était pénible, et sans qui je n'aurais pu passer ces 4 ans de plaisir.

Re-Merci encore, Luc, pour avoir su me motiver, supporter mes égarements durant ces 4 années, et surtout pour ces moments de recherche excitants et agréables. Je voudrais revivre aussi ces instants de sud, de conférence rigolard, et de trajets en moto. Merci surtout pour cet encadrement génial et tout le temps que tu m'as donné et pour ton amitié.

Merci Philippe, pour savoir recarder le travail sans jamais décourager, pour m'avoir accueilli en DEA et m'avoir ouvert les portes de l'équipe formidable que tu as su construire à l'INRIA, pour ta façon de voir les problèmes simplement et avec rigueur même sans couleur. Merci à toi comme à Luc pour avoir rendu aussi impérissables et joyeux le souvenir qui me reste de ces années de thèse, et tes conseils de vie aussi.

Merci aussi à Jean-Claude Marchetaux, pour tes compétences et ta patience, et pour tes explications précieuses. Mes contacts avec toi m'ont beaucoup apportés et m'ont donné une jolie image du monde de l'entreprise. Merci aussi pour le temps que tu as passé sur PicoDBMS. Sans toi et Alain Boudou nous n'aurions pu produire ce travail.

Thank you very much to Mike Franklin and Patrick Valduriez for reading this work during a very busy period. Someone told me that PhD reports, in average, are read by two or three people in addition to their author. Your are the first two of these people in reading my document, and it is a great honor already.

Merci à Georges Gardarin et Patrick pour votre soutient, qui me permet d'envisager encore de nombreuses années de recherches. J'espère prendre autant de plaisir que j'en ai pris ici dans la suite.

Merci aussi à Chantal Ducoin qui m'a administrativement sauvé la mise plusieurs fois !

Merci Aussi à Christophe et Saida pour ces instants de cohabitation dans notre bureau du PRiSM. Merci aussi Philippe, Luc, Thierry, Béatrice et Zoubida pour vos précieux

conseils de préparation de cours et de TD.

Merci à tous ceux avec qui nous avons passé ces petits moments de détente entre cigarettes et machine à café, Jean-Pierre, François, Sophie, Lilan, Christian et Aurélian.

Merci à Minère, Dods, Maia, Marion, Jacky, Dji-Luc, et à toute la famille, d'avoir entrepris pour certain un si long voyage pour être présent à la soutenance, et merci à mes beaux-parents, Angel et Garooj, qui sont parfois passés en France ces derniers temps et ont du dormir avec la lumière de l'ordinateur allumé.

Et enfin, merci Anaenza d'avoir pu te libérer pour ma soutenance, et merci pour les fous-rires, pour ta voix joyeuse, et pour ton amour pour les excès et les passions en tout genre, merci pour ton amitié.

Nicolas Anciaux

# Table of Content

# List of Figures

# List of Tables

# Chapter 1 – Introduction

We are moving towards the third wave of computing, in which a large number of specialized computing devices will be everywhere, serving everybody. Mark Weiser, the pioneer of ubiquitous computing, was the first to paint this vision in the early 1990s [Wei91]. He defined ubiquitous computing as the opposite of virtual reality: "Where virtual reality puts people inside a computer-generated world, ubiquitous computing forces the computer to live out here in the world with people."

In this vision, people are surrounded by an electronic environment made of a proliferation of *smart objects*. The smart objects composing this environment would be aware of people's presence and needs, and could adjust their actions in consequence. The concept of smart object appears in the middle of the 1990s, at the crossroad between component miniaturization, information processing technologies and wireless communications.

We see smart objects as chips endowed with data acquisition, data storage, data processing, and communication abilities. Examples are smart cards (Java multi-application cards, banking cards, SIM cards for cell phones, identification cards etc.), sensors monitoring their environment (gathering weather, pollution, traffic information, etc.), embedded chips in domestic appliances [BID+99, KOA+99, Wei96] (*e.g.*, embedded chip in a set top box). Medicine researchers also expect chips (*e.g.*, Smartdust [KKP]) to be ingested or inhaled (*e.g.*, to monitor sugar rate in blood and automate its regulation for diabetics).

Since smart objects acquire and process data, the need for embedded database techniques arises. For instance, in [BGS01], networks of sensors collecting environmental data are compared to distributed databases, each sensor acting as a micro-data server answering queries. [MaF02] brought out the need for executing local computation on the data, such as aggregation in push-based systems, in order to save communications and thereby power consumption. Local computation is also required to participate in distributed pull-based queries [MFH+02]. Also, protecting the confidentiality of collected contextual data or portable folders (*e.g.*, healthcare folders, phone and address books, agendas) stored in secure smart objects leads to embed sophisticated query engines to prevent any information disclosure [PBV+01]. These examples are representative of the increasing need for embedding more and more database functionalities in smart objects.

This manuscript focuses on the impact of smart objects' hardware constraints on

embedded database techniques. The first section highlights these hardware constraints and motivates the need for embedding database components in small devices. Section 2 presents the main issues addressed during the thesis. Section 3 summarizes the three main contributions developed in this document. Section 4 briefly presents the prototypes implemented to assess the different contributions of this thesis. Finally, Section 4 gives the outline of the manuscript.

# 1 Context of the study

This section describes the constrained computing environment of smart objects and motivates the need for embedding database components in smart objects.

Smart objects have highly constrained resources due to technical and economical motives. Smart objects have to be physically small. Indeed, several application domains require smart objects to be small enough to be embedded (*e.g.*, in domestic appliance), disseminated (*e.g.*, sensors), portable (*e.g.*, smart cards), ingested (*e.g.*, medical applications), etc. This reduced size obviously impacts the available resources notably in terms of stable storage capacity and RAM size. Moreover, many smart objects are autonomous, and then energy considerations conduct also to limit the resources notably in terms of CPU speed and communication bandwidth. Security exacerbates the previous concerns, the chip size (*e.g.*, in smart cards) being reduced to render the cost of attacks prohibitive and to avoid tampering by monitoring the chip activities (*e.g.*, monitoring the data bus). Finally, economical considerations lead to reduce the production cost of mass-market appliances to their minimum. For a given fabrication process (*e.g.*, 0.18 micron), the cost of the chip is directly linked to its size, therefore on-board resources must be carefully calibrated.

Designing embedded database components for smart objects is an important issue [ABP03c] for the three following reasons:

Smart objects may collect or store information related to individuals, thus leading to privacy preservation concerns. As soon as the confidentiality of the acquired data is a concern[1], the query processing must remain confined in the smart object to build and deliver only the authorized subpart of the embedded data. In that case, the complexity of the embedded query engine depends on the sophistication of the access rights. In the future *aware environment* [BID+99, KOA+99], many programs and physical users may interact with disseminated smart objects, each one with specific access rights.

Saving communication costs represents another motivation for embedding database components. Indeed, in a wireless context, energy consumption is directly linked to the amount of transferred data. Therefore, database filtering and aggregation facilities, and more

generally any database functions allowing the reduction of transmitted data, are indispensable partners of smart objects. Since communication savings allow reducing energy consumption [MFH+03], this motivation will remain valid for a while.

Finally, embedded data management techniques are required in every context where computations have to be performed in a disconnected mode. For example, smart objects embedded in a human body to monitor the cardiac pulse and rapidly liberate chemical medication in case of abnormal activity, must process the data locally without relying on any remote operation.

Therefore, confidentiality, communication and thus energy saving, and disconnected activities are three motivations for embedding database components in smart-objects. The design of embedded database components for smart objects is thus an important problem. This problem is complicated given that each architecture exhibits specific properties (*e.g.*, energy consumption, tamper resistance) as well as specific hardware constraints (*e.g.*, bounded silicon's die size, hardware resources asymmetry).

## 2 Addressed Issues

This dissertation focuses on three different and complementary facets of the above problem. The first issue addressed is the RAM consumption problem in the design of database components embedded on chips. This study is motivated by the fact that the RAM turns to be the most limiting factor in a smart object context. The second and third issues addressed are more specifically related to the smart card environment, the today's most widespread and secure representative of smart objects. More precisely, our second study capitalizes on the PicoDBMS study [PBV+01] to develop and assess processing techniques for evaluating authorized database views in a smart card. Finally, our third study proposes to bypass the storage limitation of the smart card by using remote - but insecure - storage space. The unifying thread of these three studies is the query evaluation problem in hardware-constrained environment. We discuss below the motivation of each of these three studies.

*RAM-constrained query processing techniques:* Although the computing power of lightweight devices globally evolves according to Moore's law, the discrepancy between RAM capacity and the other resources, notably CPU speed and stable storage capacity, still increases. This trade-off is recurrent each time the chip's size needs be reduced to match the aforementioned constraints such as thinness, energy consumption, tamper resistance or production costs on large-scale markets. In addition, the large size of RAM cells compared with stable memory cells (16 times larger than ROM cells) makes matters worse. Since the RAM is highly reduced, strategies to overcome main memory overflows must be found. However, query evaluation techniques cannot rely on swapping, as in traditional DBMSs, for

[1] Even ordinary data may become sensitive once grouped and well organized in databases.

several reasons. First, smart objects use commonly electronic stable memory like EE-PROM or Flash memory. Although these technologies provide compactness (small cell size), reduced energy consumption and production costs, their writing costs are extremely high (up to 10 ms per word in EE-PROM) and the memory cell lifetime is limited to about $10^5$ write cycles. Second, a swapping area would compete with the area devoted to the on-board persistent data and there is no way to bound it accurately. Obviously, if we assume that external resources can be exploited, the RAM problem vanishes. But, as stated previously, communication costs, data confidentiality and disconnected activities lead to the execution of on-board queries. As a conclusion, RAM will remain the critical resource in most smart object environments and the ability to calibrate it against data management requirements turns out to be a major challenge.

*Query processing techniques for smart card databases:* Smart card is today the most widely represented smart object (roughly two billion smart cards are produced annually). Moreover, smart card is considered as the most secure computing device today and is thus used in many applications exhibiting high security requirements such as banking, cellular phone, identification, and healthcare. In data centric smart card applications (*e.g.*, personal folder on chip), confidential data reside inside the chip, and thus benefit from the smart card tamper resistance. However, this forces to embed a specific code in the chip, which authenticates the users and solely delivers the data corresponding to their access rights (*i.e.*, authorized database views). A trivial solution to this problem could be to embed a file system and to materialize each authorized database view in a separate file. However, this solution badly adapts to data and access right updates and results in a huge data replication among files, hurting the smart card storage constraint. Thus, a dynamic evaluation of the authorized views is highly required. To this end, ad-hoc query evaluation techniques matching the strong smart card hardware constraints have to be devised. Current powerful smart cards are endowed with 64 kilobytes of stable storage (EEPROM), 4 kilobytes of random access memory (RAM) from which only hundreds of bytes are available for the applications (most of the embedded RAM is reserved for the operating system), and 96 kilobytes of read-only memory (ROM) holding the operating system and possibly a Java virtual machine. These smart cards are equipped with CPU powerful enough to sustain cryptographic operations (typically 32 bits CPU running at 50MHZ). Such hardware architectures exhibit an untraditional balance in terms of storage and computing resources. First, the discrepancy between stable storage reads and writes is untraditional since reads are random and very fast (comparable to RAM reads), while write latency is very high (several ms/word). Second, the CPU processing power is oversized compared to the on-board amount of persistent data. Furthermore, smart cards are not autonomous, *i.e.*, they have no independent power supply, which precludes asynchronous and/or disconnected processing. Thus, the definition and assessment of query processing techniques dedicated to existing and future smart card platforms are highly required.

*Tamper-resistant query processing techniques for external data:* Given the smartcard

tiny die size, the amount of stable storage will remain limited to few megabytes for a while. Indeed, new stable memory technologies (*e.g.*, micro- and nanomechanical technology [VGD+02]) are not expected to be available soon. An interesting challenge is therefore to use remote resources to increase smart cards abilities. Since smart cards are plugged into or linked to a more powerful device (*e.g.*, PC, PDA, cellular phone), the objective is to take advantage of the device's storage capabilities to overcome the smart card limitations. This proposal is particularly well suited to emerging smart card based architecture where a secure smart card chip is connected to large but insecure memory modules. For instance, the Gemplus' SUMO "smart card" provides 224 megabytes of FLASH memory dispatched within the plastic's substrate. Another example is the MOPASS consortium [MOP04], which proposes to combine FLASH memory cards with a smart card. However, the issue is to use remote resources without losing the smart card benefits, *i.e.*, the high level of security provided to the data. Since the data is stored remotely in an insecure environment, its confidentiality can be violated and its contents can be altered (even if encrypted). This motivates the study of query processing techniques that allow to externalizing persistent (and intermediate) data on an insecure remote storage while keeping the same tamper-resistance as smart card databases.

# 3 Contributions

As noted earlier, smart objects hardware configurations are more and more specialized to cope with specific requirements in terms of lightness, battery life, security and production cost. Building an ad-hoc query evaluator for each of them will rapidly become cumbersome and, above all, will incur a prohibitive design cost. Thus, there is a clear need for defining pre-designed and portable database components that can be rapidly integrated in smart objects. The different studies conducted in this thesis typically address this requirement. We summarize below our contributions related to the three research issues highlighted in the preceding section.

*RAM constrained query processing techniques:* we propose a framework for designing RAM-constrained query evaluators allowing to process SQL-like queries on on-board relational data. Our study follows a three-step approach. First, we devise a *RAM lower bound query execution model* by focusing on the algorithmic structure of each relational operator and on the way the dataflow must be organized between these operators. Second, we propose a new form of optimization to this lower bound execution model, called *iteration filters*. Iteration filters drastically reduce the prohibitive cost incurred by the previous model, without increasing its memory requirement. Finally, we study the *impact of an incremental growth of RAM*. We propose an adaptation of the previous execution techniques that best exploit each RAM incremental growth and we conduct a performance evaluation. We observe that very small RAM growths may lead to considerable performance gains and show that the proposed techniques constitute an accurate alternative to the index in a wide range of

situations. These evaluations allow proposing co-design guidelines helping to find the best compromise between RAM capacity, volume of on-board data, query complexity and response time.

*Query processing techniques for smart card databases:* our second contribution deals with the management of complex access rights on embedded data, coping with the constraints of the smart card. This work capitalizes on the previous PicoDBMS study [PBV+01], which proposes to embed a full-fledged DBMS on a smart card in order to provide highly portable and secure database folders, with medical applications in mind. Our contribution is twofold. First, we categorize the different access rights requirements that may be defined on any kind of embedded data, with the objective to generalize the PicoDBMS approach. To this end, we define three main categories of authorizations, of increasing complexity, covering a wide range of situations: schema, occurrence and computed data authorizations. We then define seven access right types derived from these three categories that should be supported by a PicoDBMS-like system. Second, we make an in-depth performance evaluation: for each access right type, we measure the response time and/or the throughput when processing the corresponding authorized views (implementing the given access right). We investigate different storage and indexation alternatives and assess the effectiveness of each of them according to the amount of embedded data and to the access rights required by the application. The conclusions of this performance evaluation allow to decide which database technique/strategy should be embedded in a smart card with respect to a targeted - class of - application.

*Tamper-resistant query processing techniques for external data:* our third contribution addresses the way to circumvent the smart card stable storage limitation using remote insecure storage. Since the data is stored remotely in an insecure environment, we use cryptographic techniques to ensure its confidentiality and its tamper-resistance. As opposed to several existing studies [HIL+02, BoP02, DDJ+03], we propose to reduce to a minimum the amount of processing that may be delegated to the device handling the insecure storage for three distinct reasons. First, the external device being insecure, the honesty of any processing done externally has to be checked afterward. Actually, such verification might be as expensive as the processing itself. Some techniques indeed exist for simple selections [GKM+04] but the problem remains open for more complex operations. Second, even if the preceding point could be solved, allowing processing on the encrypted data generally disclose some (fuzzy) information. Third, the external device considered may not have processing capabilities (*e.g.*, SUMO, MOPASS). These reasons led us to consider the external storage as a file system delivering on requests encrypted data blocks. To increase performance, we show that each file should be divided in minimal data blocks and that each block should include its own tamper-resistance information. We then describe an adequate storage model relying strongly on encrypted indexes, and algorithms for processing queries in the smart card.

## 4 Prototypes

Three prototypes have been implemented in parallel to the studies conducted during this thesis. These three implementations are chronologically depicted in the following.

A first prototype, called PicoDBMS, has been implemented during the year 2001, to study in practical the DBMS design proposal dedicated to the smart cards context envisioned in [PBV+01]. PicoDBMS has been build using the JavaCard language (subset of Java for smart cards) on a smart card prototype provided by Bull-CP8 (now Axalto, the smart card subsidiary of Schlumberger). The main objective of this implementation was to assess the feasibility of the full-fledge DBMS. However, performance measurements could not been conducted using this plate-form, due to the inadequate implementation of the on-board virtual machine in case of PicoDBMS like data centric applications (e.g., security controls increase each memory access by more thant 3 orders of magnitude, the RAM was totally disabled in the provided prototype precluding storing execution variables on the heap).

A second prototype has been implemented during year 2002 to conduct the RAM aware query processing study. This study has been conducted considering flat data (no indices), a query execution engine based on repetitive access to the base data, and a small amount (bounded) of additional RAM available to materialize intermediate result. The context of this study (flat data, no indices, available RAM) and the envisioned optimization techniques (iteration filters) are highly different from the PicoDBMS prototype. Thus, a second prototype has been build on a PC, using the JavaCard language, to dump execution statistics into files used to give a response time based on a cost model calibrated with the smart card characteristics.

The third prototype has been build during year 2003 and 2004 to measure the precise performance of PicoDBMS on an enhanced smart card prototype provided by Axalto, endowed with the native operating system Zeplatform (programmed in C Language). The PicoDBMS code available in JavaCard has been ported to C, and the operating system has been modified in partnership with Axalto to adapt to data centric applications. This prototype has been intensively studied to provide the performance measurements presented in Chapter 4. In parallel to this implementation on the real smart card, the code has been ported to a smart card hardware simulator (also provided by Axalto) that can be used for prospective measurements assuming future smart cards endowed with larger stable memories.

## 5 Outline

This study is at the crossroad of three domains: smart cards hardware architectures, data management techniques, and cryptography. The second chapter of this thesis gives the necessary background on these three domains in order to make the proposed contributions

understandable[2]. Chapter 3 focuses on embedded query processing in a RAM constrained environment and delivers co-design guidelines helping to find the best compromise between RAM capacity, volume of on-board data, query complexity and response time. Chapter 4 investigates different storage and indexation alternatives for smart card databases and assess the effectiveness of each of them according to the amount of embedded data and to the access rights required by a target smart card application. Chapter 5 investigates cryptographic and query processing techniques to make the management of data externalized in an insecure storage tamper-resistant. Finally, Chapter 6 concludes the document and opens up new research perspectives.

[2] The background presented in each subsection (smart card, database and cryptography) can be skipped by specialists of the corresponding area(s).

# Chapter 2 – Smart Card, Databases and Cryptography Background

*The study conducted in this document is at the crossroad of three domains: smart card architecture, database query processing and cryptography. This chapter gives the needed prerequisites of theses area in order to make the proposed contributions understandable.*

## 1 Smart Card Technology

A smart card is made of two main components: a plastic substrate (PVC or polyester resin) and a module, which contains a small integrated circuit. Its average price is between 50 cents and $4 [Kim04], depending on its internal complexity.

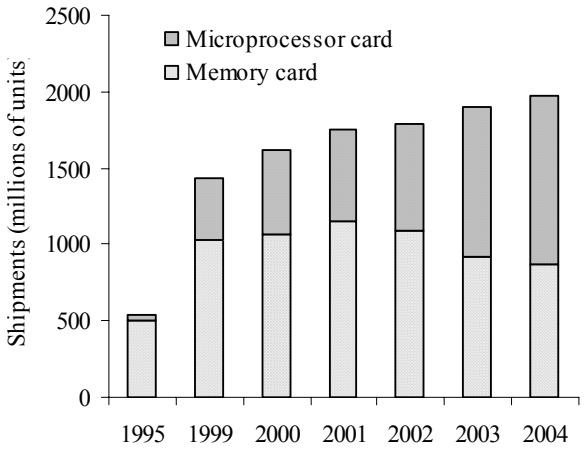



**Figure 1** *Smart Cards Shipments*[3].

Many manufacturers offer smart card products. The top five card manufacturers in volume of cards sales are Gemplus, Axalto (a Schlumberger subsidiary), Giesecke&Devrient, Oberthur, and Orga [Gar03]. The top ten smart card integrated circuit chip manufacturers are Infineon technology, ST Microelectronics, Philips Semiconductor, Atmel, Fujitsu, Hitachi, NEC, Samsung, Sony and Xicor [Mar03]. This is a very competitive

[3] Source : Gartner Dataquest survey and www.eurosmart.com, May 2003.

market, which introduces many product variants and innovations. Since 1999, more than 1.4 billion smart cards have been shipped each year, and this number will probably reach 2 billion this year (see Figure 1). Moreover, the share of microprocessor cards (the most elaborated type of smart cards) represents now more than half of the smart card market, compared to 25% four years ago. The current trend leads clearly to more powerful cards holding more data and programs and penetrating many domains including health care, loyalty, cellular phones, electronic payments, identity cards and driving licenses.

Revenue generated by smart card sells reached $1.40 billion in 2003. In the future, analysts consider that smart cards will have an impact as big as the emergence of PCs [Sha02a] and revenue is expected to reach $2.38 billion by 2007[4].

The following sections introduce some prerequisite knowledge required for an in-depth analysis of smart cards data management techniques. Section 1.1 describes the specific smart card environment while Section 1.2 depicts the unusual balance of resources in the on-board integrated circuit. The objective is to state the main difference between this constrained environment and traditional database ones. For an in-depth presentation of smart cards technology, we refer the reader to the IBM red book [IBM98], the smart card security and applications book [Hen01] and the U.S. government smart card handbook [Hol04].

## 1.1 Smart Cards Computing Environment

According to Webster [Web], a smart card is "a small plastic card that has a built-in microprocessor to store and process data and records". However, this concept covers several devices, which are differentiated by both the capabilities of the on-board *Integrated Circuit Chip (ICC)* and by the interface used to communicate with the reader. Section 1.1.1 describes the different types of ICC while Section 1.1.2 focuses on the interface aspect. Section 1.1.3 enumerates the various ways one can use to break into a smart card and introduces the many security features developed to make smart cards tamper-resistant. Section 1.1.4 highlights recent initiatives to make smart cards an open platform. Current and future smart cards applications are presented in Section 1.1.5. Section 1.1.6 depicts future smart cards enhancements. In conclusion, Section 1.1.7 describes how the smart card unusual computing environment impacts the design of embedded database components.

### *1.1.1 Memory Card versus Smart Cards*

There are two main types of chips that can be associated with cards: memory chips which are possibly enhanced with hard-wired procedures, and chips endowed with a microcontroller.

*Memory-Only ICC* is a data storage medium, which does not contain logic nor perform calculations. It is usually dedicated to a single application (*e.g.*, phone cards) and provides a

[4] Source: Frost & Sullivan, www.Smart cards.frost.com.

low level of security, slightly more secure than a *magnetic stripe card* [Hol04]. It can be enhanced with hard-wired modules to provide higher security (*e.g.*, authentication features, protection against updating the data from the outside, encryption abilities [IBM98]). Note that since the embedded intelligence is hard-wired, it can only be modified or upgraded by a complete redesign of the chip. Typical applications of these devices include phone cards, pre-paid cards, car parking cards, and public transport cards.

*Microcontroller ICC* embeds on a single silicon die a processor, some memory for programs and data (all or part of it can be updated many times), I/O lines and security modules. It can be viewed as a highly secure tiny computer fitting in a wallet. All it needs to operate is power and a communication terminal. Unlike memory-only products, this ICC has been designed to meet security targets. Therefore, security modules and cryptographic co-processor are added to ensure the card tamper resistance and efficient cryptographic features. This type of card with an embedded chip has a wide range of applications including access control (*e.g.*, Hilton hotels check-in), secure and portable personal storage (*e.g.*, SIM cellular phones), user environment (*e.g.*, campus card), electronic payment (*e.g.*, Geldkarte in Germany), secure messaging and voting (*e.g.*, digital signature card) and health card (*e.g.*, Vitale II in France).

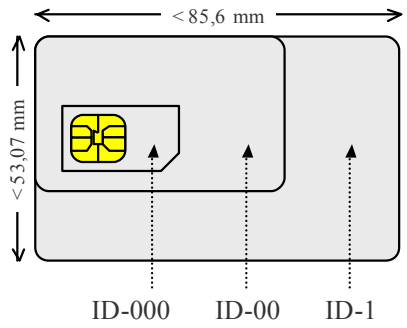



**Figure 2** *Smart Card's Normalized Formats (ISO and GSM).*

Today's smart cards market offers a variety of memory-only and microcontroller cards. However, only microcontroller endowed cards will be addressed in this document since memory cards are not suitable for secure data management. In the following, we use the term smart card for microcontroller endowed cards.

Smart cards exist in various standard formats (see Figure 2). The most widely known is the ID-1 format [ISOp1] since it is also shared by *Embossed cards*[5] and *magnetic stripe cards*[6]. The next most frequently used format is the ID-000 format used for *Subscriber Identification Module* (SIM) cards [ETS99, GSM] integrated into mobile phones. Personal information and preferences are stored securely on a SIM card (PIN code protection) and are

[5] Cards with a text in relief or design that can be easily transferred to paper (*e.g.*, carbon copy of credit card payment).

[6] Cards storing up to 1000 bits of data on its magnetic stripe.

movable from one phone to another.

### *1.1.2 Physical Interface*

Smart cards are passive computing devices: they only respond to remote commands in a query/answer manner, but never make any standalone computation. Smart cards always take electrical energy and clock signal from a reader [Tua99]. The link between a card and a reader can be physical through direct electrical contacts (*contact cards*) or wireless, based on radio or electromagnetic waves (*contactless cards*).

*Contact smart cards* require insertion into a reader with a direct connection to the I/O, Vcc and CLK ports (see on Figure 5). The ISO 7816-4 [ISOp4] is the main standard for smart card communications and specifies the data exchange commands called APDU (Application Protocol Data Unit). In this model, the card is viewed as a server, waiting for APDU commands to process, eventually producing a result. The card and the terminal most often communicate using a serial connection, with a 1.2 kilobytes per second transmission rate. Currently, USB card connections appear either as ID-1 format cards using a USB reader or under the ID-000 format in USB keys [Gor00, Axa03, Jun03]. The USB interface allows transmission rates as high as 1.2 megabytes/s. While smart cards currently available on the market don't exhibit such a high transmission rate, manufacturers and academic researchers view the delivery constraint as a very short-term issue (see [RES03] Chapter 4: technology challenges, synthetic overview). In fact, our industrial partner Axalto [Axa] already makes smart card prototypes with a 500 kilobytes/s transmission rate.

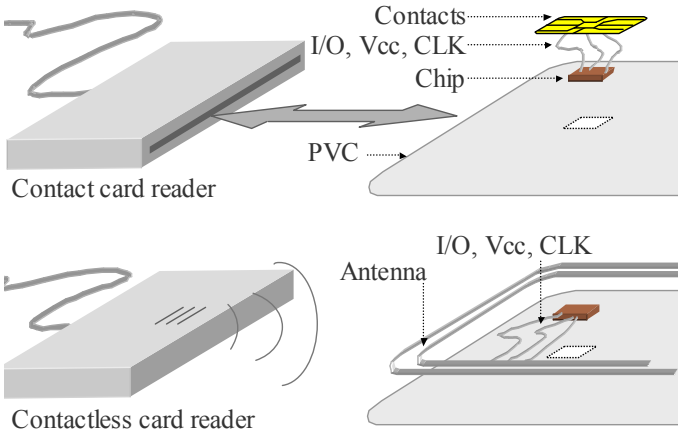



**Figure 3** *Data Transmission between Smart Card and User.*

*Contactless smart cards* include an antenna embedded in the plastic substrate [ABR+01]. This antenna supplies energy to the chip when the card is placed in near proximity to the reader (usually within ten centimeters[7]), and enables data exchange to take

[7] This small distance has also a security role since messages can be intercepted or altered whether long distance exchanges occur. So, long distance induces communication's encryption.

place. The contactless interface complies with the ISO 14443 standard [ISO14], which specifies high transmission rates, up to 800 kilobytes/s. This standard has also been developed to be compatible with the contact smart card standard and provides the same format for data exchanges (APDU format [ISOp4] mentioned above). The two main design of contactless smart cards are the *hybrid smart card design* which has two chips on the card and the *dual-interface smart card design* which has a single chip with traditional contacts plus an antenna (*e.g.*, Sharp RK410 smart card product [Sha02b]). In both cases, these smart cards can be accessed either in a contact or contactless fashion. Currently, the many working groups pushing for more standardization in the contactless area, like the *Secure and Interoperable Networking for Contactless in Europe* (SINCE) initiative [SIN, SIN03c, SIN04a, SIN04b], reveal the growing interest for wireless interfaces. For further details on contactless technology see [SIN02a].

Beside their different interfaces, contact and contactless smart cards differ in their application domains. Generally contactless applications are restricted to identity control (*e.g.*, transportation, building access, etc.). In this manuscript, we will thus consider contact smart cards.

### *1.1.3 Security*

The inherent security offered by smart cards often makes them the trusted party [Ber98] of a secure procedure including identification, protection of data stores [Big99], financial transactions [ARS99], e-commerce [She99], cryptographic protocols, and network security [MuD02]. Indeed, smart cards have been created to reduce payment frauds in the early 80's by replacing magnetic stripe cards, which do not protect the stored data against unauthorized access. Since their inception, smart cards are designed to provide a high level of security, which makes them, as of today, the most secure computing device. This section describes the inherent tamper resistance of smart cards, and explains to which extent they can be considered as a highly trusted environment. For further information about internal security in smart cards see [Cha97, Phi97, NPS+03, ScS99].

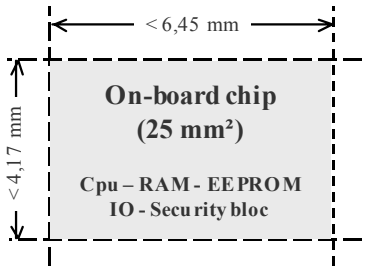



**Figure 4** *Embedded Chip Dimension (ISO).*

*Invasive attacks* [TrC], such as m*icro-probing techniques* [BFL+96], access the microcontroller directly, thus the pirate can alter the integrated circuit and interpret (part of) the embedded data. These techniques require to extract the chip from the plastic substrate

and to remove the surrounding resin using acid. Even if the smart card can be used after tampering, the piracy is clearly visible. Moreover, the small size of the chip (bounded to a 25-mm² by the ISO standard [ISOp1] as shown Figure 4) makes the cost of these attacks prohibitive. They are made even harder with recent smart card design where additional layers of metal covering the chip are used to detect invasion attempts and are automatically triggering the destruction of the smart card's confidential content (*e.g.*, light is detected by silver plate layers).

*Non-invasive attacks*, such as *software attacks*[8], *eavesdropping techniques*[9] and *fault generation techniques*[10] [BCN+04], are much more dangerous since they usually can be reproduced at a low cost [AnK97], once a single smart card has been cracked. Moreover, the tampered card remains physically intact and the equipment used in the attack can usually stand for a normal smart card reader. Traditional smart card contacts (see Figure 5) are often used to accomplish the tampering. Indeed, smart cards don't have their own battery, hence the reader must supply their power (Vcc wire), whether they have a contact or contactless interface. The internal clock (CLK) is also set externally [ISOp2]. All these parameters can be modified and/or monitored by the hacker, as shown in [AnK96].

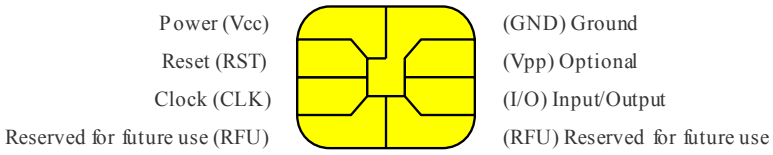



**Figure 5** *Smart card's Contacts (ISO).*

To counter software attacks, the internal cryptographic modules are made in hardware and libraries are checked against security holes (*e.g.*, Hitachi AE5 smart cards use a Certified Crypto Library). Hence, the embedded algorithms have to be proved secure [BDL97] before being deployed and widely used. This method enforces protection against eavesdropping attacks as well. Also, the radiations produced by the processor during normal operations are limited to a minimum to hide them, and the power consumption is maintained roughly constant as well as the chip's temperature. Moreover, fault generation is avoided thanks to on-chip sensors (low frequency and/or voltage alarms[11]) that detect abnormal input signals and then disable the chip. These sensors are also hardware protected since disabling the sensor destroys the whole chip. Further security enhancements are based on randomized clock signal, which however reduce the processing speed by a factor of 4 as explained in [KoK99] and self-timed chip [Phi04] technologies (*i.e.*, generates clock signal internally). Also, bus-scrambling methods protect the internal data transfers with alarm and partial encoding [MMC+02] from tampering or sensing. Finally, as shown in [Moo96], additional

[8] Exploit security vulnerabilities of the application's protocols using normally the card interface.
[9] Monitor the characteristics of smart cards connections and radiations produced by the processor.
[10] Use abnormal environmental conditions (clock cycles, power supply) to generate malfunctions and provide additional access.
[11] While tampered, the smart card clock is often considerably slowed down by the pirate.

security measures will be provided in the future thanks to multithreaded chips.

Security provided by smart cards can also reside in the operating system [AHJ+96] or in the application itself. Many studies propose code verifying methods [ReB00, Ler01, LaR98, BDJ+01] and assessment of applications code based on signatures before download into the card.

Currently, governmental agencies require security measures for devices holding personal data such as medical data, insurance, licenses, profile information, etc. Smart cards are therefore more and more attractive to hold medical data [Sma03a, Sma03c], be part of financial transactions [SPE03, Sma03b], and serve as individual ID multi-purpose card (see [Acc02] for e-government initiative examples).

Manufacturers rely on security criteria to define the security provided by a smart card. Common Criteria for Information Technology Security Evaluation (so called *CC* [CC]) are widely used since they are understandable by the international community (they compile existing European, US, and Canadian security criteria). The CC attributes a security level to computer devices from the *Evaluation Assurance Level 1* (EAL1) to *Evaluation Assurance Level 7* (EAL7). Other security levels for smart cards are also assessed by national organizations like NSA and NIST from the USA, CSE from Canada, CESG from UK, SCSSI from France, etc. Thus, each smart card product provides a certain level of security, which can comply with any national or international security standard. Examples of security evaluation process for smart cards are described in [Joi02, Hit02].

### *1.1.4 Open Platform*

Smart cards have been a closed environment for many years. They were built for a single application; the application's code and needed subset of the operating system were embedded in the card being *masked* permanently on-chip. The main issue with this approach was the increased time to market since developing a new card for each new application added several months to the overall development process.

Nowadays smart card technology provides the opportunity to download multiple applications on a single card, both for contact and contactless smart cards.

The two dominant and standardized operating systems[12,13] for multi-application smart cards are *JavaCard* [Che00] and *MULTi-application Operating System* (MULTOS) [Mao]. Embedded application must be programmed in JavaCard API [Sun00a] (a subset of Sun Microsystems Java language) to be successfully interpreted by the *JavaCard Virtual Machine* [Sun00c], or in the *Multos Executable Language* (MEL) [MEL] for MULTOS platforms.

[12] Microsoft Smart Card for Windows [Tal99] has been abandoned.
[13] These systems are called either operating systems or virtual machines in the smart card literature.

JavaCard is currently the leading platform [FrS04] for multi-application cards with 95% of the multi-application cards market share. In 2003, of the 228.3 million smart cards sold, 220 millions used the JavaCard virtual machine while only 8.3 million were based on MULTOS. In 2008, market unit shipments[14] are expected to reach 867 million for JavaCard and only 46.8 million for MULTOS. Examples of current smart card products running MULTOS are available in [Key01, Mao04].

At execution time, smart cards can be accessed by applications using many techniques such as RPC, RMI, Corba, HTTP, TCP/IP, etc. All the middleware for smart card is studied in [VRK03] and further details about smart cards multi-application operating systems are given in [Jea01, DGG+03].

Dynamic programs download and execution should not however compromise the inherently highly secure computing environment provided by smart cards [GiL99]. Although Java inherits natural security features since it is a strongly typed object language, the JavaCard platform also provides a *firewall* [Sun00b] between different applications embedded in the same card to ensure that each application accesses only its authorized memory sub-part, hence protecting application against software attacks. In the last five years, products incorporating the JavaCard platform have passed real-world security evaluations by major industries around the world, and JavaCard inter-application security comes now close to the one provided by distinct cards running each application separately [Sun01]. Nevertheless, many studies goes on enhancing security of JavaCard environment and programs [MaB00, Gir99, MoK99].

There are currently a large number of projects, associations and consortium focusing on the open smart card infrastructure. For example, organizations for multi-applications cards, like the *Smart Card Alliance* [Sma], the *Global-Platform* association [Glo, Kek03] and *eEurope Smart Card* [Eeu, OSC02] are publishing open smart cards specifications in order to accelerate the emergence of multi-application smart card technology and its widespread acceptance by.

In this open and multi-application infrastructure, data and processes must be protected from access by any un-authorized application or user. In such a context, data sharing and access control features provided by traditional DBMS would be a great addition.

### *1.1.5 Applications*

Smart cards are the most secure portable computing devices today. The first smart card was developed by Bull for the French banking system in the 80's to significantly reduce the losses associated with magnetic stripe credit card fraud. Since then, smart cards have been used successfully around the world for various applications [MIP02] involving identity control (banking, pay TV [Pot04], access to a network, physical access to buildings) and/or

[14] Source: "Battle of Platforms", an analysis from Frost & Sullivan (www.smart cards.frost.com).

personal data (telephony, loyalty, healthcare [Mal01], insurance, etc.). Even though smart cards are secure, their wider adoption has been impaired for two reasons: one being the increased time to market smart card based applications and the other is the limited amount of available memory. With the emergence of multi-application smart cards and their rapidly increasing capabilities, many industrial consortiums and large-scale government projects are now materializing.

For instance, applications complying with the *US Federal Government Smart Card Interoperability Specification* [NIS02] are currently used for employee identification and authentication to access building, to secure networks, but also to digitally sign or encrypt documents such as secure e-mail and administrative requests.

Recently, MasterCard introduced the *MasterCard Open Data Storage* specifications [Mas03] to enable cardholders and service providers (or any merchant third party) to store and retrieve objects directly on customers' smart cards, using very basic access rights.

Another example is the NETC@RDS Project [NET], which aims at improving the access of mobile European citizens to national health care systems using advanced smart card technology. This project involves the main health authority instances[15] of nine European countries.

Multi-purpose ID cards projects [MTS+04, SIN02b] (*e.g.*, passport, driving license, e-voting, insurance, transport, etc.) have been launched in many countries even outside of Europe including USA [USG03, OHT03, Kim04], Japan [Dep01] and China [APE03]. As a result, a large percentage of applications involving identity control and personal data would benefit from smart card technology [All95, Per02a].

We distinguish four major classes of smart card applications involving at least a subset of the features used by a DBMS:

- *Identification and offline transactions tools*: such applications belong to the traditional smart card market segment, which include credit cards, e-purse, SIM for GSM, phone cards, ID cards (holds biometric identity proof) like transportation cards, and crypto cards (gives secure access to a network). They involve few data (typically the holder's identifier and some status information), but might require efficient processing and communication abilities (crypto-card can encrypt e-mails on the fly). Querying is not a concern and access rights are useless since these cards are protected by PIN-codes. Their unique requirement is update atomicity.

- *Downloadable databases:* they are predefined packages of data (*e.g.,* list of restaurants, hotels and tourist sites, catalogs…) that can be downloaded on the card – for example, before traveling – and be accessed from any terminal. Data availability is

[15] For instance, the consortium includes in France, the GIE Sesam-Vitale (SESAM-Vitale Economic Interest Grouping), the CNAM (National Main Health Insurance Fund of Salaried Workers), the AP-HP (Hospitals of the region of Paris), etc.

the major concern here. Volume of data can be important and queries are often complex. The data are typically read-only and public.

- *User environment:* the objective is to store in a smart card an extended profile of the card's holder [Pot02] including, among others, data regarding the user (*e.g.*, biometric information [She03]) and the computing environment (*e.g.*, PC's configuration, passwords, cookies, bookmarks, software licenses…), an address book as well as an agenda. Queries remain simple, as data are not related. However, data is highly private and must be protected by sophisticated access rights (*e.g.,* the card's holder may want to share a subset of his/her address book or bookmark list with a subset of persons). Transaction atomicity and durability are also required.

- *Personal folders:* personal folders may be of different nature: scholastic, healthcare, car maintenance history, loyalty. However, they roughly share the same requirements. Note that queries involving data issued from different folders are possible. For instance, one may be interested in discovering associations between some disease and the education level of the cardholder. This raises the interesting issue of maintaining statistics on a population of cards or mining their content asynchronously.

| *Applications* | | *Volume* | *Select Project* | *Join* | *Group by Distinct* | *Access rights views* | *Atomicity* | *Durability* | *Statistics* |
|---|---|---|---|---|---|---|---|---|---|
| Money & identification | | tiny | | | | | √ | | |
| Data Centric | Downloadable DB | high | √ | √ | √ | | | | |
| | User environment | medium | √ | | | √ | √ | √ | |
| | Personal folder | high | √ | √ | √ | √ | √ | √ | √ |

**Table 1** *Typical Application's Profiles.*

Table 1 summarizes the database management requirements for the four classes of smart card applications listed above. Note that multi-application cards allow a single card to hold applications belonging to several classes, thus pushing to embed, by default, powerful DBMS kernel into smart cards.

Computing systems turn presently into the trusted information age. Security projects like *Palladium* [Mic02] led by Microsoft or *Trusting Computing Platform* (TCP) promoted by the *TCP Alliance* [TCP04], started to study techniques based on secure hardware and trusted software (relying on encryption, *Digital Signature* and *Digital Right Management*) to increase the security of personal computers [IBM, Mic02]. This initial effort recently

morphed into a larger project led by the *Trusting Computing Group* [TCG04], composed by the main hardware and software industries around the world including Intel, IBM, HP, Microsoft, AMD and Sony. The goal is to find ways to secure global networks and applications [Did04] made of a wide range of systems, from single computing devices like PDA, cellular phone or PC [Mic04], to a whole enterprise information system [Can04, TCG04]. In parallel, governmental councils [FTC02] are studying the feasibility of secure systems in terms of individual freedom while several users' associations protest against such kind of invasive control [NoT04]. In such a complex context, many perspectives can be envisioned.

First, the overall interest in security clearly leads manufacturers to offer more secure chips for computing systems. Representatives of these secure components are smart cards but also ibuttons [Ibut00, HWH01], secure mass memory cards (*e.g.*, *Secure Multimedia Memory Cards*), serial/parallel/USB secure dongles, and USB "crypto tokens" or rings (*e.g.*, *JavaRing*). Most of these devices, at least the last two, can be considered as smart card devices since they use an embedded 32-bit microcontroller to manage the security process and simply differ in their interface to the host they connect to [VRK03].

Second, main smart card manufacturers interest for security domains is driven by the multiple applications prospects [Yos03] and huge potential financial benefits (*e.g.*, applications to restrict free multimedia sharing and copying or to limit enterprise data files theft). Indeed, the inherent hardware security of smart cards and their computation power make them an ideal security token [DhF01]. Moreover, since security is managed by a smart card and thus resides outside of the core system, programs and even operating systems have no control over it. Smart cards can therefore implement secure access right policies stated by the owner, as opposed to mandatory security features imposed by computer manufacturers, which might be resented by end-users. Hence, smart cards can be viewed as the most easily user controllable component to guarantee security of computing devices owned by end-users.

All these reasons make the smart card an essential partner in such global security architectures, especially to manage security policies. In such an environment, embedding DBMS like access rights into smart cards is crucial.

### *1.1.6 Smart card's Future Directions*

The constraints of smart cards, particularly in term of data storage amount, can be a handicap to expand the smart card's market, as well as to satisfy the increasing user's demands in existing markets (*e.g.*, cellular phone market, see [Dis04]). Thus, smart cards manufacturers and researchers are trying to enhance the smart card architecture [CRn04].

To relax the storage limit, manufacturers extend the internal storage with external memory modules. In 1999, Gemplus announced Pinocchio [Gem99], a smart card equipped with 2 megabytes of Flash memory linked to the microcontroller by a bus. Since hardware

security can no longer be provided on this memory, its content must be either non-sensitive (*e.g.,* downloadable databases for which privacy is not a concern) or encrypted. Limit in storage capacity has also pushed other chip's manufacturers (Ecebs Ltd and Atmel Corporation) to enhance their smart cards with a 16 megabytes external FLASH memory [DHo04]. The smart card is compliant with ISO 7816, but the memory chip is larger. The additional memory module is also linked to the secure chip by a bus, and its content is encrypted. Recently, Gemplus demonstrated the *Secure hUge Memory On-card* (SUMO) smart cards [Gem], providing 224 megabytes of stable memory. Memory is dispatched on the plastic cards in 7 blocs of 11mm x 12mm memory modules (Hitachi FLASH product) linked together by a bus. Efforts have been made to make these memory blocs flexible enough to be compliant with traditional cards torsions requirements as well as to provide a high data transmission rate (2.5 megabytes/s).

In the same spirit, a new trend aims at protecting memory cards with smart cards, plug-able into the SIM card slot of a cellular phone [Ren04]. Also, the *MObile PASSport* (MOPASS) consortium [MOP04], composed by the main smart card and chip manufacturers (*e.g.*, Gemplus, Hitachi, Sharp, NEC, etc.) establishes the standard for FLASH memory cards combined with smart cards. In all cases, the smart card is viewed as the intelligence providing its high level of security to the added memory. Readers are also extended to provide a dual port [Atm02] to plug simultaneously a smart card and its corresponding memory card. Note that another approach might be envisioned to solve the internal storage limitation, which relies on un-portable (but connected) resources like the Internet [BiW98, Big99]. This approach enables on-line activities and requires cryptographic techniques. This topic will be further discussed in Chapter 5.

Recent improvement made to smart cards throughput (see Section 1.1.2) broaden smart cards application domain to on the fly data flow processing (*e.g.*, encoding and/or decoding). In addition, the integration of multi-applications operating systems drastically reduces the time to market (several days instead of several months to deploy new applications). According to Gemplus, further progress will occur to make smart cards more secure, powerful and ergonomic: multithreading will be provided on smart cards (long term issue as shown in [RES03]), battery will be added [Vac03], small keyboards and screens might be provided on the plastic substrate of the card. All these enhancements improve the security abilities provided by smart cards and/or widen their application domains.

### *1.1.7 Impact of Smart Cards Computing Environment on Embedded Database Components*

We describe here the impact of the smart card's environment on the design of embedded database components (called DB components in the following).

First, smart cards are *highly secure*. On-board hardware security makes them the ideal storage support for private data. The DB components must contribute to the data security by

providing access right management and a view mechanism that allows complex view definitions (*i.e.,* supporting data composition and aggregation). The DB components code must not present security holes due to the use of sophisticated algorithms[16].

Second, smart card is undoubtedly the most *portable personal computer* (the wallet computer). The data located on the smart card is thus highly available. It is also highly vulnerable since the smart card can be lost, stolen or accidentally destroyed. The main consequence is that durability cannot be enforced locally.

Third, the *tiny die size* embedded in a smart card is fixed to 25 mm² in the ISO standard [IOSp1], which pushes for more integration. This limited size is due to security considerations (to minimize the risk of physical attack [TrC]) and practical constraints (*e.g.*, the chip should not break when the smart card is flexed). Despite the foreseen increase in storage capacity, the smart card will remain the lightest representative of personal computers for a long time. This means that specific storage models and execution techniques must be devised to minimize the volume of persistent data (*i.e.*, the database) and to limit the consumption of memory during execution. In addition, the embedded DB functionalities must be carefully selected and their implementation must be as light as possible, to allow for the largest onboard database.

Finally, smart cards are non autonomous. Compared to other computers, the smart card has no independent power supply, thereby precluding disconnected and asynchronous processing. Thus, all transactions must be completed while the card is inserted in a terminal (unlike PDA, write operations cannot be cached in RAM and reported on stable storage asynchronously).

## 1.2 Smart Card Hardware Resources

This section analyzes the current smart card hardware resources especially in terms of embedded memory, since this aspect can have a strong impact on data management. Then, we present the evolution of smart card architecture and highlight the hardware constraint that will remain in the future.

### *1.2.1 Current Architecture*

Current advanced smart cards include, in a monolithic chip, a 32-bit RISC processor at about 50 MHz, several memory modules (about 96 kilobytes of ROM, 4 kilobytes of static RAM and 64 kilobytes of EEPROM), and security hardware modules (random number generator, cryptographic co-processor, etc.). The ROM is used to store the operating system, fixed data and standard routines. The RAM is used as working memory to compute results and only a few bytes remain for programs since the operating system is using it almost entirely.

[16] Most security holes are the results of software bugs [ScS99].

EEPROM is used to store persistent information, and holds data and downloaded programs (in case of multi-application cards). Figure 6 pictures the current smart card architecture.

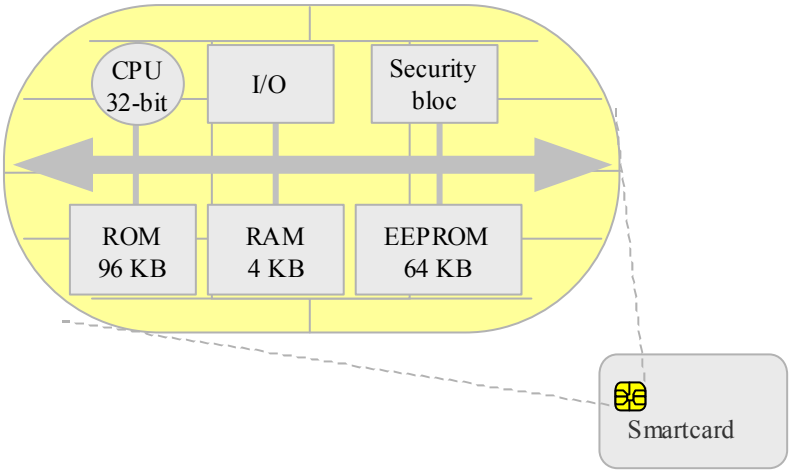



**Figure 6** *Smart Card's Architecture.*

The smart card's internal resource balance is very unique. Indeed, the on-board processing power, calibrated to sustain cryptographic computations and enforce smart cards security properties, has a disproportionate power compared to the small amount of embedded data. In addition, stable memory (EEPROM) has very fast read time comparable to RAM (60-100 ns/word), but a dramatically slow write time, about 10 ms per word. Finally, only a few bytes of free RAM are available for the embedded applications. This unusual computing environment compare to more traditional ones commonly used by DBMSs (*i.e.*, disk based DB and main memory DB, see Chapter 2, Section 2.1) can be summarized by the following properties:

*(P1) High processing power relatively to small data amount.*

*(P2) Very tight RAM relatively to the amount of data to process.*

*(P3) Fast reads and slow writes to stable memory.*

These properties undoubtedly call for a new type of DBMS design. However, one can wonder whether these three properties will remain valid considering traditional hardware evolution and Moore's law[17].

The evolution of smart card chips is closely linked to applications requirements. The benefits of multi-application cards and the emerging large-scale government projects (*e.g.*, medical folders) pushes for larger amount of embedded memory and increasing level of

[17] Gordon Moore observes in 1965 that the computer's power doubles each 18 months at constant price. This empirical rule is still verified today.

security (*e.g.*, HIPAA security requirements [OCR02] for personal information in the US). This translates into more powerful processors and cryptographic co-processors. The aforementioned chip properties are successively analyzed with respect to this evolution.

### *1.2.2 Processing Power*

During the last ten years, embedded processors improved from the first 8-bit generation running at 2 MHz to the current generation of smart cards based on 32-bit processor running up to 100 MHz [Gem04b] with an added cryptographic co-processor. However, one can think that such powerful processing power is already enough to guarantee smart cards internal security, and that further improvements on that front are not mandatory. Moreover, users might prefer less functionality at a lower cost [CRn04]. We give here three opposing arguments and we strongly believe that property *P1* will remain valid in the future.

Currently, many smart cards' applications have evolved to process data flows using cryptographic functionalities [Pot04] (*e.g.*, pay TV based on chips decryption and embedded profile). The rapid evolution of smart card throughput (*e.g.*, high delivery contactless cards, USB cards) makes such applications feasible. Since the hardware improvements are driven by the application's requirements, this will certainly lead to more powerful processing power in future smart cards.

Moreover, many research efforts and manufacturers investments focus on the development of multi-application and multi-threaded operating systems necessitating even more CPU power.

Finally, the smart card cost is mainly driven by the amount of embedded memory (RAM plus EEPROM), but not so much by the amount of processing power [Jea01]. Since smart cards are massively produced, such a consideration favors enhancing the processing power rather than the amount of memory.

To conclude, we consider in this study that property *P1* will remain valid because of future application's requirements and smart cards improvements. This property should be taken in consideration while developing embedded DB components.

### *1.2.3 Amount of Working Memory*

Currently, smart cards hold a few kilobytes of static RAM storing temporary data (see Table 4 for the most recent smart cards). This amount of memory is almost entirely consumed by the operating system and/or by the virtual machine. Only few bytes remain for the embedded application(s). We examine here whether the small amount of working memory assumption could be reconsidered in a near future, in particular since smart cards are becoming multi-application.

The computing power of lightweight devices globally evolves according to Moore's

law, but the gap between the RAM available to the applications and the other resources (*e.g.*, CPU speed and stable storage capacity) will keep on increasing for three reasons:

The main reason is the big push by manufacturers to reduce the hardware resources to their minimum in order to save production costs on large-scale markets [SIA02]. Since the cost of a chip is closely linked to its surface, the relative cell size of static RAM (see Table 2) makes it a critical component in the overall cost. For instance, Table 3 presents the on-chip surface of the different embedded components in a representative smart card with 4 kilobytes of RAM and a total of 164 kilobytes of memory. In this configuration 2.4% of the total memory occupies 15% of the chip surface.

| *Memory* | *Size factor* |
|---|---|
| ROM | 1 |
| FLASH | 1 |
| EEPROM | 4 |
| FeRAM | 16 |
| SRAM | 16 |

**Table 2** *Memory Cell Relative Size.*

Second, there is no clear motivation to have a large RAM area because (i) the smart card is not autonomous, thus precluding delaying costly operations (like writes) to be performed asynchronously; (ii) the available non volatile electronic memory does not need traditional memory hierarchy to cache the data while reading; and (iii) the RAM competes with other components on the same silicon die [NRC01, GDM98]; the more RAM, the less stable storage, and therefore the less embedded data. As a consequence, manufacturers of smart cards privilege stable storage to the detriment of RAM. This later is strictly calibrated to hold the execution stack required by the on-board programs.

| *Component* | *Power* | *Occupied surface (%)* |
|---|---|---|
| CPU | 32-bit / 50 MHz | 10 |
| ROM | 96 KB | 20 |
| RAM | 4 KB | 15 |
| EEPROM | 64 KB | 45 |
| Other | | 10 |

**Table 3** *On-chip occupation surface.*

Finally, even considering that RAM amount may increase for multi-application and multi-threaded smart cards, the more complex operating system and virtual machines, as well as the on-board applications will also have increasing RAM needs.

Thus, we consider that property *P2* will remain valid as long as working and stable memory reside on the same chip and as long as the SRAM cell size is the most costly part because of the surface it occupies. Since these are long-term challenges, property *P2* should also be taken into account while designing embedded DB components.

### *1.2.4 Stable Storage Asymmetry*

Let us now consider how hardware advances can impact the memory size limit, and its read/write performance asymmetry. Current smart cards rely on a well established and slightly out-of-date hardware technology (0.35 micron) in order to minimize production costs (less than five dollars per smart card) and increase security [ScS99]. Furthermore, up to now, there was not a real need for large memory in smart card applications like holder's identification. The market pressure generated by emerging applications with large storage needs led to a rapid increase of the smart card storage capacity. This evolution is however constrained by the smart card tiny die size fixed to 25 mm² by the ISO standard [ISOp1], which pushes for more integration.

Today's most popular stable memory for smart cards is EEPROM, which provides fast read (comparable to RAM reads), with random access to the data, but very slow writes (several millisecond per byte). This asymmetry between the time to read and write (property *P3*) must be taken into account while designing DB components. However, the validity of this property in the future is not so obvious since it depends on future embedded memory technology.

| *Manufacturer* | *Product's name* | *ROM* | *EEPROM* | *RAM* |
|---|---|---|---|---|
| Atmel | AT91SC25672RC | 256 KB | 72 KB | 10 KB |
| Renesas | AE57C | 320 KB | 132 KB | 10 KB |
| ST Micro. | ST22WL128 | 224 KB | 128 KB | 8 KB |

**Table 4** *Current Smart Cards Products.*

Current efforts to increase the amount of stable memory benefit from 0,24 micron technologies or less, which allows having up to 132 kilobytes of EEPROM [Ren03] (see Table 4).

| *Manufacturer* | *Product's name* | *ROM* | *FLASH* | *RAM* |
|---|---|---|---|---|
| Infineon | SLE88CFX4002P | 240 KB | 400 KB | 16 KB |
| Sharp | Sharp1MB | 8 KB | 1024 KB | 4 KB |

**Table 5** *Next Generation of Cards (coarse-grain FLASH).*

At the same time, several manufacturers are attempting to integrate denser memory on chip. In current prototypes, FLASH[18] memory replaces both ROM and EEPROM, since it is more compact than EEPROM and thus represents a good candidate for high capacity smart cards [Gem99]. Several kind of FLASH memory can be used and have different characteristics[19]. Infineon [Inf04] and Sharp are proposing two smart card prototypes (see

[18] Flash memory is a later form of EEPROM. There is a convention to reserve the term EEPROM to byte-wise updateable memory compared to block-wise updateable flash memory.

[19] Classical FLASH components are based on NOR or NAND technology. We deliberately avoid these names in the following since the characteristics of the FLASH memory given by the smart

Table 5) using *coarse-grain* FLASH memory. In these prototypes, programs can be executed directly in FLASH since it provides *eXecute In Place* (XIP). However, while read and write times in coarse-grain FLASH are comparable to EEPROM, flash banks need to be erased before writing them. Thus, the cost of an update to coarse-grain FLASH is very high since it requires reset the entire block (*e.g.*, 64 kilobytes) touched by the update before rewriting it with the modified data. This makes coarse-grain FLASH memory inappropriate for data-centric applications. However, using this type of memory instead of traditional ROM and EEPROM associations allows having up to 1 megabyte of stable storage available for the operating system, the virtual machine and the applications.

| *Manufacturer* | *Product's name* | *ROM* | *FLASH* | *Page-FLASH/EEPROM* | *RAM* |
|---|---|---|---|---|---|
| ST Micro. | ST22FJ1M | 128 KB | 768 KB | 256 KB (page-FLASH) | 16 |
| Philips | P9ST024 | 512 KB | 256 KB | 256 KB (EEPROM) | 16 |
| Philips | P9SC648 | No | 512 KB | 142 KB (EEPROM) | 16 |
| EM Micro. | EMTCG256 | 128 KB | 64 KB | 64 KB (EEPROM) | 4 |

**Table 6** *Next Generation of Cards (fine-grain FLASH or EEPROM).*

Another approach to provide more memory for multi-application smart cards, without proscribing updateable data centric applications, is to combine traditional coarse grain FLASH with *fine-grain* updateable FLASH or EEPROM (see Table 6). In this case, the coarse-grain FLASH holds the programs (operating system, virtual machine and downloaded applications) and stands for ROM, while EEPROM or fine-grain FLASH memory is used as the data store [STM04]. ST Microelectronics provides a very fine-grain FLASH memory (called page-FLASH [STm02]) allowing updating each word individually. Notice that the asymmetry between reads and writes still exists (property *P3*) in this case (see Table 7).

| *Memory type* | *EEPROM* | *Page-FLASH* | *FLASH* | *FeRAM* |
|---|---|---|---|---|
| Read time (/word) | 60 to 150 ns | 70 to 200 ns | 70 to 200 ns | 150 to 200 ns |
| Write time | 10 ms / word | 5 to 10 µs / word | 5 to 10 µs / word | 150 to 200 ns |
| Erase time | None | 10 ms / word | 0.6 to 5 s / bloc(**) | None |
| Lifetime (*) | $10^5$ write cycles | $10^5$ erase cycles | $10^5$ erase cycles | $10^{10}$ to $10^{12}$ |

* A memory cell can be overwritten a finite number of time.
** Traditional block size is 4 to 128 KB

**Table 7** *Performance of Stable Memory.*

Another possible alternative to the EEPROM is *Ferro-electric RAM* (FeRAM) [Dip97] (see Table 7 for performance comparisons). FeRAM is undoubtedly an interesting option for smart cards [Fuj02, Fuj03] as read and write times are both fast. Although its theoretical foundation was set in the early 50s, FeRAM is just emerging as an industrial solution. Therefore, FeRAM is expensive, less secure than EEPROM or FLASH, and its integration with traditional technologies (such as CPUs) remains an open issue. Thus FeRAM could be

card manufacturers are not consistent with either NOR or NAND one.

considered a serious alternative only in a distant future [Dip97, RES03]. For the time being, only few kilobytes of FeRAM are embedded into smart card prototypes for applications with high speed and low power requirements [Nik03].

| *Memory type* | *Relative cell size* | *read* | *Write* | *grain* |
|---|---|---|---|---|
| ROM | 1 | | | |
| MRAM | 1 – 3 | <100 ns | < 100 ns | Bit |
| PCM / OUM | 0.8 – 2 | <100 ns | < 100 ns | Bit |
| Millipede | 0,1 | 10µS | 50 µS | Bit |
| Nanocrystal | 1 - 2 | 10 ns | 10 ns | Bit |
| NEMS | 0,5 - 1 | 10 ns | 10 ns | Bit |
| SET | 0,5 - 1 | 10 ns | 100 ns | Bit |

**Table 8** *Long Term Memory Alternatives.*

The hardware researchers in memory technologies aim at developing the perfect alternative, meaning a highly compact non-volatile memory providing fast read/write operations with fine grain access. Table 8 lists some of the studied technologies. However, for many smart card manufacturers, integration of these kinds of memory into smart cards is an even more distant future [RES03]. This document clearly does not address these alternatives.

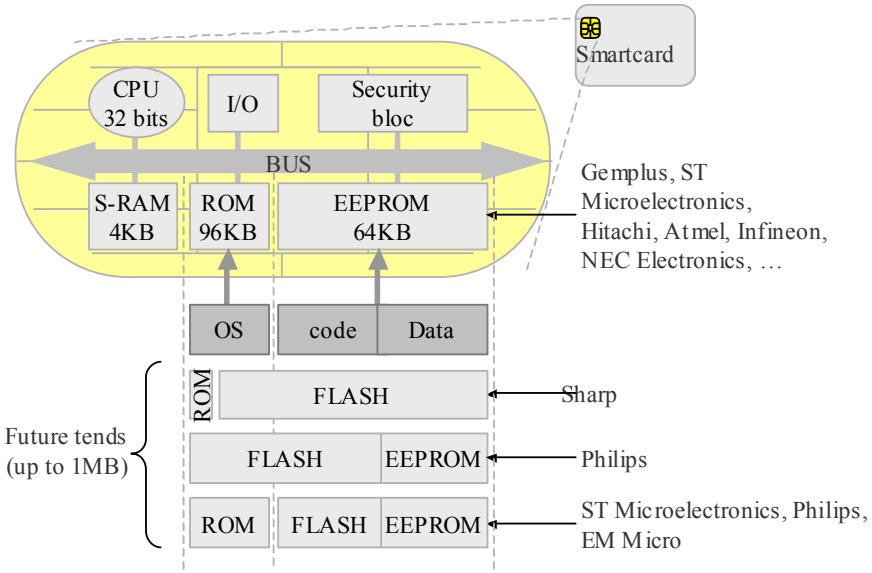



**Figure 7** *Evolution of Smart Card Prototypes Architecture.*

To summarize, the amount of embedded memory will reach 1 megabyte for the next generation of smart card (see Figure 7). Even considering FLASH memory, updatable data will remain in fine-grain FLASH and programs in traditional coarse grain FLASH, which also complies with property *P3*. Others embedded memory like FeRAM could modify this assumption but we consider this to be a long term alternative. We thus consider that property *P3* must be considered when designing embedded DB components.

# 2 DBMS Techniques

This section gives some necessary background on Database Management System, and focuses specifically on query processing techniques. Traditional relational database concepts, such as data representation, index structure, execution model, operator algorithms and optimization techniques are outlined. Since smart card stable storage is electronic memory and provides the read speed and direct access granularity of main memories, we present some relevant techniques from *Main Memory DBMS* (MMDBMS) literature. For more details on database principles, we refer to the following books [KoS91, ElN94, AHV95, RaG02, GUW02].

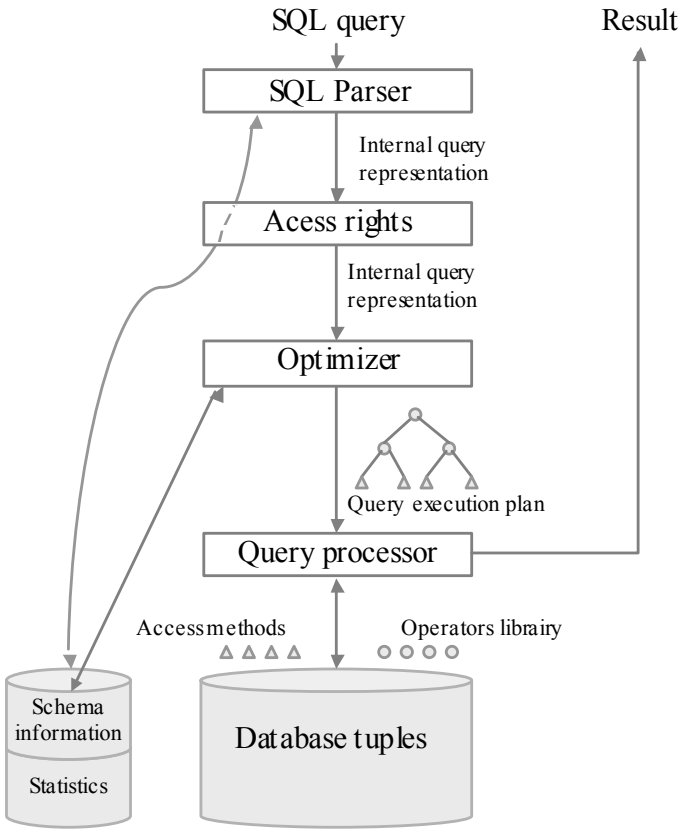



**Figure 8** *Query Processing Architecture.*

Figure 8 sketches the query processing architecture of typical relational database systems. Users authenticate to the DBMS and can formulate their queries using query languages such as the *Structured Query Language* (SQL). SQL is a declarative language, so the users only have to specify the wanted result without stipulating the retrieval method; the DBMS engine is in charge of finding the 'best' way to provide the requested information.

Briefly, query processing consists of the following three steps. First, the SQL query is parsed and checked for syntax and semantic correctness using database schema information.

The query is then translated into an internal representation to be transmitted to the database kernel.

Second, the query is checked with respect to the user access rights and possibly rejected if unauthorized data is requested. If the query is approved, it is translated by the optimizer into an efficient executable program using operators available in the operator library as well as appropriate data access methods. This program called a *query execution plan* (QEP), consists of an efficient (ideally optimal) alternative to retrieve the result. Note that several QEP can be found to retrieve the same result, with different operators ordering, different operators algorithms (*e.g.*, based on sequential scan, sorting, hashing), and different access methods (*i.e.*, indices). The QEP chosen by the optimizer is the one that minimizes the estimated processing cost, using cost functions and database statistics (note that depending on the application context, the goal of the optimizer is to minimize the execution time, the resource consumption, the power consumption, or the first result production time, etc.).

Finally, the execution engine executes the QEP; the result is built and delivered.

This section explains the query processing architecture in a bottom-up approach. It describes existing data representations, browses access methods and indexations techniques (in particular those for main memory resident data), presents the operator library and the execution model. Finally we make a quick description of the query optimization process.

## 2.1 Impact of Data Storage Medium on Query Processing

In the literature, two major media are considered to store the database data: magnetic disks and main memory. Since they provide different properties, the corresponding query processing strategies differ drastically. So, before going into query processing details, this introductive section presents the influence of the database storage medium on database query processing.

In traditional DBMS, data resides on magnetic disks. Random access on magnetic disks is very costly, due to seek time (positioning the arm) and rotational delay (positioning on the required sector), in the range of some milliseconds (*e.g.*, 5 ms). Once positioned, the data can be transferred at a high throughput (*e.g.*, 75 megabytes/s) thus leading to cheap sequential access. Due to these properties, efficiency is obtained by reducing the disk accesses to the minimum, and by choosing an appropriate policy to cache data in main memory [Sac87].

In *Main Memory DBMS* (MMDBMS), the data resides in main memory and thus can be randomly and rapidly accessed. Thus, MMDBMS storage models take advantage of these characteristics and are based on efficient pointer following. MMDBMS query processing techniques try to minimize memory copies and optimize the processor cache usage. We should remark that traditional DBMS also requires memory accesses optimizations since

sequential disk accesses can be overlapped[20] by main memory processing [MBK00a, BMK99, AWH+99].

Since some non-volatile electronic memory shares some properties with disk (*e.g.*, dissymmetry between random and sequential accesses for NAND FLASH) and also with RAM main memory (*e.g.*, fast and direct access for EEPROM), we explore, in the following, query processing techniques in both contexts.

For further information about main memory techniques, see the survey from [GaS92], and documentation of MMDBMS prototypes and products including *Monet* [BoK95], *Dali* [JLR+94], *PRISMA/DB* [ABF+92], *Datablitz* [BBG+98, BBG+99], *TimesTen* [Ten99], *Xmas* [PKK+98], etc.

## 2.2 Data Layouts

Traditional data placement for relational data on-disk is the *N-ary Storage Model* (NSM) which stores records contiguously. Applied to MMDBMS, NSM offers high intra-tuple locality, however query operators often access only a sub-part of each tuple damaging cache performance. This storage model is the most commonly used in today's database systems. Two storage alternatives to NSM have been developed as shown in Figure 9.

[CoK85] introduces the *Decomposition Storage Model* (DSM), which can be used as well in MMDBMS [BoK95]. DSM stores each column of a relational table in a separate array of two-field records (ID, value) in memory. However, reconstructing the tuples induces joins between the different arrays, or fixed-size couples (domain ID, value) as implemented in certain MMDBMS [BoK95].

Note also that a recent data layout called *Partition Attributes Across*[21] (PAX) combines NSM and DSM, to optimize the cache behavior [ADH+01, ADH02] and tuples reconstruction. For a given relation, PAX stores blocks of tuples sequentially. However, within each bloc, PAX groups all the values of a particular attribute together on a sub-array. Thus, PAX fully utilizes the cache resources, because on each miss a number of the same attribute's values are loaded into the cache together. At the same time, all parts of the tuple belong to the same bloc. The reconstruction requires a join among sub-arrays, which incurs minimal cost because it does not have to look beyond the bloc.

Efficient space utilization, necessary to hold the active database entirely in main memory, is not addressed by these storage models, initially designed for disk based DBMS. The compactness goal is easily achieved in main memory storing pointers rather than values in data structures [PTV90, GaS92]. The concept of a domain storing distinct data values has

[20] I/O bandwidth follows Moore's law, I/O latency does not.

[21] Although this is a disk oriented storage model, we present it here since its objective is also improving main memory processing.

been used for its compactness since the early 70's, storing only pointers to domain values (or domain IDs) in place of values in each column [MiS83, BRS82, Mis82, AHK85]. Indeed, this storage model offers the benefit of space saving in the presence of duplicate values, simplifies the handling of variable-length fields and provides good cache performance (this last advantage explains its renewed interest). Note also that domain IDs comparisons can be used for equality tests, rather than the original values. Since values in the domain can be kept ordered [RaR99], both equality and inequality tests can be directly processed on domain IDs. Keeping values in order incurs an extra cost, however in many contexts[22] the domain data is infrequently updated, which makes it an acceptable solution.

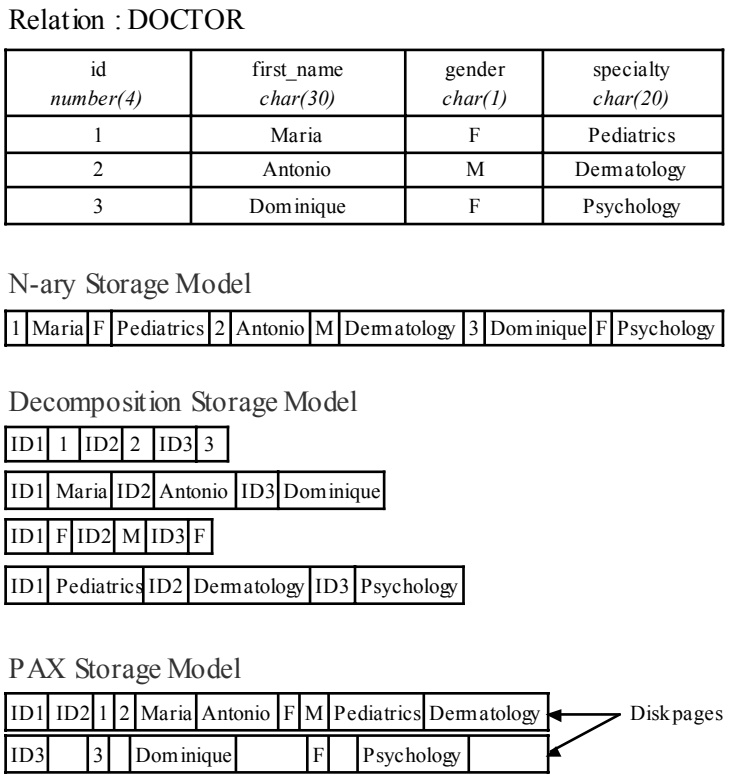



**Figure 9** *Database Storage Models.*

## 2.3 Access Methods

Clearly, access methods designed for disk-based databases and MMDBMS are radically different since efficient data processing for memory resident data aims at minimizing computations, memory access time (and also memory space in some cases) instead of disk I/O's. To achieve these goals, MMDBMS index structures exploit the fast and direct data access capability of main memory [DKO+84] and reduce the space overhead. Thus, indices can store pointers rather than the data [GaS92] and accelerating structures can even be embedded within the data to achieve both compactness and efficiency [PTV90]. We first

[22] Append contexts like historical information storage.

present a representative subset of traditional sort or hash based access structures, and then other interesting index structures, which may be useful in our context.

### *2.3.1 Sort and Hash Based Index Structures*

We concentrate here on traditional access methods, which have been investigated in the literature for either disk oriented or memory resident databases. We can distinguish two different categories of access structures. The first category preserves some ordering in the data, while the second randomizes the data. Here we look at representative access structures of each category taken from literature - for order preserving structures: B-Trees, AVL-Trees, and T-Trees; - for the randomizing class: Chained Bucket Hashing, Linear Hashing and Extensible Hashing,. We describe these structures and illustrate their usage giving the search cost of the tuples that are sharing one particular value in term of values comparisons and pointer followings. This cost analysis does not take into consideration the possible disk access and only stands for the sake of understanding.

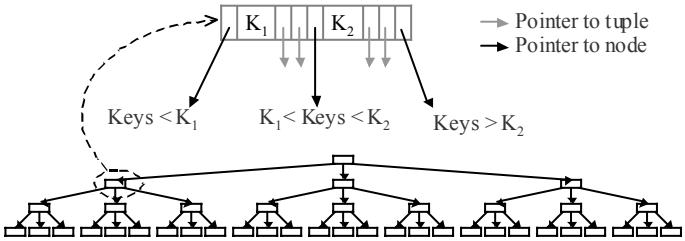



**Figure 10** *B-Tree.*

*B-Tree* is a balanced tree based structure (see Figure 10). An *m* order B-Tree contains *p* sorted keys in each node, with $m \leq p \leq 2m$ for intermediate nodes, and $1 \leq p \leq 2m$ for the root. Each node has *p+1* children, the $i^{th}$ child owning key values between those of the $(i-1)^{th}$ and $i^{th}$ one of its father. The cost of searching tuples sharing a particular key value can be computed as follow: Suppose a B-Tree holding *k* keys with an average of *p* keys per node. $log_2(p)$ key comparisons occur in each node due to binary search, and $log_{p+1}((k)$ nodes are involved in the search (since it is the height of the tree). So, the search costs at most $log_2(p)*log_{p+1}(k)$ key comparisons and $log_{p+1}(k)$ pointers followings (we consider that the search ends when we get the pointer to the first qualified tuple). As an example, with *1000* keys (*i.e.*, *k=1000*) stored by *16* (*i.e.*, *p=16*) in average in each node, the search induces *12* key comparisons and *3* pointer followings to reach the references to qualified tuples. The B+ Tree variant of the B-Tree duplicates all the keys in the leaves, which are chained to provide fast full scan of the keys (and possibly tuples) in the sorted order.

*AVL-Trees* are binary search tree. They are balanced (similarly to B-Trees; AVL-Trees are however deeper since they are binary), and might provide poor storage utilization (two sub-tree pointers for each stored key). Note that B-trees can be much more space consuming since nodes are often half empty. AVL-Trees have been transformed for main memory into

*T-Trees* [LeC86a, LeC86b], which store multiple keys in each node (see Figure 11). [BMR01] states that all main memory database products implement the T-Tree index structure. As for AVL-Trees, index traversal is cheap since it only induces one comparison with the first (and possibly the last) key of each node and a binary search in the last node only. Considering the previous example, with *1000* keys stored by *16* in average in each node, the search induces an average of *9* key comparisons and *6* pointer followings to reach references to qualified tuples.

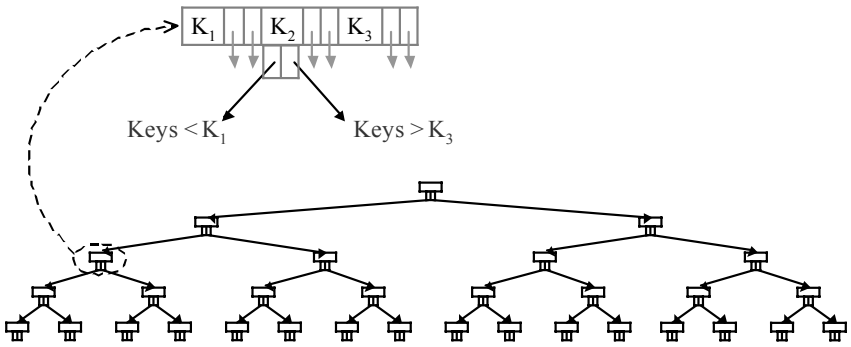
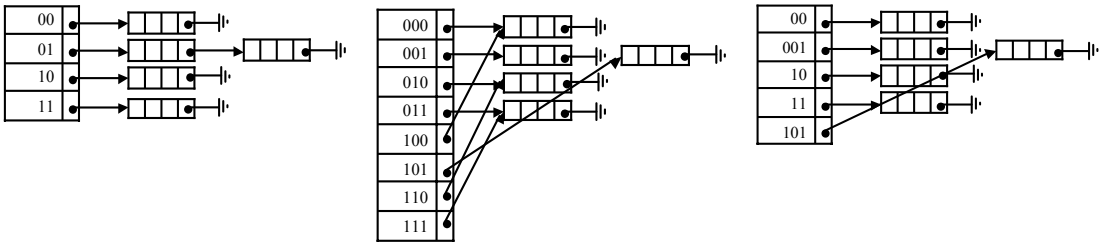



**Figure 11** *T-Tree.*

*Hash-based indices* [AnP92, PKK+98], pictured on Figure 12, can be either static (the size of the hash table must be known in advance, *e.g.*, *Chained Bucket Hashing* [Knut73]), or dynamic (*e.g.*, *Extensible Hashing* [FNP+79], *Linear Hashing* [Lit80]). It induces low storage utilization for good performance. We concentrate here on dynamic structure. *Extensible Hashing* [FNP+79] uses a dynamic hash table. Each hash bucket containing pointers to tuples corresponds to one entry on the hash table. When a hash bucket overflows, it splits and the hash table grows in powers of two. An additional bit resulting from the hash function is used to identify the buckets. The main problem of this index structure is the possible growth of the hash table in case of successive collisions. To solve this problem, *Linear Hashing* [Lit80] provides that the hash table can grow linearly and the buckets can split in a predefined order. Thus, each overflow causes only one additional entry in the hash table taking into account an additional bit from the hash function. This technique can be modified towards memory saving. Such structures induce a hash computation and key comparison in a single bucket to retrieve the qualified tuples. However, they are useless in case of range search.

Recent studies show that cache conscious index strategies outperform conventional indices. We refer the reader to [RaR99, RaR00, ADH+01] for cache conscious index strategies and to [CKJ+98, ADH02] for cache conscious data placement.




*(a) Chained Bucket Hashing.* *(b) Extensible Hashing.* *(c) Linear Hashing.*

**Figure 12** *Hash Based Structures.*

### *2.3.2 Other Index Structures*

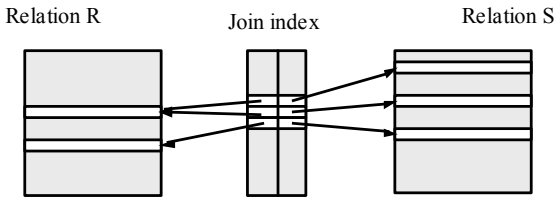



**Figure 13** *Join Indices.*

Valduriez [Val87] proposed in 1987 the *Join indices* for traditional disk-based DBMS. A Join index is an additional binary relation that links two relations on the join attributes, thus allowing to speed-up the relational join operations (see Figure 13). In a MMDBMS context, it constitutes a very efficient accelerating structure since it materializes in memory the expected result of the join operation.

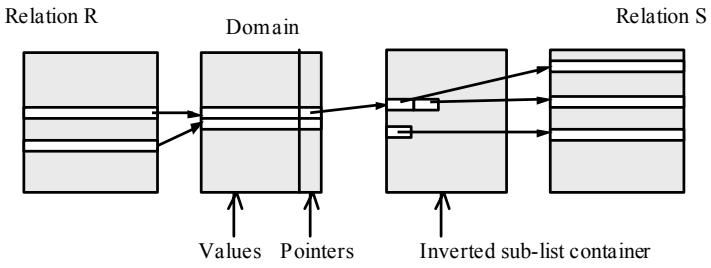



**Figure 14** *DBGraph Possible Implementation for Efficient Joins.*

[PTV90] describes the *DBGraph* storage model, specifically designed for the MMDBMS context. In the DBGraph model, the database is represented as a unique graph-based data structure (basically a graph linking tuples-vertices to values-vertices). Implementation of the DBGraph storage model can be done efficiently by exploiting the direct access capability of the main memory systems. Figure 14 illustrates a possible implementation of the DBGraph storage model. For example, to join two relations *R* and *S* over a common attribute *a*, we scan the smaller relation (*e.g.*, *R*) and for each tuple we

follow the *a* pointer to reach its value. From this value, we follow the pointers referencing all matching *S* tuples, etc…

## 2.4 Operators Library and Execution Model

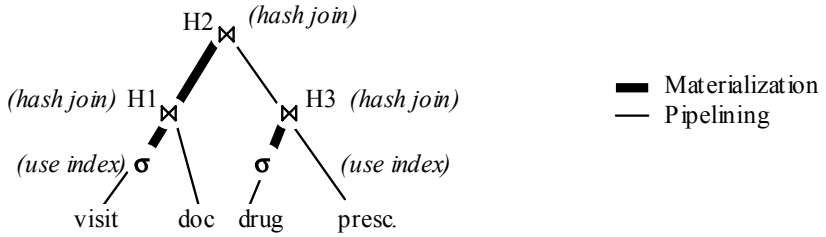
The index methods presented above allow efficient access to the data. This section focuses on the relational operator implementations called *Operators Library* and on the way they are combined in a *Query Execution Plan*. Finally we detail the more widespread execution model: the *Iterator Model*.

To process SQL queries on the database, several efficient implementations of each relational algebra operators must coexist in the DBMS. For instance, a relational selection can either be performed sequentially testing the selection predicate on all the tuples of the accessed relation or using one of the existing indices. Thus, a simple operator like the selection one may be implemented in several ways to allow the optimizer to choose the best operator for a given query in a given context (see next section). This is true for all relational operators. Basically, there are three possible ways to perform relational operations: (i) use an existing index, (ii) create a temporary index (hash or sort based), or (iii) sequentially. Generally, the relational operation (*e.g.*, selection) is called a *logical operator* while the different implementations are called *physical operators*. The operator library is a library of physical implementation of logical relational operators; the larger it is, the more optimization opportunities are available for the query optimizer. We refer the reader to [RaG02] for details on the different physical implementations of relational operators for more information.

A *Query Execution Plan (QEP)* or query evaluation plan [RaG02, Gra93] is a tree representation of a relational algebra expression corresponding to one possible evaluation plan for a given query. We represent the QEPs using the traditional tree representation: Tree's nodes represent relational algebra operators while annotations indicate the index that can be used and the operator algorithm. Edges stand for data flows. These edges can be either blocking or pipelining edges, indicating whether the operator result must be, or not, entirely computed before being consumed by the next operator. Figure 15 shows a possible QEP for a sample query on a simplified healthcare database schema. We use the [ScD90] convention, which materializes the left entry of binary operators.

For the sake of explanation[23], we quickly review the hash join algorithm commonly used in RDBMS. The hash join proceeds in two phases: in the *build* phase, the tuples of the left input are inserted in a hash table using a hash function on the joining attribute. In the *probe* phase, tuples of the right input are hashed on the joining attribute and *probed* on the corresponding entry in the hash table.

[23] The hash join is described here to ease the understanding of the Iterator model presented below. Indeed, Chapter 3 requires a good comprehension of the Iterator model.




**Sample healthcare schema:**

*Doctor (*DocId, name, specialty, …*)*
*Visit (*VisId, DocId, date, diagnostic, …*)*
*Prescription (*VisId, DrugId, qty, …*)*
*Drug (*DrugId, name, type, …*)*

**Query:**

```
SELECT D.name
FROM Doctor D, Visit V, Prescription P, Drug D
WHERE D.DocId = V. DocId AND V.VisId = P.VisId
    AND P.DrugId = D.DrugId AND D.type = 'ANTIBITIC'
    AND V.date > 01-jan-2000;
```

**Figure 15** *A Sample QEP.*

In Figure 15, the *visit* relation will be accessed through the index on the visit date. Tuples matching the predicate will be inserted in hash table H1. Then tuples from the *doc* relation will be hashed and probed on H1. Matching tuples will be inserted in hash table H2. Index based selection of drugs will then begin and selected tuples will be inserted in H3. Finally prescriptions will be scanned and each tuple will be successively probed on H3 and (if it matches) on H2.

In fact, the query processing is generally triggered in a demand driven manner, using the *Iterator Model* [Gra90, Gra93] which provides a simple, but powerful way to process a QEP and synchronize its execution. With the Iterator model, each operator has an Iterator interface composed of the *Open*, *Next* and *Close* functions. *Open* initializes the operator. It opens the operators (or relations) entries, and allocates the required data structures for the execution (*e.g.*, a hash table). *Next* produces an output tuple or *End Of Stream* (EOS), which indicates that all the results have been produced. And *Close* terminates the execution. It closes the input relations or operators, and frees the temporary structures used during the execution.

The concept of Iterator makes the QEP evaluation very simple. Its initialization is carried out in a recursive way by opening the root operator of the QEP. Next calls on the root operator deliver one by one all result tuples. The query evaluation terminates when an EOS is received, by closing the root of the QEP. This last call recursively closes all the remaining operators.

| **HashJoin.Open** | **HashJoin.Next** | **HashJoin.Close** |
|---|---|---|
| **Allocate** Hash table<br>**Open** the left input<br>Build hash table calling **Next** on the left input until EOS<br>**Close** the left input<br>**Open** the right input | Call **Next** on the right input until a match with the hash table is found (or EOS) | **Close** the right input<br>**Deallocate** the hash table |

**Figure 16** *Hash Join Iterator [Gra93].*

Figure 17 gives the open, next and close implementations of the hash join in pseudo-code. The build phase is implemented in the open function, while the probe phase is done, in pipeline using the next function. Figure 16 illustrates the successive function calls while processing the QEP of Figure 17.

| **Join2.Open** | **Join2.Next** | **Join2.Close** |
|---|---|---|
| **Allocate** Hash table **H2**<br>**Join1.Open**<br>**Allocate** Hash table **H1**<br>**IndexScan.Open** (visit)<br>**Open** the index<br>**IndexScan.Next**<br>Retrieve a visit<br>Insert in H1<br>**IndexScan.Open**<br>Retrieve a visit<br>Insert in H1<br>**IndexScan.Open**<br>Retrieve a visit<br>Insert in H1<br>**IndexScan.Open**<br>**EOS**<br>**IndexScan.Close** (visit)<br>**Scan.open** (doc)<br>Etc. | **Join3.Next**<br>**Scan.Next** (presc)<br>Retrieve a prescription<br>Probe H3 ➔ KO<br>**Scan.Next** (presc)<br>Retrieve a prescription<br>Probe H3 ➔ OK<br>Probe H2 ➔ KO<br>**Join3.Next**<br>**Scan.Next** (presc)<br>Retrieve a prescription<br>Probe H3 ➔ OK<br>Probe H2 ➔ OK | **Join3.Close**<br>**Scan.Close**<br>Close relation presc<br>**Deallocate H3**<br>**Deallocate H2** |

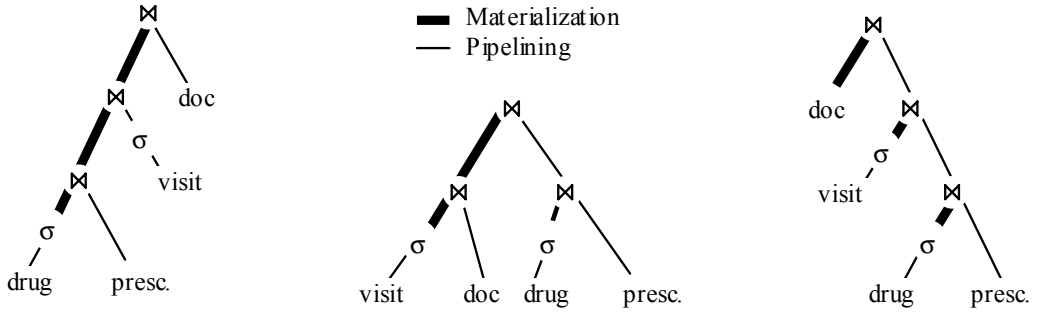
**Figure 17** *Sample QEP Processing Using the Iterator Model.*

## 2.5 Optimization

The goal of the optimizer is to find out the QEP with the lowest cost; , depending on the application context, the cost is expressed in terms of response time, resource consumption, time to produce the first tuples, etc. Query optimization involves two tasks (which can be interleaved): the *QEP generation* and the *QEP cost estimation*. They are presented below.

The QEP generation consists of generating several alternatives QEPs. Generally, only a subset of all possible QEPs are examined by the query optimize because the number of equivalent plan can be very large. The size of the *search space* (*i.e.*, the space of equivalent QEPs) depends on the number of operators in the QEP (typically, the number of binary operators), the number of algebraic equivalence rules (*e.g.*, joins operations are commutative), the number of physical operators available in the operator library and on the considered *tree shape* of the QEP.

Indeed, the optimizer can consider several tree shapes: linear left or right deep tree [ScD90] or bushy tree [SYo93]. We give examples of these tree shapes on Figure 18.




***Left deep tree*** ***Bushy tree*** ***Right deep tree***

**Sample healthcare schema:**

*Doctor (*DocId, name, specialty, *…)*
*Visit (*VisId, DocId, date, diagnostic, *…)*
*Prescription (*VisId, DrugId, qty, *…)*
*Drug (*DrugId, name, type, *…)*

**Query:**

```
SELECT D.name
FROM Doctor D, Visit V, Prescription P, Drug D
WHERE D.DocId = V. DocId AND V.VisId = P.VisId
    AND P.DrugId = D.DrugId AND D.type = 'ANTIBITIC'
    AND V.date > 01-jan-2000;
```

**Figure 18** *Different Tree Shapes of the Same Query.*

In a left deep tree, joins are processed sequentially, each join result being materialized in main memory (or swapped to persistent storage when the cache memory is full). Conversely, right deep trees operate all joins in a pipeline fashion. Thus, each produced tuple can be used by the following operator before the entire result is produced. Bushy trees are most common since they can materialize or pipeline any operator. Most optimizers consider only linear trees since bushy trees lead to a very high number of QEPs.

A cost can be associated to each QEP, thanks to database statistics (*e.g.*, including relation's cardinality or valued distribution) and result size estimations (refer to [MCS88] for an extensive survey). In case of main memory resident data, the query optimizer must focus on CPU and memory access costs to build the most appropriate QEP [GaS92]. We refer the reader to [GaS92, LiN96, LiN97] for detailed cost models in main memory DBMS.

The QEP search space can be reduced using heuristics (*e.g.*, always choose the smallest relation for the left input), dynamic programming techniques [SAC+79], or others - less used - strategies (*e.g.*, randomized techniques [IoK90]).

Since the goal of this section is only to give basic notion on query optimization, we refer the reader to [SAC+79, JaK84, Cha98, Ion96] for an overview of disk-based DBMS optimization techniques.

# 3 Cryptography

Cryptography is the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin

authentication [MOV97].

The objective of this section is not to give basic knowledge about cryptography but to describe only on a subset of the existing techniques that may be necessary to understand (or criticize) the solution proposed in Chapter 5.

Our first approach to secure confidential data is to keep it inside a secure chip (*i.e.*, a smart card). Indeed, the hardware properties of smart cards give to data at rest a very high security level. In Chapter 5, we propose to externalize the data on a remote storage while keeping the same security level. Unfortunately, the remote storage cannot be considered secure since it is outside the security sphere of the smart card[24]. The goal of this section is thus to detail possible alternatives to provide a high security level for the data stored on the remote storage.

This section focuses on two main methods to prevent data hacking. We first present encryption basics, which guarantees data opacity and prevents an unauthorized person to read or interpret the encrypted data. Then, we describe traditional integrity techniques, which detect unexpected modifications on the data. Such cryptographic mechanisms can be viewed as a sensor mesh, coating the data, which breaks as soon as a pirate updates a single bit of the data. Note that another important security concern is guarding data against replay attacks. In our context, replay attacks consists of exchanging a data on the insecure storage by an older (and non-altered) version of this data. In our case, we avoid this attack including timestamps into the encrypted blocs as presented in Chapter 5. Therefore, we do not detail here the usual cryptographic techniques guarding data against replay.

For further information on cryptographic techniques, we refer the reader to specialized books: [MOV97, Sti00, PfL97, Sch96].

## 3.1 Data Encryption

The purpose of encryption is to ensure privacy by keeping the information hidden from any unauthorized person who may see the encrypted data. In a secure system, the plaintext cannot be recovered from the ciphertext except by using the decryption key.

### *3.1.1 Symmetric versus Asymmetric Encryption*

Encryption methods can be divided into *symmetric-key algorithms* and *asymmetric-key algorithms*.

Symmetric-key algorithms use the same *secret key* to encrypt and decrypt the data. It can be used when two parties (*e.g.*, the famous Bob and Alice) share a secret key or when a unique party encrypts and decrypts the data (this last case corresponds to our needs).

[24] For instance, the external memory of the SUMO or the MOPASS project; the external memory of the device in which the smart card is plugged or even an external database server.

Asymmetric-key algorithms use two separate keys (a public key and a private key), and allow, for instance, to securely transfer data between two parties which do not share a secret. Using asymmetric-key algorithms, the data is encrypted with a public key and decrypted with the corresponding private one (the opposite is also possible). For instance, Bob wanting to send a message to Alice can encrypt it with Alice's -widely available- public key. Alice, who owns her private key, will be able to decrypt the message. Asymmetric encryption is extremely useful in practice for secure communication protocols like SSL, digital signature, etc. Asymmetric encryption is however substantially slower than symmetric encryption. For our specific problem, such techniques are useless, since the data is encrypted and decrypted by a single party.

In the following, we thus focus on symmetric-key algorithms and more specifically on *symmetric-key block ciphers*[25]. Block cipher encryption breaks up the plaintext data into blocks of a fixed length of *n* bits (*e.g.*, n = 64), and encrypts one block at a time. Plaintext data having less than *n* bits must be padded before encryption.

### *3.1.2 Encryption Modes of Operation*

Several *modes of operation* for block ciphers can be used when the plaintext data exceeds one block in length: The simplest approach consists of encrypting each block separately and is called *electronic-codebook* (ECB) (see Figure 19).

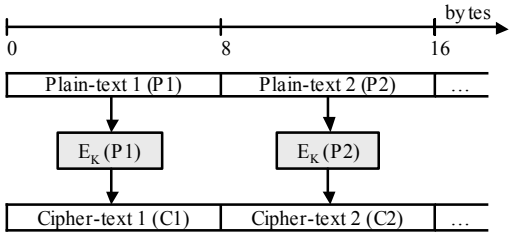



**Figure 19** *Electronic CodeBook (ECB) Encryption.*

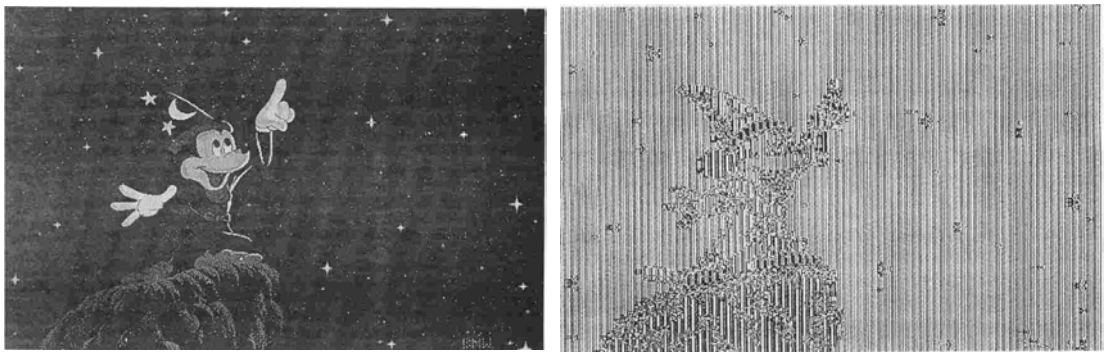
The ECB mode of operation provides two noticeable properties:

*Blocks independencies*: Each block is enciphered independently of other blocks. Malicious reordering of ciphertext blocks does not affect the decryption and produces reordered plaintext blocks. Also, one or more bit errors in a single ciphertext block affect the decryption of that block only.

*Repetitive patterns*: Using the same encryption key, identical plaintext blocks result in identical ciphertext. Thus, it hurts opacity of encrypted data since data patterns (*i.e.*, repetitive sequences) can be observed in the ciphertext. This problem is exemplified on

[25] Indeed, the other class of symmetric-key algorithms, *stream ciphers*, is not adequate for encrypting data at a rest. Moreover theses algorithms are not available in current smart cards.

Figure 20(b)[26]. It clearly shows that the ECB encryption (using the 3-DES algorithm – see below) of the picture of Figure 20(a) is not sufficient to provide secrecy.



*(a) Original* *(b) 3-DES-ECB encryption*

**Figure 20** *ECB Mode Opacity.*

For many applications, this last drawback motivates the use of other modes of operation, providing higher opacity.

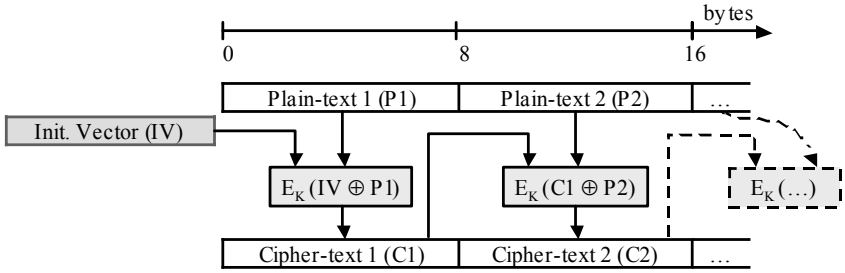



**Figure 21** *Cipher Block Chaining (CBC) Encryption.*

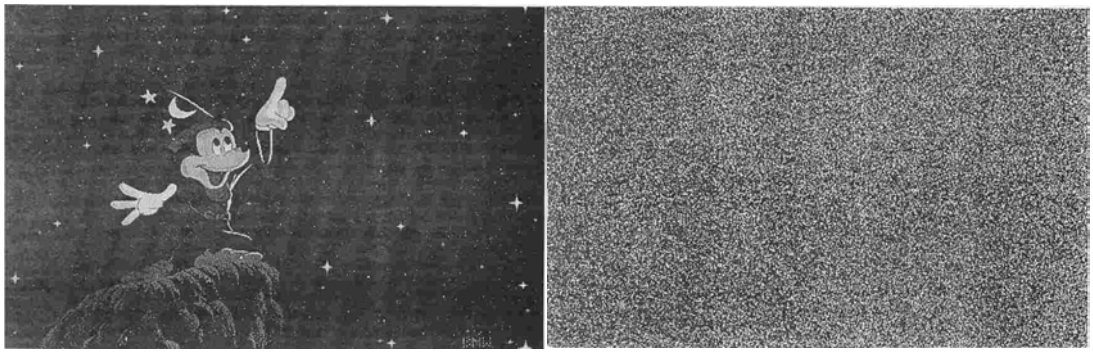
The *Cipher Block Chaining* (CBC) avoids this drawback by using the following encryption method: Before encrypting a plaintext block $P_i$ ($i > 1$), an exclusive-or is made between $P_i$ and the result of the previous encryption (*i.e.*, block $C_{i-1}$) (see Figure 21). An *Initialization Vector* (IV) is required to encrypt the first block (IV can be public without hurting security). Thus, encryption depends not only on the key and plaintext but also on the previous encryption. CBC decryption is done using the reverse process, *i.e.*, making an exclusive-or after decryption with the previous encrypted block.

CBC exhibits two important properties in our context:

*Non-repetitive patterns*: identical ciphertext blocks are obtained when the same plaintext is enciphered under the same key and IV. Sometimes, the IV is called *nonce* in the literature, meaning that it should be used only once. This property allows hiding data pattern in the ciphertext, thus providing opacity. Figure 22 exemplifies this property and show the advantage of the CBC mode of operation on the previous example.

[26] These figures are taken from [Sch].

*Chaining dependency*: the chaining mechanism causes ciphertext block to depend on all preceding plaintext blocks. The entire dependency on preceding blocks is, however, contained in the value of the previous ciphertext block, thus decryption of a ciphertext block only requires the preceding ciphertext block. Consequently, and differently from ECB, malicious reordering of ciphertext blocks does affect decryption.



*(a) Original* *(b) 3-DES-CBC encryption*

**Figure 22** *CBC Mode Opacity.*

Other block chaining modes process $r$ bits at a time, with $r < n$, like the *Cipher FeedBack* mode (CFB) and the *Output FeedBack* mode (OFB) [MOV97]. However, these algorithms are not further discussed here since the induced throughput for processing successive blocks of $r$ bits $(r < n)$ is decreased by a factor of $n/r$ versus CBC. This is due to the fact that each encryption/decryption function yields only $r$ bits of the ciphertext output, and discards the remaining $n$-$r$ bits.

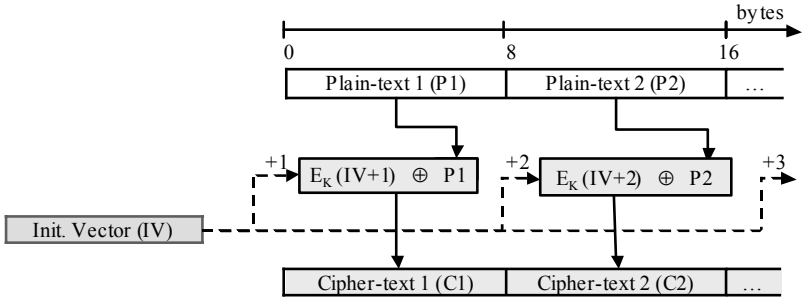



**Figure 23** *Counter Mode (CTR).*

Finally, the *Counter* mode (CTR) was proposed by Diffie and Hellman in 1979 and is based on the OFB mode. It also requires an initialization vector $IV$. A plaintext block $P_i$ will be encrypted by making an exclusive-or with the encryption result of $IV+i$ (see Figure 23). The benefit of this encryption mode is that any block can be easily decrypted as soon as the IV, the key, and the position are known, thus allowing encryption/decryption in parallel as opposed to the CBC mode.

### *3.1.3 Encryption Algorithms*

There are many popular implementations of symmetric key ciphers, some of which being standardized or placed in the public domain. They include AES [NIS01], DES [NIS77, NIS88, NIS93], Triple DES [KMS95], blowfish and RC5.

The well-known DES cipher (Data Encryption Standard) was first published in 1977 as an open and fully specified US standard. DES works on blocks of 64 bits and uses a 64-bit key, the effective size being 56 bits (8 bits are used as parity bits). It was extensively used and is still being used despite its weakness. Indeed, with the evolution of the computers processing power, it was recognized that a 56-bit key was simply not large enough for high security applications.

Triple DES [ANS98] is in fact another mode of DES operation using three distinct keys. The procedure for 3-DES encryptions is as follows: encrypt with the first key, decrypt with the second key, and encrypt again with the third key (see Figure 24). The advantage of 3-DES is its longer key (192 bits with 168 effective) which make it much more secure than DES.

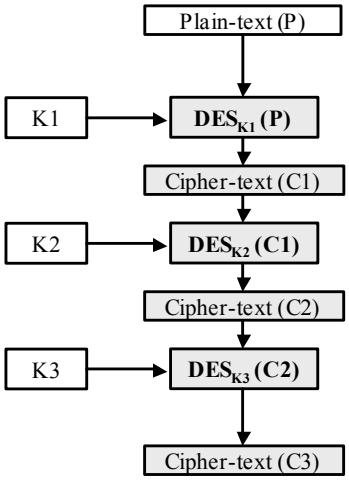



**Figure 24** *Triple DES.*

As a result of DES weaknesses, the National Institute of Standards and Technology (NIST) abandoned their support of DES in 1997 and launch an international competition to develop the Advanced Encryption Standard (AES) as a replacement for DES. In the meantime, NIST approved triple DES (3-DES) as a temporary standard. In 2001, NIST [NIS01] recommended the use of Advanced Encryption Standard (AES) to replace DES, and if this is not possible then to use Triple DES.

The AES is based on the Rijndael block cipher. Rijndael is one of the five finalists chosen by NIST from a first list of 15 candidates for the AES competition. The algorithm has a variable block length and a key length. It is currently specified to use keys with a length of 128, 192, or 256 bits to encrypt blocks with a length of 128, 192 or 256 bits. Rijndael can be

implemented very efficiently on a wide range of processors and in hardware. Like all AES candidates, Rijndael is very secure and has no known weaknesses. Detailed information on Rijndael can be found on [Rij].

The recommended minimum key length for all symmetric ciphers is 128 bits. Key length is extremely important to data security. Indeed, *brute force* attacks is becoming more successful as increasingly powerful computers make it possible to create and test a large number of keys in relatively short periods of time. For instance, in 1999, *Distributed.Net* used the specifically developed computer *DES Cracker* and a worldwide network of nearly 100,000 PCs to break the DES algorithm in less than 24 hours. The DES Cracker and PCs combined were testing 245 billion keys per second when the correct key was found. More information on the level of protection afforded by different encryption algorithms and key lengths is available in *Information Warfare and Security* [Den00].

## 3.2 Authentication and Integrity

The assumption that an attacker must know the secret key in order to manipulate encrypted data is simply not correct. In some cases, he may be able to effectively manipulate ciphertext despite not being able to control its specific content. Three examples will help understanding this point: (i) an attacker may randomly manipulate the data (*e.g.*, manipulate bits of an encrypted medical prescription may induce refund piracy); (ii) he may simply suppress a subset of the encrypted data (*e.g.*, unwanted logs) or duplicate data (*e.g.*, bank transfers); (iii) he may substitute blocks within the same data (*e.g.*, access unauthorized information by exchanging authorized and non authorized blocks). Moreover, even if this tampering generates incorrect decryption, the system (or the users) might not always perceive it. This section gives an overview of the cryptographic techniques, which provide integrity to encrypted data.

In a first section, we describe hash functions, which are essential for build integrity checking techniques. The results of hash functions, used directly to detect whether the data has been tampered are called *Modification Detection Codes* (MDC). When hash functions are associated to a secret key, their output are called *Message Authentication Code* (MAC). Since it is assumed that the algorithmic specification of a hash function is public knowledge, anyone may compute the MDC of a given message, whereas only the party with knowledge of the key may compute its MAC. These main hashing techniques and their usage in the tampering detection process are described in the following.

### *3.2.1 Cryptographic Hash Functions*

*Cryptographic Hash Functions* (CHF) are mathematical operations which, roughly speaking, take some data of arbitrary length and compute *easily* a hash value, *i.e.*, a fixed size "digest", of that data. Therefore, CHF share the two basics properties of any hash

function: *ease of computation* and *compression.* To be cryptographically sound, a CHF should ensure some additional properties:

- *Preimage resistance:* it is computationally infeasible to find any input of the hash function given its output, (*i.e.*, given a hash-value *h*, find some data *D* such that *H(D) = h*). This property should be true for all possible hash-values (*h)*.
- *2nd preimage resistance:* it is computationally infeasible to find any second input which has the same hash value as a given input, (*i.e.*, given some data *D*, find some data *D'* such that *H(D) = H(D')*). This property should be true for any possible input (*D*).
- *Collision resistance:* it is computationally infeasible to find two distinct inputs which hash to the same value (*i.e.*, find any couple of data *D* and *D'* such that *H(D) = H(D')*).

*One-way hash functions* (OWHF) ensure preimage resistance and 2nd preimage resistance while *collision resistant hash functions* (CRHF) ensure 2nd preimage resistance and collision resistance.

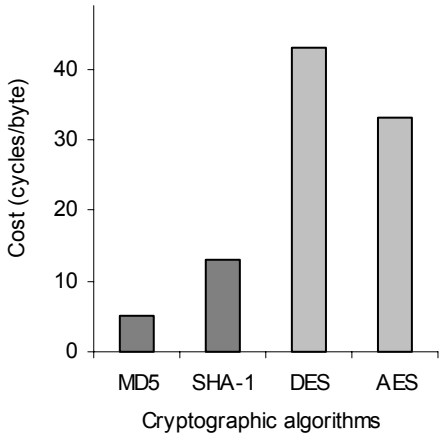


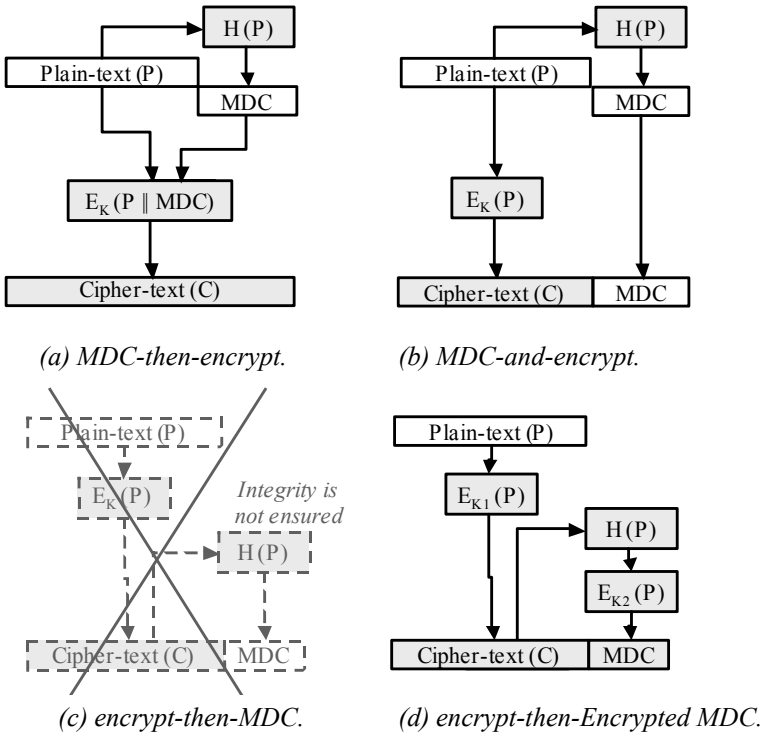

**Figure 25** *C ANSI Comparative Performance on Pentium [Nes01, Nes03].*

Popular cryptographic hash functions include *Message Digest 5* (MD5) [Riv92] and *Secure Hash Algorithm 1* (SHA-1) [NIS95]. Both hash functions process the data by blocks of 512 bits, the size of the hash function output being 160 bits for SHA-1 and 128 bits for MD5. Both hash functions are also considered secure since finding a preimage for SHA-1 (resp. MD5) requires $2^{160}$ (resp. $2^{128}$) hash operations while finding a collision requires $2^{80}$ (resp. $2^{64}$) hash operations. Considering algorithms performance, hash functions are faster than encryption. We plot on Figure 25 the performances of popular encryption and hashing primitives, as reported in [Nes01, Nes03]. A comparative performance study of cryptographic protocols on PC can be found in [PRB98] and on embedded systems and

sensor networks in [GVP+03]. Moreover, energy consumption analyses have been conducted in [PRR+03].

### *3.2.2 MDC Based Integrity*

Ensuring data integrity using a MDC requires the data or its hash representation to be protected by encryption, in order to prevent a pirate from altering the data and re-computing a valid hash. In our context, the data must at least be encrypted since it is confidential.




*(a) MDC-then-encrypt.* *(b) MDC-and-encrypt.*

*(c) encrypt-then-MDC.* *(d) encrypt-then-Encrypted MDC.*

**Figure 26** *MDC Based Integrity.*

Several integrity schemes can be envisioned. We detail below (and in Figure 26) some interesting schemes:

- *MDC-then-encrypt:* (1) the plaintext is hashed; (2) the plaintext and the MDC are encrypted.
- *MDC-and-encrypt:* (1) the plaintext is hashed; (2) only the plaintext is encrypted (the MDC is not).
- *Encrypt-then-MDC*: (1) the plaintext is encrypted; (2) the ciphertext is hashed (the MDC is not encrypted)
- *Encrypt-then-Encrypted MDC*: (1) the plaintext is encrypted; (2) the ciphertext is hashed (3) the MDC is encrypted

The hash value properties (*i.e.*, preimage resistance) prevent a hacker from retrieving the plaintext from the MDC for the two first schemes. Thus, while plaintext is necessarily encrypted to ensure confidentiality, the MDC encryption seems optional. However, whereas two identical data should be encrypted differently (for opacity, see Section 3.1.2), they will result in the same MDC using the MDC-and-encrypt solution. Thus, for the sake of opacity, we should avoid this scheme or add a nonce (*i.e.*, number used only once) or, for instance, the initialization vector of the CBC, prior to the MDC computation.

Encrypt-then-MDC does not provide any integrity. Indeed, a pirate may modify the ciphertext then re-compute a valid MDC (since the algorithms are public). The last scheme we propose is to encrypt the MDC computed on the encrypted data. This scheme allows for checking the integrity without having to decrypt the whole data. This property can be very interesting, in our case, since we may be interested in only a subset of a large encrypted data.

### *3.2.3 MAC Based Integrity*

MAC algorithms take two distinct inputs, a message and a secret key, and produce a fixed-size output. The objective is that in practice the same output cannot be produced without knowledge of the key. MACs can be used to provide data integrity and symmetric data origin authentication.

MAC algorithms can be based on block ciphers like CBC-MAC [Iwa03], or be based MDC by performing additional operations to plaintext with a secret key before computing the hash value. In the latter case, the cost of the operation is comparable to MDC computation. We refer to [MOV97] for the description of various MAC algorithms and restrict our discussion to the description of HMAC, which is a low cost key hashing algorithm that produces a compact hash value.

HMAC [BCK96, KBC97] is not an algorithm but a construction that can be based on MD5 and SHA-1 to obtain HMAC-MD5-96 [MaG98a] and HMAC-SHA-1-96 [MaG98b]. Both reduce the hash value to 96 bits (instead of 128 or 192 bits). With *e.g.*, SHA-1 and a key $k$ of 512 bits[27], HMAC works as follows: We compute the HMAC on a data $D$ as $HMAC(D) = SHA\text{-}1(k \,||\, SHA\text{-}1(k \,||\, D))$. Moreover, the result can be truncated (with at least 80 bits)

The MAC being based on a secret key, it can either be computed on the plaintext or the ciphertext. Basing the MAC on the ciphertext enables to check data integrity without deciphering it. Note, however, that enciphering a message with the key used to compute the MAC has been proven as insecure, leading to use two separate keys, one for computing the checksum and one for encrypting the data. Figure 27 shows several possible ways to ensure integrity and privacy with a MAC and a cipher: MAC-then-encrypt, MAC-and-encrypt, and encrypt-then-MAC. Note that, as mentioned for MDC-and-encrypt, MAC-and-encrypt hurts

[27] For the sake of clarity, we use simple examples that avoid padding.

data opacity. We refer to [BeN00] for a complete analysis of the relative strengths of these different processes.

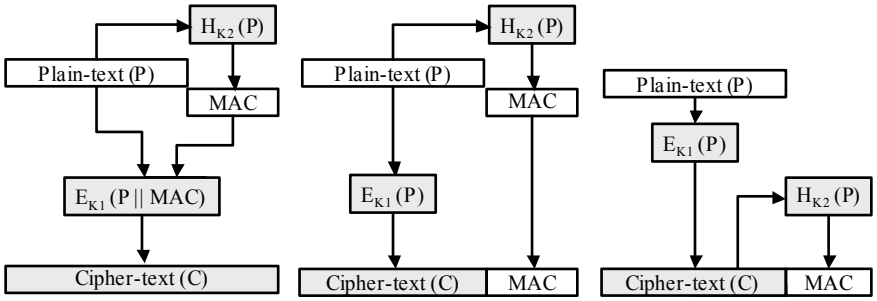



**Figure 27** *MAC Based Integrity.*

### *3.2.4 Authenticated Encryption Schemes*

During the past three years, cryptographs have worked at combining integrity (or authentication) and encryption in the same algorithm. Indeed, traditional techniques based on MAC require two passes on the data to provide data integrity (security imposes two different keys for encryption and MAC), and thus generate performance slowdown. Representative algorithms that provide both encryption and integrity are IACBC [Jut01] which provides integrity with a 10% slowdown compared to traditional DES-CBC and is patented by IBM, RPC mode [KaY00] and OCB [RBB+01]. These algorithms are compared in [AnB01].

# Chapter 3 – Query Processing in a RAM Constrained Environment

*This chapter is taken from [ABP03a] and gives a thorough analysis of the RAM consumption problem. It also paves the way for setting up co-design rules helping to calibrate the RAM resource of a hardware platform according to given application's requirements as well as to adapt an application to an existing hardware platform.*

## 1 Introduction

Pervasive computing is now a reality and intelligent devices flood many aspects of our everyday life. As stated by the Semiconductor Industry Association, the part of the semiconductors integrated in traditional computers represents today less than 50% of a market of $204Billion [SIA02]. As new applications appear, the need for database techniques embedded in various forms of lightweight computing devices arises. For example, the vision of the future dataspace, a physical space enhanced with digital information made available through large scale ad-hoc sensor networks is paint in [ImN02]. Sensor networks gathering weather, pollution or traffic information have motivated several recent works [MaH02, BGS00]. They have brought out the need for executing local computation on the data, like aggregation, sort and top-n queries [CaK97], either to save communication bandwidth in push-based systems or to participate in distributed pull-based queries [MFH+02]. Personal folders on chip constitute another motivation to execute on-board queries. Typically, smart cards are used in various applications involving personal data (such as healthcare, insurance, phone books etc.). In this context, queries can be fairly complex (*i.e.*, they can involve selections, joins and aggregation) and their execution must be confined on the chip to prevent any disclosure of confidential data [PBV+01]. Hand-held devices are other forms of autonomous mobile hosts that can be used to execute on-board queries on data downloaded before a disconnection (*e.g.*, personal databases, diary, tourist information). Thus, saving communication costs, preserving data confidentiality and allowing disconnected activities are three different concerns that motivate the execution of on-board queries on lightweight computing devices [NRC01, Ses99].

While the computing power of lightweight devices globally evolves according to Moore's law, the discrepancy between RAM capacity and the other resources, notably CPU

speed and stable storage capacity, still increases. This is especially true for Systems on Chip (SoC) [NRC01, GDM98] where RAM competes with other components on the same silicon die. Thus, the more RAM, the less stable storage, and then the less embedded data. As a consequence, SoC manufacturers privilege stable storage to the detriment of a RAM strictly calibrated to hold the execution stack required by on-board programs (typically, less than 1 kilobytes of RAM is left to the applications in smart cards even in the advance prototypes we recently experimented). This trade-off is recurrent each time the silicon die size needs to be reduced to match physical constraints such as thinness, energy consumption or tamper resistance. Another concern of manufacturers is reducing the hardware resources to their minimum in order to save production costs on large-scale markets [SIA02]. Thus, RAM will remain the critical resource in these environments (see Chapter 2, Section 1.2.3 for details) and being able to calibrate it against data management requirements turns out to be a major challenge.

As far as we know, there is today no tool nor academic study helping to calibrate the RAM size of a new hardware platform to match the requirements of on-board data centric applications. In traditional DBMSs, query evaluation techniques resort to swapping to overcome the RAM limitation. Unfortunately, swapping is proscribed in highly constrained devices because it incurs prohibitive write costs in electronic stable memories and it competes with the area dedicated to on-board data. In the absence of a deep understanding of RAM consumption principles, pragmatic solutions have been developed so far. Light versions of popular DBMS like Sybase Adaptive Server Anywhere [Gig01], Oracle 9i Lite [Ora02a], SQLServer for Windows CE [SeG01] or DB2 Everyplace [KLL+01] have been designed for hand-held devices. Their main concern is reducing the DBMS footprint by simplifying and componentizing the DBMS code [Gra98]. However, they do not address the RAM issue. Query executions exceeding the RAM capacity are simply precluded, thereby introducing strong restrictions on the query complexity and on the volume of the data that can be queried. Other studies have been conducted to scale down database techniques in the particular context of smart cards [ISOp7, PBV+01]. In [PBV+01], we tackled the RAM issue by proposing a dedicated query evaluator relying on an ad-hoc storage and indexation model. This solution, called PicoDBMS, has been shown convenient for complex personal folders (*e.g.*, healthcare folders) embedded in advanced smart card platforms [ABB+01]. Without denying the interest of the PicoDBMS approach, its application's domain is reduced by two factors. First, PicoDBMS makes an intensive use of indices with a side effect on the update cost and on the complexity of the transaction mechanisms enforcing update atomicity. One may thus wonder whether the resort to indices could be avoided, and in which situations. Second, PicoDBMS constitutes an ad-hoc answer to a specific hardware platform. As noticed earlier, hardware configurations are more and more specialized to cope with specific requirements in terms of lightness, battery life, security and production cost. Building an ad-hoc query evaluator for each of them will rapidly become cumbersome and, above all, will incur a prohibitive design cost [NRC01].

In the light of the preceding discussion, there is a clear need for defining pre-designed and portable database components that can be integrated in Systems on Chip (SoC). The objective is to be able to quickly differentiate or personalize systems in order to reduce their cost and their time-to-market [GDM98]. To this end, a framework for designing RAM-constrained query evaluators has to be provided. We precisely address here this issue, following the three steps approach outlined below.

*Devising a RAM lower bound query execution model*

This study proscribes swapping and indexing for the reasons stated earlier. Searching for a RAM lower bound query execution model in this context forces us to concentrate on the algorithmic structure of each relational operator and on the way the dataflow between these operators must be organized. The contribution of this step is to propose operators' algorithms that reach a RAM lower bound and to guidelines that remain valid when a small quantity of RAM is added to the architecture.

*Devising optimization techniques that do not hurt this RAM lower bound*

Obviously, a RAM lower bound query execution model exhibits poor performance. The absence of swapping and indexing leads to recompute repeatedly every information that cannot be buffered in RAM. The consequence on the query execution algorithms is an inflation in the number of iterations performed on the on-board data. The contribution of this step is to propose new optimization techniques that drastically reduce the number of irrelevant tuples processed at each iteration.

*Studying the impact of an incremental growth of RAM*

Mastering the impact of RAM incremental growths has two major practical outcomes. In a co-design perspective, it allows to determine the minimum amount of RAM required to meet a given application's requirement. In the context of an existing hardware platform, it allows to calibrate the volume of on-board data and the maximum complexity of on-board queries that remain compatible with the amount of available RAM. As demonstrated by our performance evaluation, very small RAM growths may lead to considerable performance gains. This motivates further the study of a RAM lower bound query execution model and of RAM incremental variations. The contribution of this step is twofold. First, it proposes an adaptation of the preceding execution techniques that best exploit each RAM incremental growth and demonstrates that they constitute an accurate alternative to the index in a wide range of situations. Second, it proposes co-design guidelines helping to find the best compromise between RAM capacity, volume of on-board data, query complexity and response time.

This chapter is organized as follows. Section 2 introduces important assumptions that

delimit the context of the study. Section 3 presents our RAM lower bound query execution model. Section 4 addresses optimization issues in this RAM lower bound context. Section 5 describes the impact of RAM incremental growths on the query execution model. Section 6 presents our performance evaluation. Finally, Section 7 concludes the chapter.

# 2 Context of the Study

This section introduces hypothesis on the data sources, on the queries and on the computing devices, and discusses their relevance with respect to the target of this study.

*H1 : On-board data sources are sequential files*

We assume that the data sources are hosted by the device and do not benefit from any index structure. The reason to push indices aside from this study is threefold. First, indices have a negative side effect on the update cost (as we will see, this effect is magnified by hypothesis H4). Second, indices make update atomicity more complex to implement [PBV+01] and then have also a negative side effect on the algorithm's footprint. Finally, indices compete with on-board data on stable memory, thereby reducing the net storage capacity of the device. This does not mean that indices are definitely useless or unsuitable. As shown in Section 6, indices remain the sole solution to cope with strict response time constraints in the case of complex queries over a large amount of data. One objective - and contribution - of this study is to demonstrate that alternatives to indices exist and are convenient in a wide range of situations.

*H2 : Queries are unnested SQL queries*

We consider relational data for the sake of generality and simplicity. Note that the relational model has become a standard even in highly constrained environments [ISOp7, PBV+01]. The queries of interest are unnested SQL queries including Group by, Having and Order by statements. Even if more complex queries could be considered, unnested SQL queries are expressive enough to cover the need of the target on-board data centric applications (sensors, personal folders, ambient intelligence).

*H3 : Computing devices are autonomous*

Autonomy means that the execution of on-board queries relies only on local computing resources. Obviously, if we assume that external resources can be exploited, the RAM problem vanishes. As stated in the introduction, saving communication costs, preserving data confidentiality and allowing disconnected activities are three common motivations to execute on-board queries in autonomy.

*H4 : Stable storage is made of electronic memory*

This assumption simply expresses the reality since lightweight computers use commonly EE-PROM, Flash or power-supplied RAM technologies for their stable storage [NRC01]. These technologies exhibit good properties in terms of lightness, robustness, energy consumption, security and production costs compared with magnetic disks. In addition, fine grain data can be read-accessed from stable storage at a very low cost (the gap between EE-PROM, Flash and RAM in terms of direct read-access time is less than an order of magnitude). On the other hand, writes in EEPROM and Flash are extremely expensive (up to 10 ms per word in EE-PROM) and the memory cell lifetime is limited to about $10^5$ write cycles.

The conjunction of H3 and H4 precludes a query evaluator resorting to memory swap. Indeed, swapping incurs prohibitive write costs and may hurt the memory lifetime depending on the stable storage technology used. But above all, the swapping area must be local and there is no way to bound it accurately. Again, the swapping area competes with the on-board data in stable storage.

# 3 RAM Lower Bound Query Execution Model

This section concentrates on the definition of a query execution model that reaches a lower bound in terms of RAM consumption, whatever be the complexity of the query to be computed and the volume of the data it involves. The RAM consumption of a *Query Execution Plan* (QEP) corresponds to the cumulative size of the data structures used internally by all relational operators active at the same time in the QEP plus the size of the intermediate results moving along the QEP. We first present two design rules that help us to derive the principles of a *RLB* (RAM Lower Bound) query evaluator. Then, we propose data structures and algorithms implementing this query evaluator.

## 3.1 Design Rules and Consistency Constraints

Two design rules guide the quest for a RLB query evaluator.

*R1 (Invariability): Proscribe variable size data structures*

*R2 (Minimality): Never store information that could be recomputed*

Rule R1 states that a data structure whose size varies with the cardinality of the queried data is incompatible with the reach of a RAM lower bound. As a consequence of R1, if an operator OP1 consumes the output of an operator OP2, OP1's consumption rate must be higher than or equal to OP2's production rate to avoid storing an unbounded flow of tuples. Rule R2 trades performance for space saving. As a consequence of R2, intermediate results

are never stored since they can be recomputed from the data sources at any time. Roughly speaking, R1 rules the synchronization between the operators in the QEP while R2 minimizes the internal storage required by each of them. The conjunction of R1 and R2 draws the outline of a strict pipelined query execution model that enforces the presence of at most one materialized tuple at any time in the whole QEP, this tuple being the next one to be delivered by the query evaluator.

Let us give the intuition of such a query evaluator on an example. Let assume a Join operator combining two input sets of tuples called *ILeft* and *IRight*. *ILeft* and *IRight* result themselves from the evaluation of two sub-queries $Q_{Left}$ and $Q_{Right}$ in the query tree. To comply to R1, each *ILeft* tuple will be compared to *IRight* tuples, one after the other, at the rate of their production by $Q_{Right}$, and the result will be itself produced one tuple at a time. To comply with R2, $Q_{Right}$ will be evaluated $\|ILeft\|$ times in order to save the storage cost associated to *IRight*. Following this principle for all operators and combining them in a pipelined fashion is the intuition to reach a RAM lower bound for a complete QEP. However, care must be taken on the way the dataflow is controlled to guarantee the consistency of the final result. To this end, we introduce two consistency constraints that impact the iteration process and the stopping condition of each algorithm presented in the next section

*Unicity: the value of a given instance of the Cartesian product of the involved relations must be reflected at most once in the result.*

*Completeness: the values of each instance of the Cartesian product of the involved relations that satisfies the qualification of the query must be reflected in the result.*

## 3.2 Execution Model and Notations

Before going into the details of each operator's algorithm, we discuss the way operators interact in a QEP. We consider the Iterator model [Gra93] where each operator externalizes three interfaces: *Open* to initialize its internal data structures, *Close* to free them and *Next* to produce the next tuple of its output (see Chapter 2, Section 2.4 for details). The QEP takes the form of a tree of operators, where nodes correspond to the operators involved in the query, leaves correspond to the Scan operator iterating on the involved relations and edges materialize the dataflow between the operators. Figure 28 pictures a simple QEP and introduces notations that will be used in the algorithm's description.

Rules R1 and R2 guarantee the minimality of the dataflow by reducing its cardinality to a single tuple. A shared data structure called *DFlow* materializes this dataflow. Each operator uses in its turn this data structure to consume its input and produce its output, one tuple at a time. More precisely, *DFlow* contains: (i) one cursor maintaining the current position reached in each relation involved in the QEP; these cursors materialize the current instance

of the Cartesian product of these relations; and (ii) a storage area for each attribute computed by an aggregate function. Thus, *DFlow* contains the minimum of information required to produce a result tuple at the root of the QEP and is the unique way by which the operators communicate. As pictured in Figure 28, cursors and attributes in *DFlow* are organized into lists to increase the readability of the algorithms (*e.g.*, *GrpLst* denotes the list of cursors referencing the relations participating in the grouping condition).

Additional notations are used all along the chapter. ILeft and IRight denote respectively the left and right inputs of a binary operator; ||input|| denotes the tuple cardinality of an operator's input; k denotes the number of distinct values in an input with respect to a grouping or a sorting condition.

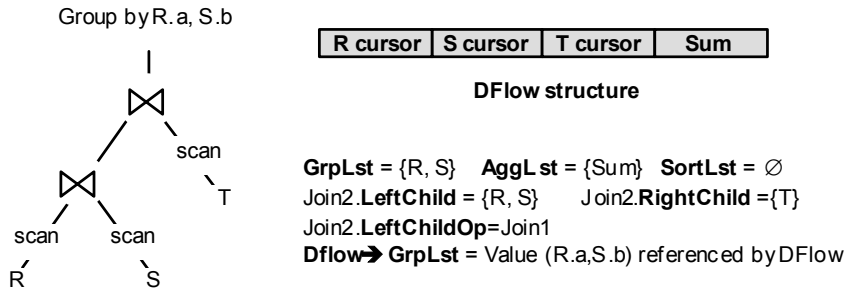



**Figure 28** *Query Example and Notations.*

## 3.3 Operator's Algorithms

Following the precept of rule R2, we present below a RLB form of each relational operator's algorithm. For each algorithm, we show its compliance with the Unicity and Completeness consistency constraints and give its RAM consumption.

### *3.3.1 Scan, Select, Project and Join Algorithms*

The algorithms implementing these operators are pictured in Figure 29, except for the Project that is rather straightforward. Project simply builds a result tuple by dereferencing the cursors present in *DFlow* and by copying the values belonging to *Dflow➔AggLst*. The Scan and Select algorithms are self-explanatory. Join implements a Cartesian product between its left and right inputs. Let us verify Unicity and Completeness for each algorithm. Regarding the Scan, the same tuple cannot be considered twice between an Open and a Close and all tuples are considered since Open initializes the cursor to the beginning of the relation and the stopping condition of Next is to reach EOF. Select and Project are unary operators that consume their input sequentially and then inherit Unicity and Completeness from their child operator in the QEP. The Join algorithm performs a nested loop on its operands so that each instance of the Cartesian product of its left and right inputs is examined exactly once. Again, Unicity and Completeness are inherited from its left and right child operators in the QEP.

```
Scan.Open   Dflow.Ri←0                          //  Initialize Ri scan
Scan.Next   Dflow.Ri ++                         //  reference next Ri tuple
            if Dflow.Ri > Ri.Cardinality then Dflow.Ri←EOF  //  check EOF
Select.Open ChildOp.Open()
Select.Next repeat                              //  Get Next child tuple and
              ChildOp.Next()                    //check the selection predicate
            until DFlow.Child = EOF or match(SelectionPredicate)
Join.Open   LeftOp.Open();RightOp.Open() //  Open L and R input
            if DFlow.LeftChild=EOF or DFlow.RightChild=EOF then
              DFlow.Child←EOF               //  One input is empty
            else LeftOp.Next()              //  get the first L tuple
Join.Next   if DFlow.Child ≠ EOF then       //  check both L&R EOF
              Repeat                        // Get a R tuple until EOF or match
                RightOp.Next()
                if DFlow.RightChild = EOF then
                  LeftOp.Next()             //if end of R, get a next L tuple
                  if DFlow.LeftChild ≠ EOF then
                    RightOp.Open()          //  and reopens R
                    RightOp.Next()
              until DFlow.Child = EOF or match(JoinPredicate)
GBY.Open    ChildOp.Open()                  //  Scan the whole input in
            Split.Value←+∞                  //  order to find the smallest
            repeat                          //  grouping value (GV) in Split
              ChildOp.Next()
              if Dflow➔GrpLst < Split.Value then
                Split←DFlow.GrpLst          //  Split converges to the
            until DFlow.Child = EOF         //  smallest GV at EOF
GBY.Next    if Split.Value ≠ +∞ then        // there is a next GV to compute
              Current←Split                 //  Initialize the computation
              DFlow.AggLst←0                //  of the current GV
              Split.Value←+∞                //  Initialize Split for computing
              ChildOp.Open()                //  the next GV
              repeat                        //  scan the whole input
                ChildOp.Next()
                if Dflow➔GrpLst = Current.Value then  // the current tuple
                  compute DFlow➔AggLst      //  shares the GV
                elseif DFlow➔GrpLst∈] Current.Value,Split.Value [ then
                  Split←DFlow.GrpLst        //  Split converges
              until DFlow.Child = EOF
              DFlow.GrpLst←Current          //  Output an aggregate
Sort.Open   ChildOp.Open()                  //  Identical to GBY.Open
            Split.Value←+∞
            repeat
              ChildOp.Next()
              if DFlow➔SortLst<Split.Value then
                Split←DFlow.SortLst
            until DFlow.Child = EOF
            Current←Split
Sort.Next   ChildOp.Next()
            while  DFlow➔SortLst ≠ Current.Value and
                   (DFlow.Child ≠ EOF or Split.Value ≠ +∞)
              if DFlow.Child = EOF then     //  The input ends but there is
                ChildOp.Open()              //  a sorting value (SV) in Split
                Current←Split
                Split.Value←+∞              //  Reinit. Split to find next SV
              elseif DFlow➔SortLst ∈ ]Current.Value, Split.Value[ then
                Split←DFlow➔SortLst         //  Converges to the next SV
              ChildOp.Next()
            // While loop ends when a tuple with Current SortLst is found
```

**Figure 29** *RLB Algorithms.*

The minimality of these four algorithms (Scan, Select, Project, Join) in terms of RAM

consumption is not questionable since their consumption equals zero. Indeed, they do not use other data structure than *DFlow*.

### *3.3.2 GroupBy Algorithm*

The GroupBy operator aggregates in a single result tuple all input tuples sharing the same grouping value. By rule R2, *DFlow* contains a single tuple and then grouping values have to be processed one after the other. By rule R1, keeping track of the list of all grouping values already processed by the algorithm is precluded. Thus R1 and R2 lead to process the grouping values in a predefined order, so that recording a single value sums up the history of the whole processing. The consequence is that the RAM consumption of the GroupBy algorithm amounts to two variables: *Current* referencing the grouping value being processed at each iteration and *Split* recording the frontier between the grouping values already processed and the ones remaining to be processed. This constitutes a RAM lower bound in the absence of assumption on the initial ordering of the input (hypothesis H1). Different RLB forms of the GroupBy algorithm can be devised depending on the way the *Split* variable is managed.

The first variation of the GroupBy algorithm, called *CompMin*, uses *Split* to reference the next grouping value to be computed. At the first iteration (implemented by Group.Open), the algorithm scans its input and searches into *Split* the smallest grouping value present in the input. At each next iteration, *Current* takes the value of *Split*, then the algorithm scans again its input and aggregates the tuples sharing the grouping value referenced by *Current*. At the same time, the algorithm searches into *Split* the grouping value to be processed at the next iteration, that is the value immediately greater than the one referenced by *Current*. This algorithm then performs (k+1) iterations on its input. Unicity and Completeness are guaranteed by the fact that the grouping values are processed in ascending order. Therefore, the same value cannot be considered in different iterations and all grouping values are considered. Indeed, Group.Open initiates the processing by finding the smallest grouping value and the stopping condition in Group.Next is that there is no grouping value greater than the last processed. In addition, each iteration scans the input sequentially, so that a tuple sharing the grouping value being processed is considered exactly once, assuming the child operator enforces Unicity and Completeness.

The second variation of the GroupBy algorithm, called *CompMax*, uses *Split* to reference the last grouping value that has been processed. During the first iteration, *Current* is used to converge to the smallest grouping value and to compute the resulting aggregate at the same time. To illustrate this, let assume the first grouping value encountered in the input be *v1*, so that *Current* references *v1*. While iterating on the input, tuples having a value higher than *v1* are not considered and tuples sharing the value *v1* participate in the aggregation. If a tuple having a value $v2 < v1$ is encountered, *Current* evolves to *v2* and the aggregation calculus is reinitialized. At the end of the first iteration, *Current* has converged

to the smallest grouping value and the resulting aggregation has been computed. At each next iteration, *Split* takes the value of *Current* and *Current* is used to converge to the next grouping value (*i.e.*, the value immediately greater than the one referenced by *Split*) and to compute the resulting group at the same time. While k iterations suffice to compute all aggregations, a $(k+1)^{th}$ iteration is required to guarantee the Completeness of the algorithm, that is to check that there exist no greater grouping value than the last processed. The remaining of the proof of Unicity and Completeness is equivalent to the CompMin algorithm.

Intuitively, and as its name indicates, *CompMax* does more job than *CompMin* since several aggregation calculus are partially performed before being discarded. However, combining the first iteration of *CompMax* with the next iterations of *CompMin* leads to a third algorithm, called *IterMin*, that implements the GroupBy in only k iterations. Indeed, the first iteration of *CompMax* does not use the *Split* variable. This variable could thus be exploited to determine the second grouping value to be processed, so that the (k-1) next iterations could be computed in the same way as *CompMin*. Figure 30 summarizes the behavior of each algorithm on an input containing a sequence G1<G2 …<Gk of grouping values. For each algorithm, we represent a snapshot of its internal state (*i.e.*, Current, Split and Agg variables) at givens iterations. The gray arrow represents the current tuple being processed at a given iteration (*e.g.*, if the tuple being considered at iteration 2 of CompMin is (G5,8), then the values of Current, Split and Agg are respectively G2, G3 and 0).

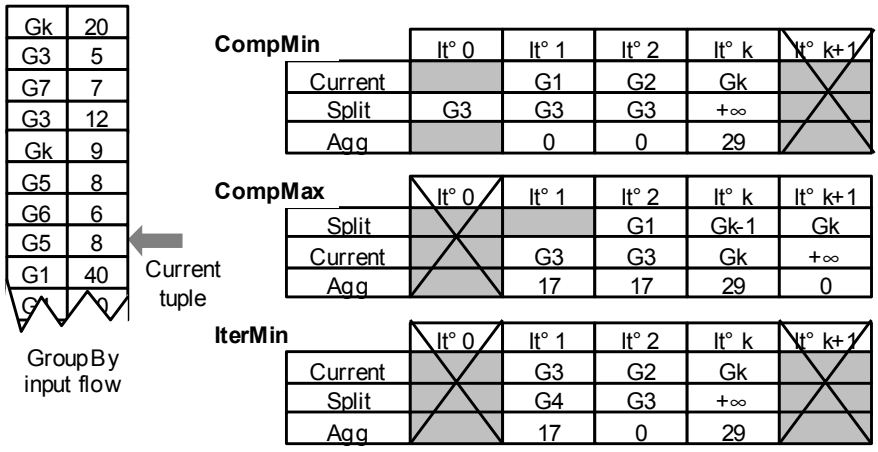



**Figure 30** *Snapshot of the Three GroupBy Algorithms.*

### *3.3.3 Sort Algorithm*

The Sort algorithm shares the same structure, RAM lower bound, number of iterations and proof of Unicity and Completeness as the *CompMin* variation of the GroupBy algorithm. The sole difference is that each iteration delivers the tuples sharing the sorting value referenced by *Current* instead of aggregating them. This algorithm is thus not discussed

further in this section. Note that a CompMax-like variation of the Sort algorithm could be devised but it would lead to $\|input\|$ iterations (instead of k+1) to sort an input flow.

### *3.3.4 Concluding Remarks*

The RAM consumption of a whole QEP incurred by our execution model can be computed as follows. According to the queries of interest (see hypothesis H2), the sole operators that may be involved several times in the same QEP are Scan, Select and Join and their RAM consumption is zero. Therefore, the RAM consumption ascribed to the QEP's operators is the RAM consumed by the GroupBy and Sort operators, namely $2*\|GrpLst\| + 2*\|SortLst\|$, which corresponds to the size of their respective *Current* and *Split* variables. The size of the dataflow corresponds to the size of the *DFlow* structure plus the size of the current tuple being delivered in the final result, that is: $\|FromLst\| + \|AggLst\| + \Sigma_i size(a_i)$, for each $a_i \in p$, $p$ being the Project condition. Consequently, the RAM lower bound to execute a query without index can be expressed by:

$$RLB = \Sigma_i\ size(a_i) + 2 \times \|GrpLst\| + 2 \times \|SortLst\| + \|FromLst\| + \|AggLst\|$$

This result demonstrates the feasibility to design a query evaluator consuming a tiny bounded RAM, independent of the cardinality of the queried data and of the query complexity. One may doubt about the practical interest of RLB expecting that lightweight platforms will be equipped with a much larger RAM. Note however that current smart card platforms provides not much than a few hundred bytes of RAM to the on-board applications, the rest of the RAM being preempted by the operating system, and the JavaCard VM. Regarding future platforms, smart card manufacturers put more emphasis on the increase of CPU speed, communication bandwidth and storage capacity than on RAM. In addition, the objective of co-design is to lower the hardware resources (among them the RAM) to their minimum in order to save production costs on large-scale markets.

But beyond this formula, the significance of this study is on providing guidelines and algorithm's structures that remain valid when a small quantity of RAM is added to the architecture.

# 4 Optimizations in RAM Lower Bound

Not surprisingly, a RAM lower bound query execution model exhibits poor performance since every information that cannot be buffered in RAM needs to be recomputed. The dramatic consequence on the number of iterations performed on the queried data is illustrated in Figure 31. Assuming that k is the number of (R.a,S.b) distinct values present in the GroupBy input, the scan of T turns to be executed k*‖R joinS ‖ times.

Different and complementary solutions can be investigated to decrease this iteration cost without hurting the RAM lower bound. First, global optimization techniques can be used to rearrange the query tree in order to minimize the total number of iterations required to evaluate it. This can be done notably by pushing selections down to the QEP and by finding an optimal join ordering. In a RAM lower bound context, Left-deep trees outperform Right-deep and Bushy trees except for extreme values of the join selectivity's. In conducting our experiments, we observed that the Left-deep tree heuristic remains valid when a small quantity of RAM is added to the model. This confirms the intuition that right subtrees have to be minimized to decrease the cost of recomputing them for each tuple of the left subtree. The ordering of joins in the Left-deep tree depends on their selectivity and on the cardinality of the relations involved, as usual. The query execution in a RAM lower bound exhibits other interesting properties such as: (i) RAM consumption is independent of the number of Join and Select operators, (ii) Join and Select algorithms are order preserving and (iii) intermediate results are never materialized in the QEP. These properties allowed us to devise original optimization techniques detailed in [ABP03b]. However, these techniques are not developed further in this chapter because they have to be reconsidered when a small quantity of RAM is added to the model.

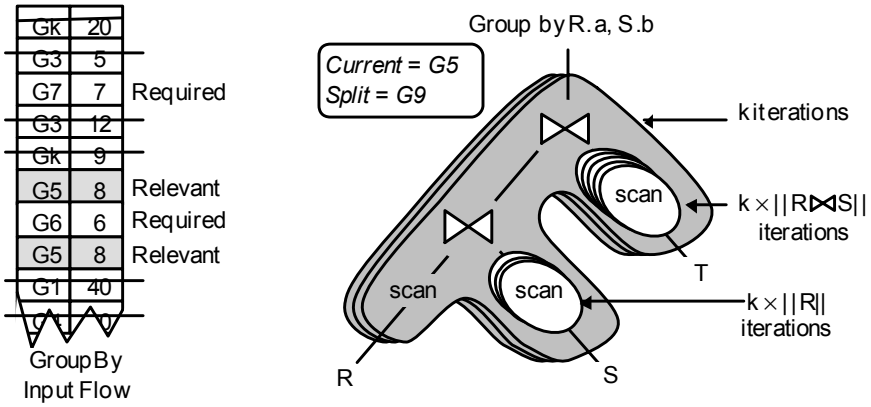



**Figure 31** *Iterations Performed by a RLB Query Evaluator.*

Second, local optimization techniques can be devised to decrease the number of tuples considered inside each iteration. Note that local optimization has here a different meaning than in the usual case since it applies to one iteration rather than to one operator. Having a deeper look on the algorithms presented in Section 3 allows to split, at each iteration, the input flow of each operator into three distinct sets of tuples. As illustrated in Figure 31, *Relevant tuple*s are tuples participating in the operator's result for the current iteration (*e.g.*, tuples sharing the current grouping value being computed by a GroupBy). Obviously, this set of tuples cannot be reduced. *Required tuple*s are tuples modifying the internal state of the algorithm without participating in the iteration's result (*e.g.*, tuples modifying the *Split* variable of the GroupBy algorithm). These tuples are required to enforce the Unicity and/or Completeness of the algorithm and their number depends on the algorithm itself. Thus, the respective merit of different algorithms implementing the same operator can be compared

with respect to the number of Required tuples they consider. Selecting the one minimizing this number is a good heuristic. Finally, *Irrelevant tuple*s are all the tuples present in the input that are not *Relevant* nor *Required*. Eager pruning should take place to avoid producing Irrelevant tuples and carrying them in the QEP up to the operator. The optimizations related to Required and Irrelevant tuples are developed in the following sections.

## 4.1 Minimizing Required Tuples

As stated earlier, the number of Required tuples considered by each algorithm depends on the way Unicity and Completeness are enforced. Regarding the Scan algorithm, one can notice that all tuples present in its input are Relevant. The inputs of the Select, Project and Join algorithms cannot contain Required tuples since they do not maintain an internal state. This comes from the fact that Unicity and Completeness are inherited from their child operator in the QEP. Therefore, these four algorithms are optimal in terms of Required tuples since this number is equal to zero.

The GroupBy algorithm is him directly impacted by the management of Required tuples. Let us first consider the CompMin variation of this algorithm. At a given iteration, all input tuples t such that t.GrpLst∈ ]Current, Split[ are Required, where Current and Split denote the current state of the corresponding variables during this iteration. Indeed, each time an input tuple falls into this interval, it updates the Split variable thereby decreasing the upper bound of the interval. This is the way by which CompMin converges to the next grouping value to be computed and guarantees Unicity and Completeness. Thus, along the same iteration, the number of Relevant tuples (*i.e.*, tuples sharing the grouping value referenced by Current) remains constant while the number of Required tuples decreases and the number of Irrelevant tuples increases from the same amount, according to this converging process. In the CompMax variation of the GroupBy algorithm, the Required tuples at a given iteration are all the input tuples t such that t.GrpLst∈ ]Split, Current]. Each input tuple falling into this interval either lower the value of Current (if t.GrpLst<Current) or participate in the aggregation of the grouping value (if t.GrpLst=Current). This is the way by which CompMax converges to the grouping value to be computed at this iteration, computes it and guarantees Unicity and Completeness. Note that once convergence is achieved, all input tuples sharing the Current value are Relevant rather than Required. Let us now compare both algorithms with respect to the number of Required tuples they consider. Along iteration i (with i>1), CompMin.Current references the same value as CompMax.Split and, at the end of the iteration, CompMin.Split references the same value as CompMax.Current. However, the upper bound of the interval is open in CompMin while it is closed in CompMax. This strongly accelerates the convergence of this upper bound and makes CompMin much more efficient than CompMax in terms of Required tuples. Indeed, CompMin considers at each iteration at most one Required tuple per distinct grouping value

remaining to be processed while CompMax considers at most all the tuples sharing these values.

Comparing *CompMin* to *IterMin* requires a deeper look at the first iterations. The first iteration of *IterMin* produces the same result as the first two iterations of *CompMin*, that is, detecting and processing the smallest grouping value. Again, converging to this value is faster in *CompMin* because at most one Required tuple has to be considered per grouping value. An algorithm considering less Required tuples cannot be envisioned without putting assumptions on the input ordering (hypothesis H1).

Thus, *CompMin* is preferred to *IterMin* and *CompMax* in the Ram Lower Bound context for it minimizes the number of Required tuples.

As the Sort algorithm shares the same structure as *CompMin*, it exhibits the same number of Required tuples. Thus, under hypothesis H1, the algorithms presented in Figure 29 are all optimal with respect to the number of Required tuples that need to be considered.

The number of Required tuples has an impact on the local cost of each algorithm. For example, all Required tuples participate in the computation of grouping values that turn to be discarded both in *CompMax* and in the first iteration of *IterMin*. But beside this local overhead, Required tuples have a much more negative impact on the global cost of the whole QEP. Indeed, they generate computations from the leaves of the QEP up to the operator that requires them, without participating in the iteration's result. Minimizing the number of Required tuples during an iteration amounts to maximize the number of Irrelevant tuples that could be pruned early in the QEP. The next section develops this point.

## 4.2 Eager Pruning of Irrelevant Tuples

The distinction between Relevant, Required and Irrelevant tuples depends on each operator. Once this distinction has been made, eager pruning of Irrelevant tuples can be implemented as follows. Each operator willing to eliminate the Irrelevant tuples from its input expresses a predicate, called *iteration filter*, that selects only the Relevant and Required tuples for a given iteration. This predicate will then be checked by the operators participating in the QEP subtree producing this input. Conceptually, an iteration filter is similar to a regular selection predicate that is pushed down to the QEP to eliminate the Irrelevant tuples as early as possible. However, iteration filters may involve several attributes issued from different base relations. So, they are more complex than regular selection predicates and care must be taken on the way they are checked to avoid redundant computations. In the following, we first describe how iteration filters are expressed, then we concentrate on the way they are checked.

## 4.3 Expressing Iteration Filters

The following discussion is conducted on an operator basis. The Scan and Project operators are not concerned by the expression of iteration filters since all tuples they consider are Relevant. Regarding the Select operator, expressing an iteration filter to eliminate the Irrelevant tuples present in its input turns to delegate the selection process to another operator belonging to the QEP subtree producing the Select input. This is nothing but pushing selections down to the QEP, as usual. The Join algorithm considers many Irrelevant tuples since, at each iteration i, the Relevant tuples from the right input are only whose matching with the current tuple $t_i$ from the left input. Thus, a Join iteration filter is the instantiation of the join predicate with $t_i$. In a Left-deep QEP, a Join filter is unfortunately inoperative. Indeed, checking it in the right subtree leads to evaluate the join predicate twice per tuple. Note that Join iteration filters may keep a strong interest in a more general context.

*GroupBy filters:* in the *CompMin* variation of the GroupBy algorithm, a tuple *t* is Relevant if t.GrpLst= Current, while it is Required if t.GrpLst∈ ]Current, Split[, where Current and Split denote the state of the corresponding variables at the time the tuple t is processed. The GroupBy iteration filter, or GroupBy filter for short, is therefore a predicate of the form (t.GrpLst ≥ Current and t.GrpLst < Split). Note that the Split variable evolves during a same iteration. It takes the value $+\infty$ at the beginning of the iteration and then converges to the value immediately greater than Current. This introduces a particular form of predicate that can be termed dynamic since its selectivity increases along a same iteration.

*Sort filters:* As already stated, the RAM lower bound version of the Sort and GroupBy algorithms share the same structure. Thus, the discussion on GroupBy filters applies as well to Sort filters and need not be repeated.

## 4.4 Checking Iteration Filters

The objective is to push the evaluation of iteration filters down to the QEP in order to prune Irrelevant tuples as early as possible. The place where an iteration filter can be checked depends on the relations it involves. Mono-relation iteration filters are simply checked by the corresponding Scan operator, at the leaf of the QEP. Multi-relation filters have to be decomposed into several predicates that are checked at different levels of the QEP. Let assume an iteration filter involving the relations $R_1, R_2, \ldots R_n$ appearing in this order in the Left-Deep tree (that is, $R_1$ is the very left leaf of the QEP). This iteration filter is split into n independent predicates as follows. The first predicate applies to $R_1$ and is checked by the corresponding Scan. The $i^{th}$ predicate applies to the result of the join between $R_1, R_2, \ldots R_i$ and is checked by the corresponding join operator, and so on up to the join with $R_n$. Figure 32 shows the instantiation of this mechanism for a multi-attribute GroupBy.

Let us have a closer look on this figure and consider a Required tuple t(x,y) produced by JoinRS. Before delivering it, JoinRS checks whether this tuple satisfies the GroupBy filter, namely t.GrpLst∈ ]Current, Split[ (assuming the use of CompMin). Once delivered and considered by JoinRST, t can match with several tuples from T thereby producing a sequence of tuples of the form $\{tt_1, tt_2, \ldots tt_n\}$, from which only the first one is Required. Thus, tt1 will update the Split bound in such a way that $\{tt_2, \ldots tt_n\}$ become Irrelevant. The tuples $\{tt_2, \ldots tt_n\}$ have been produced vainly. To avoid such situation, the join algorithm has to be slightly modified in order to abort the processing of tuple t as soon as possible. This can be done by checking at each Next call from its parent that the current left tuple is still valid with respect to the iteration filter (*i.e.*, t.GrpLst < Split).

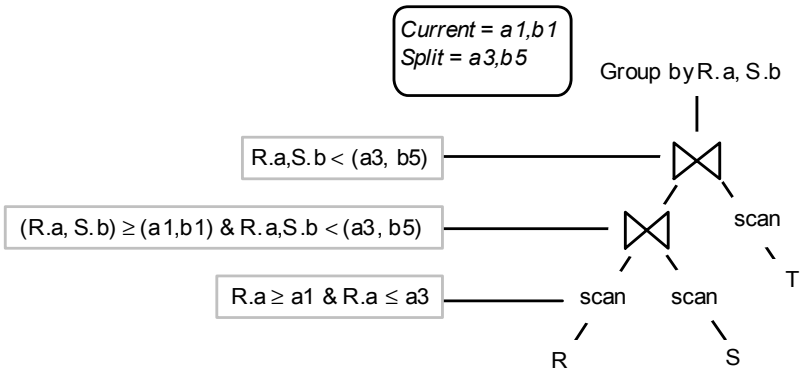



**Figure 32** *Group Filters.*

While eager pruning of Irrelevant tuples is simplified by the Left-deep shape of the QEP, it can be extended to Right-deep and Bushy trees in a straightforward fashion.

# 5 The Impact of RAM Incremental Growths

This section revisits the query evaluation and optimization techniques devised in a RAM lower bound context when a small quantity of RAM is added to the model.

## 5.1 Impact on the Operator's Algorithms

Let us first consider the impact of a RAM incremental growth on the design rules introduced in Section 3.1. Rule R1 remains unchanged because variability plays against any memory bound. While RAM growth is considered, the RAM remains bounded by a small value. By incremental growth, we are expressing a slow deviation from the RLB bound, up to reach the value satisfying the application's constraints. Rule R2 could be reformulated as follows "recompute the information that cannot be stored in the bounded RAM". While this design rule seems obvious, it means that the philosophy of the algorithms remains unchanged, that is remains based on iterating – less frequently – on the operator's input(s).

Let us now consider the impact of additional RAM on each operator. The goal is to exploit RAM to materialize intermediate results, thereby reducing the number of iterations required on the operator's input(s). Scan, Select and Project operators are insensitive to the RAM size since they implement a single iteration on their respective input.

The Join algorithm benefits from additional RAM in a straightforward fashion. The nested-loop algorithm evolves to a block nested-loop, dividing the number of iterations on the right input by the number of buffer entries allocated to the join (assuming the buffered input be the left one). Tuple comparisons can also be saved by sorting or hashing the content of the buffer. In our context where the buffer is small, sorting is preferred to hashing since hashing consumes RAM on its own and would then decrease the useful part of the buffer.

The three variations of the GroupBy algorithm can benefit from a buffer. Let us first consider the CompMin algorithm. Exploiting the RAM available leads to divide the buffer into three arrays of a same cardinality *b*, called *Current[]*, *Split[]* and *Agg[]*. At each iteration (except the first one), the algorithm computes into Agg[] the *b* aggregations corresponding to the grouping values referenced by Current[] and searches into Split[] the *b* grouping values to be processed next. These next grouping values are determined thanks to a converging process, as in the RAM lower bound context. At a given iteration, all input tuples t such that t.GrpLst > max(Current[]) and t.GrpLst < max(Split[]) and t.GrpLst $\notin$ Split[] are Required. Indeed, each time an input tuple falls into this interval, it is inserted in sorted order into Split[] and the highest value of Split[] is discarded, thereby decreasing the upper bound of the interval. At each iteration, the algorithm considers at most one Required tuple per distinct grouping value remaining to be processed, as in the RAM lower bound context. However, since the number of iterations is divided by *b*, the total number of Required tuples considered by the algorithm is much less than in the RAM lower bound context. The CompMax algorithm can be buffered in the same way (*i.e.*, by changing variables into arrays) except that a single Split variable is necessary to reference, at a given iteration, the highest grouping value previously computed. With buffering, the gap between CompMax and CompMin in terms of Required tuples increases since at each iteration, many aggregation calculus are partially performed before being discarded. On the other hand, CompMax produces less iterations than CompMin since more space is left to the Current[] array in the buffer. This makes the CompMax algorithm more efficient when iteration filters are not considered. The buffered extension of IterMin is not discussed since it has the same memory requirements as CompMin and considers more Required tuples.

The buffered adaptation of the Sort algorithm shares some similarities with CompMax. The buffer is divided into an array *Current[]* of b' entries dedicated to the storage of the smallest tuples to be delivered at a given iteration and *Split*, a single variable referencing the highest sorting value encountered at the previous iteration. At the first iteration, the algorithm scans its inputs and inserts the tuples in ascending order into Current[]. When Current[] overflows, the highest tuple in the sort order is discarded. At the end of this

iteration, Current[] contains the b' smallest tuples corresponding to a sequence of sorted values of the form $v_1v_1<v_2v_2< \ldots v_{n-1}v_{n-1}<v_nv_nv_n$ (this sequence expresses the presence of duplicates wrt the Sort condition). All Current[] tuples having a sorting value in the range $[v_1,v_{n-1}]$ are then delivered in the sort order and Split is set to $v_n$ (see Figure 33). At the next iteration, tuples of the input having a sorting value less than $v_n$ are not considered, tuples sharing the value $v_n$ are delivered and tuples having a sorting value greater than $v_n$ are inserted in ascending order into Current[], and so on. At each iteration all input tuples t such that t.SortLst∈ [Split, max(Current[])[ can be either Relevant or Required and their status is actually determined a posteriori, at the time Split is reset (*i.e.*, at iteration end). The Sort algorithm takes less benefit from a RAM growth than the GroupBy. This is due to the fact that tuples sharing the same sorting value have to be delivered together instead of being aggregated. Consequently, the performance of the Sort algorithm is driven by the number of duplicates wrt the sort condition.

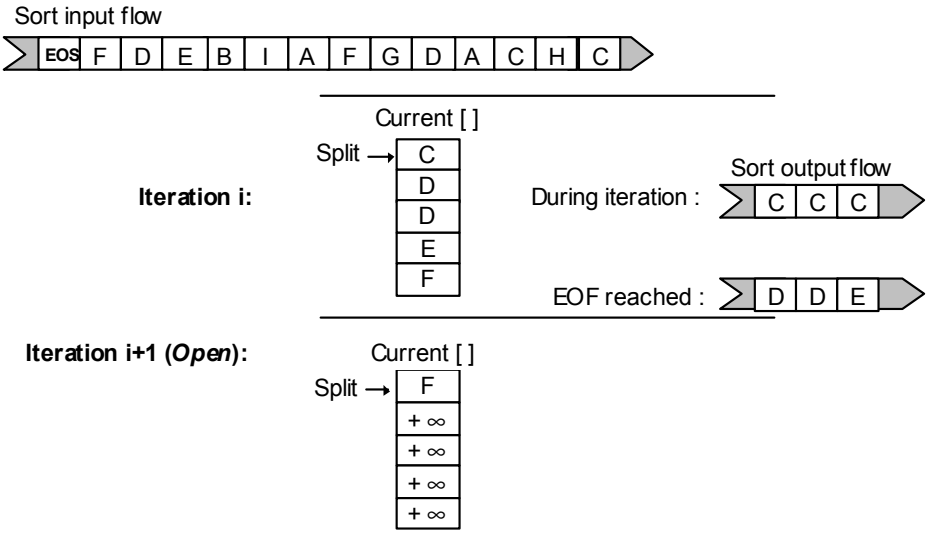



**Figure 33** *Buffered Sort.*

## 5.2 Impact on Iteration Filters

We focus below on the filter predicates expressed by the buffered version of the GroupBy and Sort algorithms. From the preceding section, it turns out that the predicate expressing a GroupBy filter for the CompMin algorithm takes the form (t.GrpLst∈ Current[] or (t.GrpLst > max(Current[]) and t.GrpLst < max(Split[]) and t.GrpLst ∉ Split[])). We do not discuss the form of GroupBy filters for the CompMax algorithm since CompMin is always preferred when iteration filters are used. The predicate expressing a Sort filter takes the following form (t.SortLst∈ [Split, Max(Current[])[). The predicates expressing these iteration filters are more costly to check than their RAM lower bound counterpart, although they are less frequently checked. Indeed, the inclusion of a tuple into an interval has now to be evaluated. Note that this evaluation can be accurate (exact match) or fuzzy (only the bounds are checked). The

cost of an accurate evaluation is decreased by the fact that the GroupBy and Sort buffers are kept sorted.

Let us now consider the modification made on the join algorithm to check the filter predicates accurately (cf. Section 4.3). Irrelevant tuples should now be discarded from a Join buffer as soon as the upper-bound of the filter predicate evolves. This leads to check all tuples present in the join buffer, on each Next call issued by the Join parent in the QEP. A less accurate alternative is to check only the current left tuple being considered. This simpler alternative has been shown more efficient by our performance evaluations.

# 6 Performance Evaluation

The first objective of this evaluation is to assess the pertinence of the algorithms proposed in this chapter, by quantifying the time required to execute different types of queries under strong RAM constraints. The expected outcome is to precisely evaluate to which extent these algorithms constitute an alternative to the use of indices. The second objective is co-design oriented. The expected outcome is here to provide valuable co-design guidelines by the means of curves helping to find the best compromise between RAM capacity, volume of on-board data, query complexity and expected response time.

## 6.1 Experimentation Platform

To conduct these experiments, we have implemented a complete query evaluator complying with the design principles introduced in this chapter. The operator's algorithms are the ones presented in Section 5.1. Regarding the GroupBy algorithm, CompMin is used when iteration filters are activated and CompMax is used otherwise. The QEPs of interest are optimized according to the heuristics described in this chapter. Thus, they take the form of Left-deep trees where joins are ordered according to their respective selectivity and to the cardinality of the relations involved. The RAM is distributed on the resulting QEP thanks to a simple (and preliminary) cost model.

Counters are introduced in the platform to capture the number of elementary operations performed during the execution of a QEP (*e.g.*, read a RAM cell, read a stable storage cell, evaluate a predicate …). These counters allow us to calibrate our prototype in order to reflect the behavior of different target hardware platforms (*e.g.*, in terms of processor speed or stable storage technology). For this study, the platform has been calibrating with the following values: RAM read time = 50 ns, Stable storage read time = 100 ns, Processor speed = 50 MHz. These values correspond to advanced smart card prototypes and are representative of the smart card technology that will be available in the near future (two to four years).

## 6.2 Data, Queries and Experiments

In the scope of these experiments, we consider an on-board database composed of four base relations named *R*, *S*, *T* and *U*. Each relation contains at least three attributes: an integer primary key attribute, a string non-key attribute on which apply GroupBy and Sort operations and a string non-key attribute complementing the tuple to reach an average size of 100 bytes. In addition, foreign key attributes are added into *S* to reference *R* and into *T* and *U* to reference S. These relations are populated by a pseudo-random generator allowing us to generate either a uniform or a Zipfian (*i.e.*, skewed) distribution of the data. The tuple cardinality of each relation, for a scale factor SF=1, is: $\|R\| = 100$, $\|S\| = 300$, $\|T\| = 1200$ and $\|U\| = 600$, leading to a 220 kilobytes database.

We consider different types of queries, summarized in Table 1, of increasing complexity. Queries Q1 to Q5 are named *Regular* for they are representative of usual queries that we could foresee in an embedded context (sensors, personal folders, ambient intelligence). The simplest Regular query (Q1) computes a single join while the most complex ones (Q4 and Q5) compute two joins followed by a GroupBy or a Sort. To make the study complete, we consider also *Complex* queries (Q6 and Q7) involving three joins and a multi-attribute GroupBy or Sort.

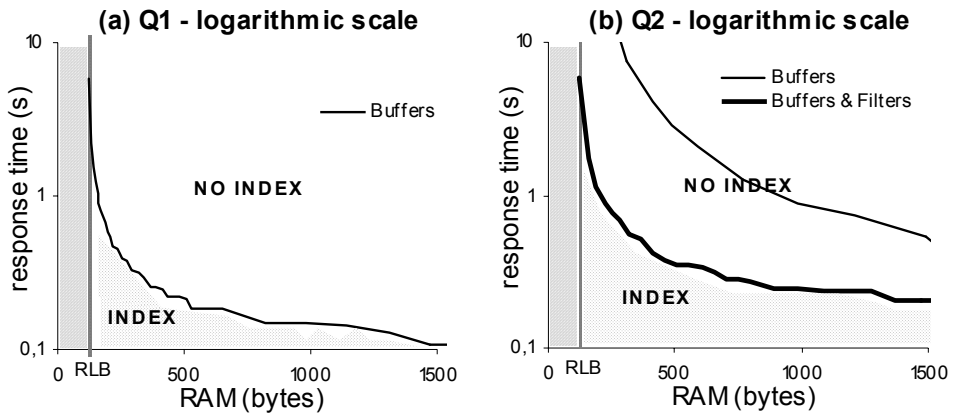
## 6.3 Interpretation of the Results

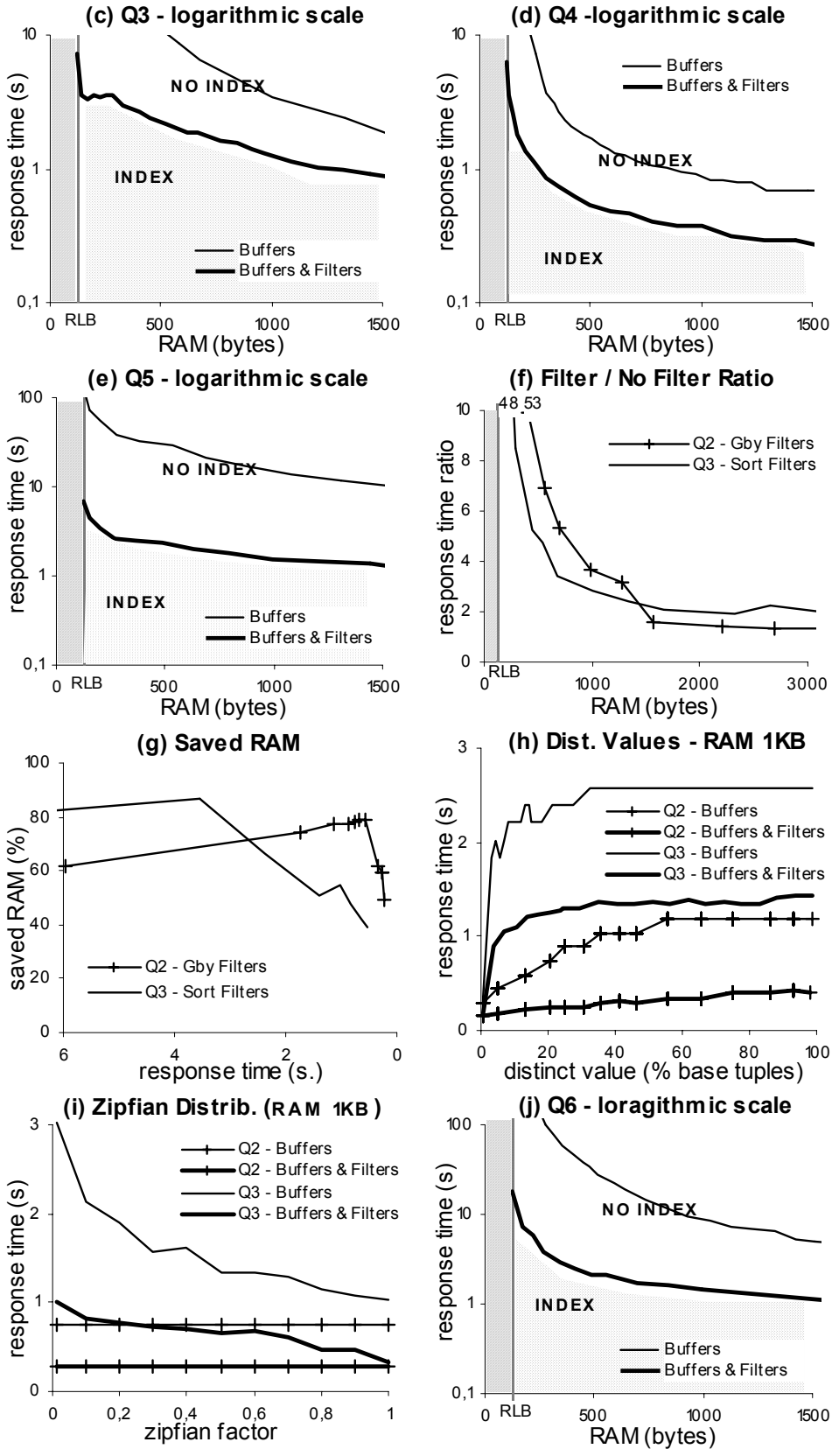
Let us first study how the query evaluator behaves in the presence of Regular queries. The curves plotted in Figure 34(a) to Figure 34(e) represent the respective execution time of queries Q1 to Q5 as a function of the RAM. On each figure, the plain curve represents the execution time required to execute the query without iteration filters while the bold curve integrates the benefit provided by iteration filters. These two curves divide the space into three areas. The area above the plain curve materializes all combinations of response time (RT) and available RAM that can be reached by exploiting our operator's algorithms, without iteration filters nor index. The area delimited by the two curves represents the RT/RAM combinations that become accessible by adding iteration filters to the preceding algorithms. Finally, the gray area represents the RT/RAM combinations that are unreachable without index. A fourth area on the left end side of each figure shows the combinations that can never be considered since they are located under the RAM lower bound. These curves deserve two important remarks. First, the hyperbolic shape of the curves shows that the operator's algorithms exploit very well any RAM incremental growth. To illustrate this, the time required to execute Q2, without index nor iteration filter, amounts to 75 seconds with 150 bytes of RAM and falls down to 12 seconds with an addition of only 100 bytes of RAM. Second, the iteration filters strongly enlarge the area where the resort to indices can be avoided. Typically, the quantity of RAM required to execute Q3 in 1 second without iteration filters is 2.5 kilobytes and falls down to 1.2 kilobytes when iteration filters are exploited. For queries Q4 and Q5, the benefit of iteration filters seems graphically less

impressive but this feeling is only due to the logarithmic scale of the figure. For example, executing Q5 in 1 second requires more than 16 kilobytes of RAM while 2 kilobytes suffice when iteration filters are used.

The main conclusion of this first series of experiments is that the combination of our operator's algorithms with iteration filters constitutes a very convincing alternative to the use of indices for the considered queries. Note that, thanks to these techniques, Q1 to Q5 can all be executed with a response time close to 1 second (the worst case being 1,4 second for Q5) with only 1 kilobytes of RAM. This result is particularly significant considering that the domain of investigation delimited by a response time around 1 second and a RAM around 1 kilobytes seems to be very challenging. Indeed, 1 second represents a "psychological" barrier reflecting well the requirement of most interactive applications. 1 kilobytes of RAM could however be considered as a two extreme bound and one may wonder whether not to consider more comfortable assumptions regarding the RAM resource. The first reason is technological. Today's ultra-light devices like smart cards are equipped with 1 to 4 kilobytes of RAM (for the most powerful ones) but only a few hundred of bytes is left to the on-board applications, the rest being consumed by the OS, the JVM and the execution stack. Thus, 1 kilobytes of RAM allocated for query processing is today a rather optimistic assumption and semiconductor manufacturers do not forecast a rapid growth of the RAM resource for several reasons like reducing the silicon die size, the power consumption and the security threats. The second reason is economic and leads to lower all hardware resources (among them the RAM) to their minimum in order to save production costs on large-scale markets. So if more RAM is not expressly required, it will not be provided.

| Query | Query Type | Output Tuples / Dist. Values |
|---|---|---|
| *Q1* | Join(S, T) | 1200 |
| *Q2* | GroupBy(S.a, join(S, T)) | 60 / 60 |
| *Q3* | Sort(S.a, join(S, T)) | 1200 / 60 |
| *Q4* | GroupBy(R.a, Join(R, S, T)) | 1200 / 20 |
| *Q5* | Sort(R.a, join(R, S, T)) | 1200 / 20 |
| *Q6* | GroupBy(R.a, S.b, join(R, S, T, U)) | 200 / 200 |
| *Q7* | Sort(R.a, S.b, join(R, S, T, U)) | 2400 / 200 |




(c) Q3 - logarithmic scale
response time (s)
NO INDEX
INDEX
Buffers
Buffers & Filters
RLB
RAM (bytes)
(d) Q4 -logarithmic scale
response time (s)
Buffers
Buffers & Filters
NO INDEX
INDEX
RLB
RAM (bytes)
(e) Q5 - logarithmic scale
response time (s)
NO INDEX
INDEX
Buffers
Buffers & Filters
RLB
RAM (bytes)
(f) Filter / No Filter Ratio
48.53
response time ratio
Q2 - Gby Filters
Q3 - Sort Filters
RLB
RAM (bytes)
(g) Saved RAM
saved RAM (%)
Q2 - Gby Filters
Q3 - Sort Filters
response time (s.)
(h) Dist. Values - RAM 1KB
response time (s)
Q2 - Buffers
Q2 - Buffers & Filters
Q3 - Buffers
Q3 - Buffers & Filters
distinct value (% base tuples)
(i) Zipfian Distrib. (RAM 1KB)
response time (s)
Q2 - Buffers
Q2 - Buffers & Filters
Q3 - Buffers
Q3 - Buffers & Filters
zipfian factor
(j) Q6 - loragithmic scale
response time (s)
NO INDEX
INDEX
Buffers
Buffers & Filters
RLB
RAM (bytes)

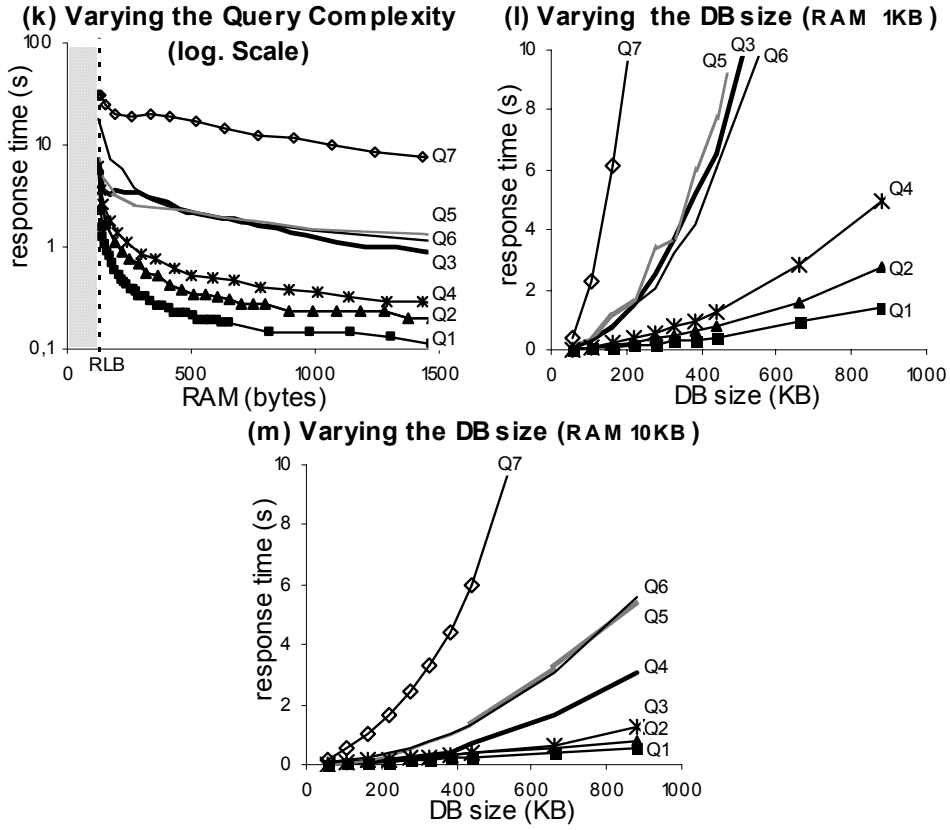



**Figure 34** *Evaluation Results.*

Figure 34(f) and Figure 34(g) gives another insight into the gain brought by iteration filters. Figure 34(f) expresses the ratio between the execution times obtained without and with iteration filters for the two queries Q2 and Q3 as a function of the RAM. Not surprisingly, the lower the RAM the higher the ratio since the RAM determines the number of iterations performed by the GroupBy and Sort algorithms. Note that this ratio would be even greater with queries Q4 and Q5 since the GroupBy and Sort algorithms would reiterate on a bigger subtree. The highest benefit is measured in the range [RLB, 1.5 kilobytes] of RAM. This motivates further the use of iteration filters when the RAM resource of a device has to be minimized. Figure 34(g) plots the percentage of RAM saved in the execution of Q2 and Q3 by iteration filters as a function of the expected execution time. For example, the amount of RAM required to execute Q2 in less than 1 second is 200 bytes with iteration filters and 940 bytes without iteration filters, leading to a gain of 78% of RAM.

Figure 34(h) and Figure 34(i) evaluate the robustness of our algorithms against an increase of distinct values and skewed data. The evaluation of Q2 and Q3 is measured with and without iteration filters for a given amount of RAM of 1 kilobytes. As shown by Figure 34(h), the use of iteration filters makes the operator's algorithm more stable when facing more distinct values. This phenomenon is due to the eager elimination of the Irrelevant tuples that participate in an increasing number of iterations. Figure 34(i) shows that the GroupBy algorithm is insensitive to skewed data, the number of iterations remaining

constant. The Sort algorithm takes advantage of skewed data because several sorting values shared by few tuples can be processed at the same iteration.

Finally, the last four figures evaluate to which extent our algorithms scale when the query complexity or the volume of data augments. Figure 34(j) illustrates the benefit of iteration filters for a complex query involving 3 joins and a multi-attribute GroupBy. For 1 kilobytes of RAM, iteration filters reduce the cost of the execution from 9,6 to 1,4 seconds. Clearly, the more complex the query, the more efficient the iteration filters. Figure 34(k) plots the execution time of the filtered execution of all queries of interest as a function of the RAM. This figure demonstrates that the proposed algorithms scale well when they face complex queries. Indeed, all queries except Q7 can be executed around one second (the worst case being 1,4 second for Q5 and Q6) with only 1 kilobytes of RAM. Figure 34(l) plots the execution time of the filtered execution of all queries as a function of the database size for a fixed quantity of RAM of 1 kilobytes . The first learning of this figure is that our algorithms scale pretty well for Regular queries. However, they scale badly in the presence of Sort or in the presence of several joins. This situation was predictable and one cannot expect execute complex queries on a large database without index and with only 1 kilobytes of RAM in less than 1 second. Two ways can be investigated to decrease the query execution time, namely adding more RAM or adding indices. Figure 34(m) plots the same curves with a larger RAM of 10 kilobytes and shows that the problem becomes less critical without totally disappearing and that complex queries involving Sort are still not tackled. Adding indices is the way followed by PicoDBMS [PBV+01]. It solves the performance problem at the price of the side effects mentioned in Section 2.

As a conclusion, these experiments show the accuracy of the proposed operator's algorithms along with their iteration filters and demonstrate that they constitute a real alternative to the index in a wide range of situations. Beyond this range, indices should be considered. Finally, note that all the curves presented in this section can be used for co-design purpose. Indeed, they provide valuable information to determine: whether indices or iteration filters are required in a given situation, how much RAM should be added to reach a given response time, how much the expected response time should be relaxed to tackle a given query with a given quantity of RAM and how much data can be embedded in a given device without hurting an expected execution time.

# 7 Conclusion

Pervasive computing and ambient intelligence motivate the development of new data-centric applications that must be tackled in a growing variety of ultra-light computing devices. As far as query execution is concerned, RAM appears to be the most critical resource in these devices. In the absence of a precise understanding of the RAM consumption problem, ad-hoc solutions have been developed. Most of them introduce strong restrictions on the type of

queries and on the amount of data that can be tackled while the others resort to dedicated index methods having negative side effects. This introduces the need for pre-designed database components that can be integrated in Systems on Chip.

This study precisely addresses this issue and proposes a framework helping to design RAM-constrained query evaluators. First, we proposed a query execution model that reaches a lower bound in terms of RAM consumption. Second, we devised a new form of optimization, called iteration filter, that drastically reduces the prohibitive cost incurred by the preceding model, without hurting the RAM lower bound. Third, we proposed variations of the preceding techniques that best exploit any incremental growth of RAM. Our performance evaluations led to two important and practical outcomes. First, they show the accuracy of the proposed techniques and demonstrate that they constitute a convincing alternative to the index in a wide range of situations. Second, they provide helpful guidelines helping to calibrate the RAM resource of a hardware platform according to given application's requirements as well as to adapt an application to an existing hardware platform.

# Chapter 4 – PicoDBMS Study

*This chapter focuses on the categorization of the different access rights required in the smart card's context and on the performance measurements of the embedded DBMS code with a relatively large databases (MB) and complex access rights requirements.*

## 1 Introduction

Smart cards are used in many applications exhibiting high security requirements (see Chapter 2, Section 1.1.5) like identifications, personal folders (*e.g.*, healthcare or insurance data), user profiles (bookmarks, passwords, configurations, agenda, address book, etc.). In data centric smart card applications, confidential data resides in the secured environment of the chip to inherent from its intrinsic security. Moreover, database components must be embedded on-board to prevent from any information disclosure to occur. These components manage the embedded data and only externalize authorized results to the connected user.

As many smart objects, smart card hardware architecture is strongly constrained. Today, smart card's microcontrollers must fit in 25 mm² due to the requirements for tamper resistance requirements and to ensure the chip will not be broken or loosened if the plastic card is bent. The peculiarities of this environment induce a specific design for embedded database components. In a preliminary study, database techniques have been scaled down for the smart card context leading to the PicoDBMS proposal [PBV+01].

Based on this design, a first prototype written in JavaCard has been built and demonstrated to the VLDB'01 conference [ABB+01]. While this prototype was fully functional, its performance was not satisfactory. We then conducted experiments on a second prototype, written in C, and running on a more advanced Axalto smart card equipped with the operating system ZePlatform. Our experiments highlighted important bottlenecks in ZePlatform when dealing with intensive fine grain accesses to the stable memory, which is a rather unusual case in traditional smart card's applications. Thus, Axalto modified internals of Zeplatform[28], providing a much better support for PicoDBMS. The performance gain obtained thanks to this modification of ZePlatform and to the rewriting of PicoDBMS from JavaCard to C is roughly two orders of magnitude. Therefore, we only recently obtained the

[28] Since operating systems are masked in ROM, any change induce a time consuming engineering process.

PicoDBMS performance measurements while the theoretical bases of PicoDBMS were set-up four years ago.

The contribution of this chapter is twofold. First, it categorizes the different access rights requirements that may be defined on any kind of embedded data, with the objective to generalize the PicoDBMS approach. Second, it makes an in-depth performance evaluation of the different aspects of the query evaluation required to match the preceding access right requirements (storage, indexation, operators and query plans). It validates their effectiveness on a real smart card prototype and on a cycle accurate hardware smart card's simulator, which allows conducting prospective evaluations on larger databases and drawing interesting conclusions. The next section formulates the problem and details the outline of the chapter.

## 2 Problem Statement

In the PicoDBMS context, confidential data is confined inside the chip, and thus benefits from the smart card tamper resistance (see Chapter 2, Section 1.1.3). Indeed, the hardware properties of the chip give to data at rest a very high security level. While this prevents from physically tampering the embedded data, it also forces to embed a specific code that externalizes only authorized data to the user. This embedded code can thus be viewed as a doorkeeper; it must authenticate the users and solely deliver the data corresponding to their access rights.

In the following, we do the assumption that the identification/authentication of the user are ensured traditionally (*e.g.*, by a login/password) as well as the authorization rules are, *i.e.*, by granting users access to database views. The point is now to make clear the kind of access rights that must be supported by a PicoDBMS-like system. To this end, we define three main categories of authorizations, of increasing complexity, covering a wide range of situations. For the sake of generality, these categories of authorizations are expressed below wrt an entity relationship database schema: *(i) Schema authorizations (SA)* are defined on the database intension (schema) without considering the database extension (occurrences); *(ii) Occurrence authorizations (OA)* grant access to some occurrences, depending on predicate(s) on its properties, or on the existence of a relationship (either direct or transitive) with another granted occurrence; and *(iii) Computed data authorizations (CA)* grant access to computed values without granting access to the occurrences taking part in the computation process (*e.g.*, averaged values are disclosed but not the individual records used to compute them).

On a relational database, and taking into account the requirements of the applications described in Section 2.2, we distinguish seven access rights types derived from the SA, OA and CA classes. They are detailed in Table 9. SA may be implemented using the classical grant/deny SQL instructions on the database relations and using views including the project operator (P access right in Table 9). The predicates of OA can apply on attributes of one

relation (SP access right), two relations (direct relationship, SPJ access right), or several relations (transitive relationship, $SPJ^n$ access right). Finally, CA are implemented in SQL by means of views including aggregate computation. The aggregation may be mono-attribute (*i.e.*, group on a single attribute) and consider a single relation (SPG access right), mono-attribute and consider several relations (SPJG access right), or multi attributes, potentially considering several relations ($SPJG^n$ access right).

| *Acro-nym* | *Auth°* | *Access right type* | *Required database operator(s)* | *View example* |
|---|---|---|---|---|
| P | SA | Subset of attributes | Project | Name of doctors |
| SP | OA | Predicates on a single relation | Select, Project | Name of non-specialist doctors |
| SPJ | OA | Predicates on two relations (direct relationship) | Select, Project, Join | Name of doctors the patient has visited last month |
| $SPJ^n$ | OA | Predicates on several relations (transitive relationship) | Select, Project, Joins | Name of doctors who prescribed antibiotics to the patient |
| SPG | CA | Single relation, mono-attribute aggregation | Select, Project, Group (aggregate) | Number of doctor per specialty |
| SPJG | CA | Several relations, mono-attribute aggregation | Select, Project, Join, Group (aggregate) | Number of visits per doctor's specialty |
| $SPJG^n$ | CA | Multi-attribute aggregation | Select, Project, Join, Group (aggregate) | Number of visits per doctor's specialty and per year |

**Table 9** *Description of Access Rights on Embedded Data.*

A trivial solution to provide the access rights functionalities described above is to embed a file system. Each view is materialized in a separate file and users are granted access to the appropriate file subset. This solution would endow very simple embed code. However, it badly adapts to data and access right updates and results in a huge data replication among files, thereby hurting the smart card limited storage constraint. Thus, a dynamic evaluation of the authorized views must be considered as a prerequisite in the problem statement. This implies to embed on chip the ability to compute database operators and run query execution plans.

The code embedded in the smart card must comply with the following design rules derived from the smart cards properties (detailed Chapter 2, Section 1.2):

- *Compactness rule*: minimize the size of the data structures and the PicoDBMS code to cope with the limited stable memory area.
- *RAM rule*: minimize the RAM use given its extremely limited size.
- *Write rule*: minimize write operations given their dramatic cost (≅10 ms/word).

- *Read rule*: take advantage of the fast read operations (≅100 ns/word).
- *Access rule*: take advantage of the low granularity and direct access capability of the stable memory for both read and write operations.
- *Security rule*: never externalize private data from the chip and minimize the algorithms' complexity to avoid security holes.
- *CPU rule*: take advantage of the over-dimensioned CPU power, compared to the amount of data.

The problem is therefore to comply with these rules while designing an embedded PicoDBMS capable of evaluating dynamically, on the embedded data, the seven classes of view expressions listed in Table 9.

In this chapter, we focus on the different aspects of the view evaluation in a smart card: storage, indexation, operators and query plans. We do not discuss others aspects, except when they infer in the evaluation process (*e.g.*, transaction processing), in which case we give the minimum information required for the understanding. Note that a user may issue a query on one or several authorized views. The evaluation of the query predicates and the views composition might be achieved out card (*e.g.* on the terminal) without hurting the security rule.

The chapter is organized as follows. Section 2 discusses the converging research efforts conducted in the embedded database systems area. Section 3 describes the architecture of PicoDBMS, the internal data and indices organization, query processing adaptation to the smart card constrained environment, and finally sums up the way transaction processing is ensured (since it can infer with query processing). Section 4 presents the in depth performance analysis of the prototype. It first gives the objectives of the experimentation and exposes the chosen metric, then discusses the code footprint of each embedded database component as well as the data footprint for each considered data structures, and finally analyzes the performance of the query execution considering several storage models. As a conclusion, Section 5 puts into practice the conducted performance measurements and interprets the preceding results on two applications' profiles representative of the smart card context.

# 3 Related Works

From the early 90's, the need for data management appears in several applications running on small devices, from industrial controllers and cable-television set-top boxes to medical-imaging systems and airplanes flight controls. This motivates manufacturers of embedded solutions to focus on the development of embedded DBMSs. The objective of embedded DBMSs is to provide efficient data access [SKW92, Hey94], transactional features (ACID),

while cohabiting with others applications (especially the one using the database) in the device. The embedded DBMS design has been driven so far by code footprint minimization, obtained by simplification and componentization, portability on multiple devices / operating systems, and self-administration [ChW00]. Generally, embedded DBMS are dynamically linked with the embedded application. Few academic studies have been conducted in this area [BaT95, Ols00, Ort00], but many commercial products exist like *Pervasive SQL 2000* [Per02b], *Empress database* [Emp00, Emp03], *Berkeley DB* [SeO99, OBS99], and *BirdStep*.

In parallel, the rapidly growing number of mobile phones, PDAs, and other portable consumer devices, stimulates the main DBMS editors to propose light version of their DBMS to meet this promising market [Ses99, Sel99]. Thus, the well-known *Sybase SQL Anywhere Studio* provides an Ultra-Lite deployment option [Syb99, Syb00] for its *Adaptive Server Anywhere* database. Also, light versions of popular DBMS like *IBM DB2 Everyplace* [IBM99, KLL+01], *Oracle Lite* [Ora02a, Ora02b] and *Microsoft SQL Server for Windows CE* [Ses00] have been designed for small devices. The DBMS editors pay a great attention to data replication and synchronization with a central database in order to allow offline executions and resynchronization at reconnection time. Again, the small footprint is obtained by simplifying the DBMS code and by a selective linking of the DBMS code with the application code. Light DBMSs are portable to a large variety of small devices and operating systems, such as *PalmOs*, *WinCE*, *QNX Neutrino* and *Embedded Linux*. While addressing different market shares and having followed different development paths, it is now quite difficult to clearly distinguish embedded DBMSs from light DBMSs (*e.g.*, Sybase Anywhere) since the former are adding incrementally DBMS features (transaction support, SQL management, etc.) while the latter are removing unnecessary features to minimize the code footprint.

By considering smart cards as traditional small devices, one could envision using embedded DBMSs or Light DBMSs technology in this environment. However the primary objective of smart cards being security, smart cards have a very specific hardware architecture. The hardware resource asymmetry pointed out in Section 2.3 entails a thorough re-thinking of the techniques currently used for embedded / light DBMSs.

Sensor networks gathering weather, pollution or traffic information have motivated several recent works related to on-chip data management. *Directed Diffusion* [IGE00], *TinyDB* [MHH04], and *Cougar* [YaG02] focus on the way to collect and process continuous streams of sensed data. Optimization techniques for queries distributed among sensors are investigated in [BGS01, BoS00]. Techniques to reduce power consumption induced by radio frequency communications, one of the major sensor's constraints, are proposed in [MaH02]. Although sensors and smart cards share some hardware constraints, the objectives of both environments diverge as well as the techniques used to meet their respective requirements. Basically, the on-chip database capabilities required in a sensor are limited to data filtering and aggregation to reduce the output flow of the sensor and thus saving energy [MFH+02]. More complex query executions, as the one mentioned in Table 9, are useless in the sensor

context. In addition, response-time constraints are strongly different in both cases. Depending on the usage of the sensor network, time constraints may be either inexistent or hard real-time while time constraints in the smart card context are related to the user interaction. Finally, sensors often compute queries on streaming data or on a single local embedded relation while a smart card PicoDBMS may have to manage several relations.

A small number of studies have addressed specifically the area of data management on smart cards. The first attempts towards a smart card DBMS were ISOL's SQLJava Machine DBMS [Car99] and the ISO standard for smart card database language, SCQL [ISOp7]. Both were addressing generation of smart cards endowed with 8 kilobytes of stable memory. While SQLJava Machine and SCQL design was limited to mono-relation queries, they exemplify the strong interest for dedicated smart card DBMS. More recently, the MasterCard Open Data Storage initiative (MODS) [Mas03] promoted a common API to allow retailers, banks and other organizations to access and store data on users' smart cards with an enhanced security for the smart card's holder. However, MODS is based on flat files, crude access rights (*i.e.*, file level access rights), and thus, does not provide neither compact data model nor query processing. However, these initiatives show the interest of developing DBMS techniques for these environments.

A recent study proposes specific storage techniques to manage data in flash memory on a smart card. The design is also limited to mono-relation queries and is strongly impacted by the physical characteristics of the target smart card architecture. Indeed, they propose to store the data in NOR type of FLASH memory generally dedicated to store programs as ROM replacement. Since updates in NOR flash memory are very costly, (updating a single data induces large and costly bloc erasure), the techniques are driven by update cost minimization (deleted bit and dummy records [BSS+03]). While this study shows the impact of hardware characteristics on the DBMS internals, it does not comply with the memory constraints of the smart card nor address complex query processing, mandatory to extract the authorized part of the data.

Finally, we should remark that *GnatDB* [Vin02], presented as a "small-footprint, secure database system" may fit in a smart card since it has an 11 kilobytes footprint. In fact, GnatDB was designed to manage a small amount of external data, the smart card being used to protect remote data against accidental or malicious corruption. GnatDB thus does not address query processing nor internal data storage and indexation.

# 4 PicoDBMS at a Glance

PicoDBMS is composed of the following main modules, stored on-chip: a storage manager organizing data and indices into the chip's stable memory, a query manager building evaluation's plans (including selections, projections, joins and aggregates operators) and capable of executing rather complex queries on-chip, a transaction manager enforcing the

atomicity of a sequence of updates and finally an access right manager providing grant and revoke functionalities on the defined user views and preventing from externalizing unauthorized data. Other modules are stored on the terminal, within the JDBC driver interfacing the user application with the embedded database engine. User's authorized database schema is externalized at connection time, allowing the JDBC to parse the query and verify the SQL syntax. As well, the *order by* clause, if any, is evaluated on the terminal. Note that these remote treatments do not hurt the security rule since they only involve authorized information. Thus, PicoDBMS allows confidential data sharing among several users connecting to the database one at a time. Figure 35 pictures the PicoDBMS architecture.

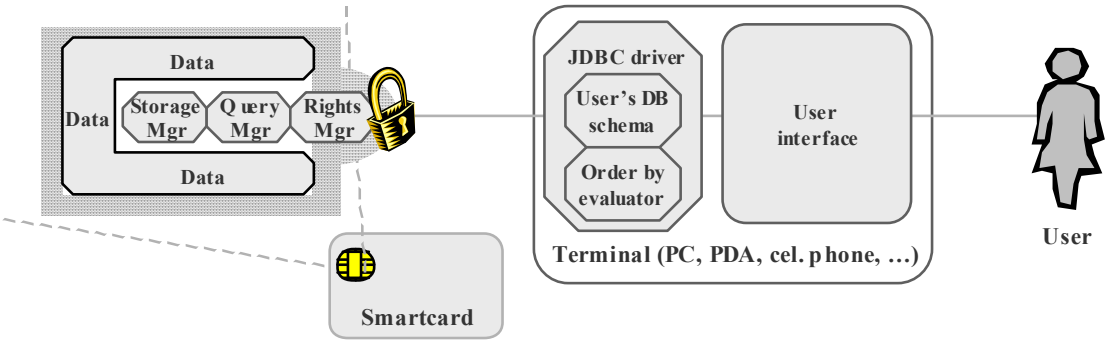



**Figure 35** *PicoDBMS Architecture.*

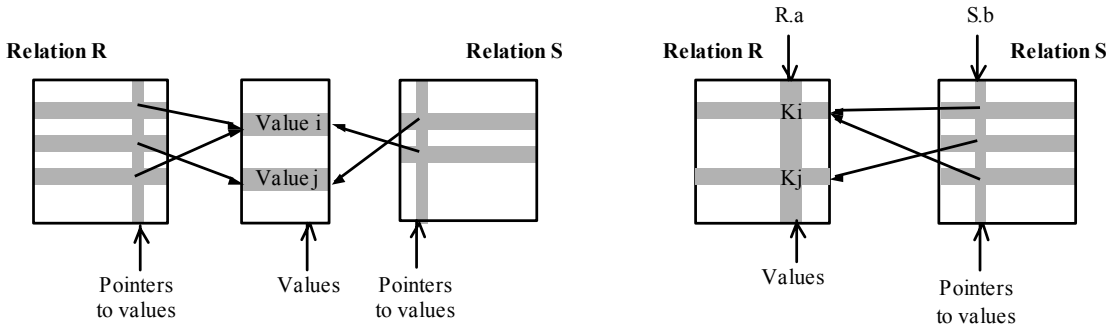
In the following, we present the concepts strictly needed for the well understanding of the access right evaluation's process, namely the storage and indexation model, the relational algebra operators and the query plans evaluation. We also briefly present the transaction management techniques (even though there are out of the scope of this study) because they interfere in the performance measurements and in the code footprint. So, transactions must be taken into account for a fair assessment of the global performance and applicability of PicoDBMS. For a more detailed description of PicoDBMS, we refer the reader to [PBV+01].

## 4.1 Storage and Indexation

The simplest way to organize data is *Flat Storage (FS)*, where tuples are stored sequentially and attribute values are embedded in the tuples. The main advantage of FS is access locality. However, in our context, FS has two main drawbacks. First, this storage organization is space consuming. While normalization rules preclude attributes conjunction redundancy to occur, they do not avoid attribute value duplicates. Second, it is inefficient, inducing the sequential computation of all operations in the absence of index structures. While this is convenient for old fashion cards (some kilobytes of storage and a mono-relation select operator), this is no longer acceptable for future cards where storage capacity is likely to exceed 1 megabytes and queries can be rather complex. Adding index structures to FS may solve the second problem while worsening the first one (*e.g.* values are stored in the

additional index structure and in the flat tuples as well). Thus, FS alone is not appropriate for a PicoDBMS. One can notice that this storage model is sometimes inevitable, as considered in Chapter 3.

Based on the critique of FS, it follows that a PicoDBMS storage model should guarantee both data and index compactness. Since locality is no longer an issue in our context (read rule), pointer-based storage models inspired by MMDBMS [MiS83, AHK85, PTV90] can help providing compactness. The basic idea is to preclude any duplicate value to occur. This can be achieved by grouping values in domains (sets of unique values). We call this model *Domain Storage (DS)*. As shown in Figure 36, tuples reference their attribute values by means of pointers. Furthermore, a domain can be shared among several attributes. This is particularly efficient for enumerated types, which vary on a small and determined set of values[29].




*(a) Domain Storage of a Regular Attribute* *(b) Domain Storage of Foreign-key Attribute*

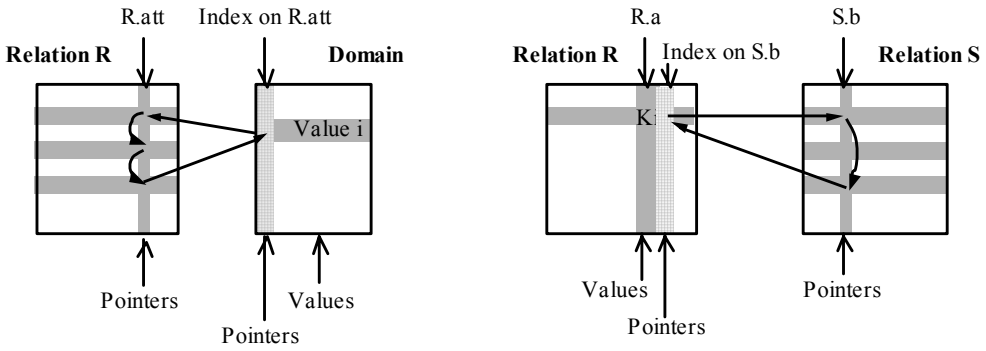
**Figure 36** *Domain Storage.*

One may wonder about the cost of tuple creation, update and deletion since they may generate insertion and deletion of values in domains. While these actions are more complex than their FS counterpart, their implementation remains more efficient in the smart card context, simply because the amount of data to be written is much smaller. To amortize the slight overhead of domain storage, we only store by domain all large attributes (*i.e.,* greater than a pointer size) containing duplicates. Obviously, attributes with no duplicates (*e.g.,* keys) need not be stored by domain but with FS. Variable-size attributes – generally larger than a pointer – can also be advantageously stored in domains even if they do not contain duplicates. The benefit is not storage savings but memory management simplicity (all tuples of all relations become fixed-size) and log compactness.

We now address index compactness along with data compactness. Unlike disk-based DBMS that favor indices which preserve access locality, smart cards should make intensive use of secondary (*i.e.,* pointer-based) indices. The point here is to make these indices

[29] Compression techniques can be advantageously used in conjunction with DS to increase compactness [Gra93]. Note that processing abilities on the compacted data must be envisioned to reach high performance.

efficient and as compact as possible, integrating them in the storage model instead of additional structures. Let us first consider select indices. A select index is typically made of two parts: a collection of values and a collection of pointers linking each value to all tuples sharing it. Assuming the indexed attribute varies on a domain, the index's collection of values can be saved since it exactly corresponds to the domain extension. The extra cost incurred by the index is then reduced to the pointers linking index values to tuples. Let us go one step further and get these pointers almost for free. The idea is to store these *value-to-tuple* pointers in place of the *tuple-to-value* pointers within the tuples (*i.e.,* pointers stored in the tuples to reference their attribute values in the domains). This yields to an index structure which makes a ring from the domain values to the tuples, as shown Figure 37(a) The ring index can also be used to access the domain values from the tuples and thus serve as data storage model. Thus we call *Ring Storage (RS)* the storage of a domain-based attribute indexed by a ring. The index storage cost is reduced to its lowest bound, that is, one pointer per domain value, whatever be the cardinality of the indexed relation. This important storage saving is obtained at the price of extra work for projecting a tuple to the corresponding attribute since retrieving the value of a ring stored attribute means traversing in average half of the ring (*i.e.,* up to reach the domain value).




*(a) Ring Index on a Regular Attribute.* *(b) Ring Index on a Foreign-key Attribute.*

**Figure 37** *Ring Storage.*

Join indices [Val87] can be treated in a similar way. A join predicate of the form (*R.a*=*S.b*) assumes that *R.a* and *S.b* vary on the same domain. Storing both *R.a* and *S.b* by means of rings leads to define a join index. In this way, each domain value is linked by two separate rings to all tuples from *R* and *S* sharing the same join attribute value. As most joins are performed on key attributes, *R.a* being a primary key and *S.b* being the foreign key referencing *R.a*, key attributes are stored with FS in our model. Nevertheless, since *R.a* is the primary key of *R*, its extension forms precisely a domain, even if not stored outside of *R*. Since attribute *S.b* takes it value in *R.a*'s domain, it references *R.a* values by means of pointers. Thus, the domain-based storage model naturally implements for free a *unidirectional join index* from *S.b* to *R.a* (see Figure 36(b)). If traversals from *R.a* to *S.b* need be optimized too, a *bi-directional join index* is required. This can be simply achieved by defining a ring index on *S.b*. Figure 37(b) shows the resulting situation where each *R*

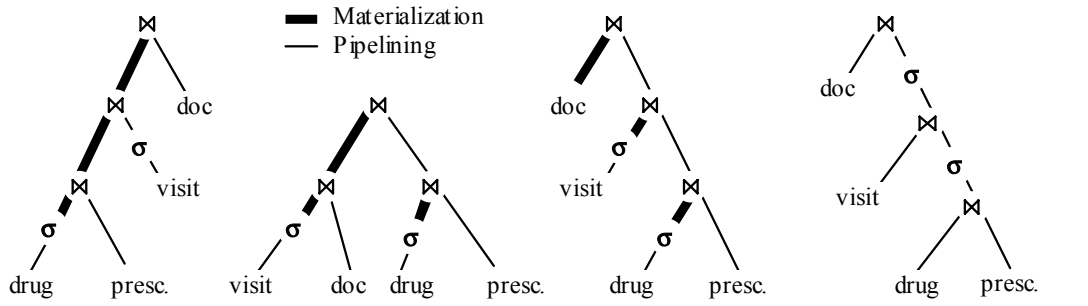
tuple is linked by a ring to all *S* tuples matching with it and vice-versa. The cost of a bi-directional join index is restricted to a single pointer per *R* tuple, whatever be the cardinality of *S*. This index structure can even be enhanced by combining RS and DS. This leads to a bi-directional join index providing direct (*i.e.,* one single pointer) traversals from *S.b* to *R.a*. This enhanced storage model, called *Ring Inverse Storage (RIS)*, requires one additional pointer per *S* and *R* tuples. Its storage cost is much more expensive, and thus must give a high performance benefit to be adopted. Note that this situation resembles the well-known Codasyl model.

Our storage model combines FS, DS, and RS. A further storage cost study can be found in [BPV+01]. The case of RIS is considered in the performance analysis.

## 4.2 Query Processing

Traditional query processing strives to exploit large main memory for storing temporary data structures (*e.g.,* hash tables) and intermediate results. When main memory is not large enough to hold some data, state-of-the-art algorithms (*e.g.,* hybrid hash join [ScD89]) resort to materialization on disk to avoid memory overflow. These algorithms cannot be used for a PicoDBMS. First, the write rule proscribes writes in stable memory for temporary materialization. Second, dedicating a specific RAM area does not help since we cannot estimate a-priori its size: choosing it small increases the risk of memory overflow thereby leading to writes in stable memory, and choosing it large reduces the stable memory area already limited in a smart card (RAM rule), with no guarantee that query execution will not produce memory overflow [BKV98]. Third, State-of-the-art algorithms are quite sophisticated, which precludes their implementation in a PicoDBMS whose code must be simple, compact and secure (compactness and security rules). To solve this problem, we propose query processing techniques that do not use any working RAM area nor incur any writes in stable memory. In the following, we describe these techniques for simple and complex queries, including aggregation and remove duplicates.

To begin with, we consider here the execution of *SPJ* (*Select/Project/Join*) queries. Query processing is classically done in two steps. The query optimizer first generates an "optimal" *query execution plan* (*QEP*). The QEP is then executed by the query engine which implements an *execution model* and uses a library of relational operators [Gra93]. The optimizer can consider different shapes of QEP: *left-deep*, *right-deep* or *bushy trees*. In a left-deep tree, operators are executed sequentially and each intermediate result is materialized. On the contrary, right-deep trees execute operators in a pipeline fashion, thus avoiding intermediate result materialization. However, they require materializing in memory all left relations. Bushy trees offer opportunities to deal with the size of intermediate results and memory consumption [SYT93].




***Left deep tree*** ***Bushy tree*** ***Right deep tree*** ***Extreme right deep tree***

**Sample healthcare schema:**

*Doctor (*DocId, name, specialty, …*)*
*Visit (*VisId, DocId, date, diagnostic, …*)*
*Prescription (*VisId, DrugId, qty, …*)*
*Drug (*DrugId, name, type, …*)*

**Query Q1:**

```
SELECT D.name
FROM Doctor D, Visit V, Prescription P, Drug D
WHERE D.DocId = V. DocId AND V.VisId = P.VisId
    AND P.DrugId = D.DrugId AND D.type = 'ANTIBITIC'
    AND V.date > 01-jan-2000;
```

**Figure 38** *Several Execution Trees for Query Q1:'Who prescribed antibiotics in 1999 ?'.*

In a PicoDBMS, the query optimizer should not consider any of these execution trees as they incur materialization. The solution is to only use pipelining with *extreme right-deep trees* where all the operators (including select) are pipelined. As left operands are always base relations, they are already materialized in stable memory, thus allowing to execute a plan with no RAM consumption. Pipeline execution can be easily achieved using the well known *Iterator Model* [Gra93]. In this model, each operator is an *iterator* that supports three procedure calls: *open* to prepare an operator for producing an item, *next* to produce an item, and *close* to perform final clean-up. A *QEP* is activated starting at the root of the operator tree and progressing towards the leaves. The dataflow in the model is demand-driven: a child operator passes a tuple to its parent node in response to a *next* call from the parent.

The Select operator tests each incoming tuple against the selection's predicates. The values of the incoming tuples needed to evaluate the predicate are directly read (FS) in the tuple and/or reached by dereferencing a pointer (DS/RIS) and/or by following a pointer's ring. While indexed (RS storage of the attribute participating in the selection), the selection's predicate (or part of, whether multi-attribute) can be evaluated on the distinct values, and the matching tuples are directly retrieved following the pointers' rings. However, extreme right-deep trees allow to speed-up a single select on the first base relation (*e.g.*, *Drug.type* in our example), but using a ring index on the other selected attributes (*e.g.*, *Visit.date*) may slow down execution as the ring need be traversed to retrieve their value.

Project simply builds a result tuple by copying the value (FS) and/or dereferencing the cursors (DS/RIS) and/or following the pointer's ring to reach the value (RS) present in the input tuple. The position of the project operator in the query tree is non-traditional : they are pushed up to the tree since no materialization occurs. Note that the final project incurs an additional cost in case of ring attributes.

Join implements a Cartesian product between its left and right inputs. Indeed, since no other join technique can be applied without ad-hoc structures (*e.g.*, hash tables) and/or working area (*e.g.*, sorting), Joining relations is done by a nested-loop algorithm. Each incoming right tuple induce a complete iteration on the left input to retrieve the matching tuples. In case of indices (DS/RS/RIS), the cost of joins depends on the way indices are traversed. Consider the indexed join between *Doctor* (*n* tuples) and *Visit* (*m* tuples) on their key attribute. Assuming a unidirectional index, the join's cost is proportional to *n*m* starting with *Doctor* (*i.e.*, right input is *Doctor*) and to *m* starting with *Visit* (*i.e.*, right input is *Visit*). Assuming now a bi-directional index, the join's cost becomes proportional to $n+m$ starting with *Doctor* and to $m^2/2n$ starting with *Visit* (retrieving the doctor associated to each visit incurs traversing half of a ring in average). In the latter case, a naïve nested loop join can be more efficient if the ring cardinality is greater than the target relation cardinality (*i.e.*, when $m>n^2$). In that case, the database designer must clearly choose a unidirectional index between the two relations.

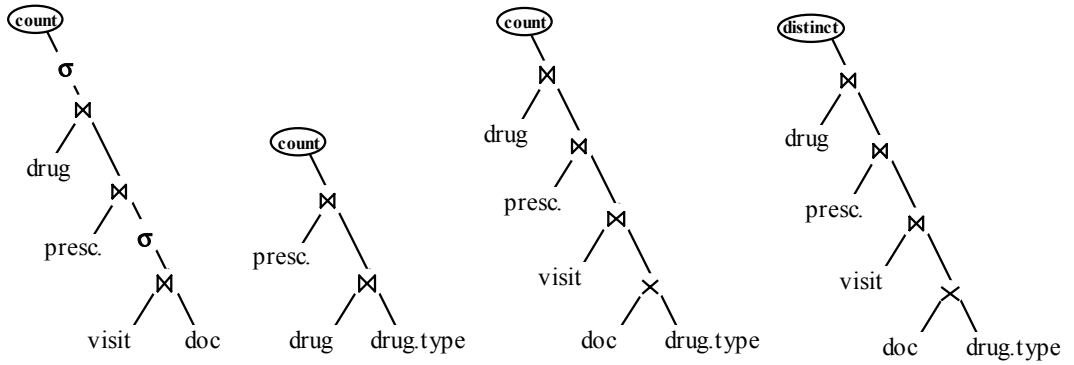
We now consider the execution of aggregate and duplicate removal operators (sort operations is not described here since it can be performed on the terminal). At a first look, pipeline execution is not compatible with these operators which are classically performed on materialized intermediate results. Such materialization cannot occur either in the smart card due to the RAM rule or in the terminal due to the security rule. Note that sorting can be done in the terminal since the output order of the result tuples is not significant, *i.e.,* depends on the DBMS algorithms. We propose a solution to the above problem by exploiting two properties: (i) aggregate and duplicate removal can be done in pipeline if the incoming tuples are yet grouped by distinct values and (ii) pipeline operators are order-preserving since they consume (and produce) tuples in the arrival order. Thus, enforcing an adequate consumption order at the leaf of the execution tree allows pipelined aggregation and duplicate removal. For instance, the extreme pipeline tree of Figure 38 delivers the tuples naturally grouped by *Drug.id*, thus allowing group queries on that attribute.

Let us consider now query Q2 of Figure 39. As pictured, executing Q2 in pipeline requires rearranging the execution tree so that the relation *Doctor* is explored first. Since the relation *Doctor* contains distinct doctors, the tuples arriving to the *count* operator are naturally grouped by doctors.

The case of Q3 is harder. As the data must be grouped by *type of drugs* rather than by *Drug.id*, an additional join is required between the relation *Drug* and the domain *Drug.type*. Domain values being unique, this join produces the tuples in the adequate order. If the domain *Drug.type* does not exist, an operator must be introduced to sort the relation *Drug* in pipeline. This can be done by performing *n* passes on *Drug* where *n* is the number of distinct values of *Drug.type*.

The case of Q4 is even trickier. The result must be grouped on two attributes (*Doctor.id* and *Drug.type*), introducing the need to start the tree with both relations! The solution is to insert a Cartesian product operator at the leaf of the tree in order to produce tuples ordered

by *Doctor.id* and *Drug.type*. In this particular case, the query response time should be approximately *n* times greater than the same query without the 'group by' clause, where *n* is the number of distinct *types of drugs*.




**Q2:** *Number of antibiotics prescribed per doctor in 1999*

**Q3:** *Number of prescription per type of drug*

**Q4:** *Number of prescriptions per doctor and type of drug*

**Q5:** *Distinct doctor and type of drug*

**Figure 39** *Four 'Complex' Query Execution Plans.*

Q5 retrieves the distinct couples of *doctor* and *type of prescribed drugs*. This query can be made similar to Q4 by expressing the distinct clause as an aggregate without function (*i.e.,* the query "*select distinct* $a_1$*,..,* $a_n$ *from ...*" is equivalent to "*select* $a_1$*, .. ,* $a_n$ *from ... group by* $a_1$*,..,* $a_n$"). The unique difference is that the computation for a given group (*i.e., distinct result tuple*) can stop as soon as one tuple has been produced.

Regarding query optimization, extreme right-deep trees context makes it rather straightforward. No sophisticated optimizer is thus required. However, simple heuristics can give attractive enhancements. For example, the selections have to be executed as soon as possible, and the access ordering can favor the join's indices traversal in the most favorable sense (only one possibility in case of unidirectional index, *i.e.*, DS, and from the referred key relation to the referent foreign key one in case of bi-directional index, *i.e.*, RS/RIS).

## 4.3 Transaction Processing

Like any data server, a PicoDBMS must enforce the well-known transactional ACID properties [BHG87] to guarantee the consistency of the local data as well as be able to participate in distributed transactions. While being out of the scope of this study, we give a minimal description of the transactional mechanisms presented in [BPV+01] to take their cost into account in the performance measurements (next section). Thus, this section deals with integrity control, concurrency and logging costs.

Regarding integrity control, traditional techniques are used and do not deserve any particular comments. Only uniqueness and referential constraints checking are considered in the performance evaluation. Indeed, they have to be taken into account since they are

inherent to relational data structures. Other integrity constraints are not taken into account since they are application dependent.

Regarding concurrency control, PicoDBMS has been developed on mono-threaded platforms[30], for a single user at a time usage. In this context, concurrency is obviously not a concern. One can imagine concurrency management in multi-threaded chips, but this perspective is clearly out of the scope of this study.

The logging process deserves a more complete explanation. Logging is necessary to ensure local atomicity, global atomicity (in a distributed setting) and durability. However, as explained in [BPV+01], durability cannot be enforced locally by the PicoDBMS because the smart card is more likely to be stolen, lost or destroyed than any traditional computer. Indeed, mobility and smallness play against durability. Consequently, durability must be enforced through the network. As stated in [AGP02], the cost of durability as well as global atomicity can entirely be transferred to the network thanks to an ad-hoc protocol named *Unilateral Commit for Mobile* (UCM). Remains the cost induced by local atomicity, described in the following.

The local atomicity process relies on two basic ways to perform updates in a DBMS. With the first technique, called *shadow update*, the updates are performed on shadow objects that are atomically integrated in the database at commit time. This model has been shown convenient for smart cards equipped with a small Flash memory [Lec97]. However, it is poorly adapted to pointer-based storage models like RS since the object location changes at every update. In addition, the cost incurred by shadowing grows with the memory size. Indeed, either the granularity of the shadow objects increases or the paths to be duplicated in the catalog become longer. In both cases, the writing cost – which is the dominant factor – increases. The second technique ensuring the local atomicity realizes *update in-place* [BHG87] in the database, and saves required data (*Write-Ahead Logging: WAL*) to undo the effects in case the transaction is aborted. Unfortunately, the relative cost of WAL is much higher in a PicoDBMS than in a traditional disk-based DBMS which uses buffering to minimize I/Os. In a smart card, the log must be written for each update since each update becomes immediately persistent. This roughly doubles the cost of writing. Despite its drawbacks, *update in-place* is better suited than *shadow update* for a PicoDBMS because it accommodates pointer-based storage models and its cost is insensitive to the rapid growth of stable memory capacity. Traditional WAL logs the values of all modified data. RS allows a finer granularity by logging pointers in place of values. The smallest the log records, the cheapest the WAL. The log record must contain the tuple address and the old attribute values, that is a pointer for each RS or DS stored attributes (pointing at the ring or domain value) and a regular value for FS stored attributes. Note that the ring comes for free in term of logging since we store a pointer to the value shared among the ringed tuples, providing the ability to chain the tuple in the corresponding ring at recovery time. In case of a tuple

[30] Multi-threaded platforms are not available yet.

insertion or deletion, assuming each tuple header contains a status bit (*i.e.,* dead or alive), only the tuple address has to be logged in order to recover its state. This induces an overhead to update the rings traversing the tuple. Note that insertion and deletion of domain values (domain values are never modified) should be logged as any other updates. This overhead can be avoided by implementing a deferred garbage collector that destroys all domain values no longer referenced by any tuple.

To conclude, the update in-place model along with pointer-based logging and deferred garbage-collector reduces logging cost to its lowest bound, that is, a tuple address for inserted and deleted tuples, and the values of updated attributes (*i.e.*, a pointer for DS, RS and RIS stored attributes).

# 5 Performance Evaluation and Lessons Learned

This section evaluates the performance of the PicoDBMS, considering our proposed models for storage and indexation, and raises the main learned lessons of this work. We first determine the performance metric of the evaluations. Then, we present the footprint of the embedded DBMS code. Besides, we evaluate the compactness benefits of each storage model, and we conduct an in-depth query performance evaluation combining measurements and simulations. Finally, we assess the performance and applicability of PicoDBMS wrt two typical smart card database applications.

## 5.1 Objective and Performance Metrics

The metrics and the method selected to conduct this system evaluation are induced by the following characteristics of the considered working environment:

(C1) The stable storage capacity is reduced: from tens to hundreds of kilobytes.

(C2) The target applications have an append-intensive update profile: historical information is frequently added to the system, while updates and deletes are less frequent.

(C3) The access rights are rather sophisticated: the authorized views may combine projections, selections, joins, and aggregates in their definition (see Section 2, Table 9).

(C4) The chip (smart card) is viewed as a smart storage medium: the users (human or software) connect to the device to access (according to its access rights) the embedded data, analogously to any other storage device (magnetic disc, USB key, etc.).

The first characteristic (C1) highlights the importance of evaluating the DBMS storage potential, meaning its ability to compress the on-board data. The question is whether the DBMS code footprint has to be taken into account in this analysis, and how. First, in a real

setting, the DBMS code should be stored in the smart card's ROM, and so do not directly compete with on-board data. To this respect, only the total code footprint need be evaluated to establish whether it fits into the available ROM. However, smart card manufacturers try to calibrate each hardware resource to its minimum to save money on mass markets (from millions to billions of units). Thus, we will give in the two following sections the size of different PicoDBMS implementations, from a simple *flat-based* version to a *full-fledged* one, as well as the amount of storable tuples for these respective implementations. In that way, both evaluations in terms of code and data footprint are provided.

Characteristic (C2) highlights the importance of the insert operations. The update and delete costs must be acceptable, but are less significant for the target applications.

Characteristic (C3) imposes to measure the performance of a broad range of queries (*i.e.*, views), from the simplest to the most complex one.

Characteristic (C4) suggests to raise measures comparable to those evaluating traditional storage medium, namely latency and transmission rate. The latency corresponds to the time spent to produce the first result tuple. The transmission rate can be expressed in terms of result tuples produced per second. This metrics is relevant in case of an application bothering about the data transmission rate (*e.g.*, to fill in the window of a cellular phone in a given time), and also about total query execution time (*e.g.*, to sort the complete result before display). Note that the transmission rate is obviously bounded by those of the chip and the smart card reader, like the bandwidth of the data bus limits the real throughput of magnetic disks. However, many smart card readers are now connected to the USB port, which predicts an important connectivity gain in the short term.

## 5.2 PicoDBMS Kernel Footprint

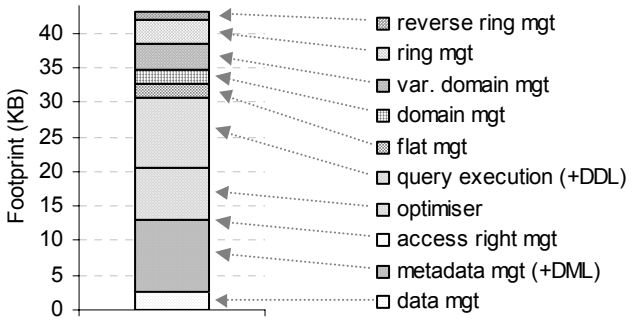
The diagram shown in Figure 40(a) presents the respective size of the PicoDBMS modules while Figure 40(b) gives the total footprint of the main implementation's alternatives. The figures are self-explanatory. However we can bring out two interesting conclusions, the first one on the embedded code footprint, and the second one on the *metadata management* module.

The code footprint is a challenge for many lightweight and embedded DBMS developers. The lightweight DBMS editors strongly exploit this argument for commercial purpose. As well, academic works try to gain on the DBMS kernel footprint. Our opinion on this point is however that code footprint minimization is far less significant than data compactness. The reason for this is manifold. First, the gap between the full-fledged and the Flat-based versions of PicoDBMS is much less important than expected. Second, ROM cells are 4 times smaller than EEPROM cells and so are the respective benefits of decreasing the code and the data footprint. Third, to reach a lower bound in terms of code footprint leads to select the flat storage model, which strongly increases the database footprint (see next

section)[31].

| *PicoDBMS version* | *Storage model* | *Indexation* | *Footprint (KB)* |
|---|---|---|---|
| Flat-Based | FS | No | 26 |
| Domain-Based | FS/DS | Join | 30 |
| Index-Based | FS/DS/RS | Join & select | 39 |
| Full-Fledged | FS/DS/RS/RIS | Join & select | 42 |

*(b) Different PicoDBMS Variations.*




*(a) Full-Fledged Version of PicoDBMS.*

**Figure 40** *PicoDBMS Code Footprint.*

One can notice that the *metadata management* module (see Figure 40(a)) is rather heavy compared to the others. Indeed, several query execution plans cannot fit in RAM simultaneously. This precludes to store metadata inside traditional database relations, and to access it by means of queries. Metadata are therefore stored in ad-hoc structures, and are accessed by calling ad-hoc procedures, explaining the heavy footprint of this component.

## 5.3 Database Footprint

This section evaluates the database compactness obtained by the four candidate storage and indexation schemes (FS, DS, RS, and RIS). Such an evaluation cannot be conducted accurately using a synthetic dataset, unable to translate a real-life distribution of duplicate values. Thanks to our contact with medical partners, we used for this experience a genuine medical dataset, stored into the simplified medical database schema presented in Figure 41.

[31] Nevertheless, we designed a very light version of PicoDBMS to maintain a new generation address book in a SIM card with our industrial partner Schlumberger. The amount of data was not huge, so that the flat storage gave acceptable results (4 kilobytes savings wrt DS plus 4 kilobytes savings wrt the metadata management). Rings were added to speed-up joins but aggregate operators were useless (3 kilobytes savings). The query optimizer was also removed, the joins paths being known in advance (3 kilobytes saving). Finally, the PicoDBMS code footprint has been reduced to 28 kilobytes, showing the benefit of code componentization.

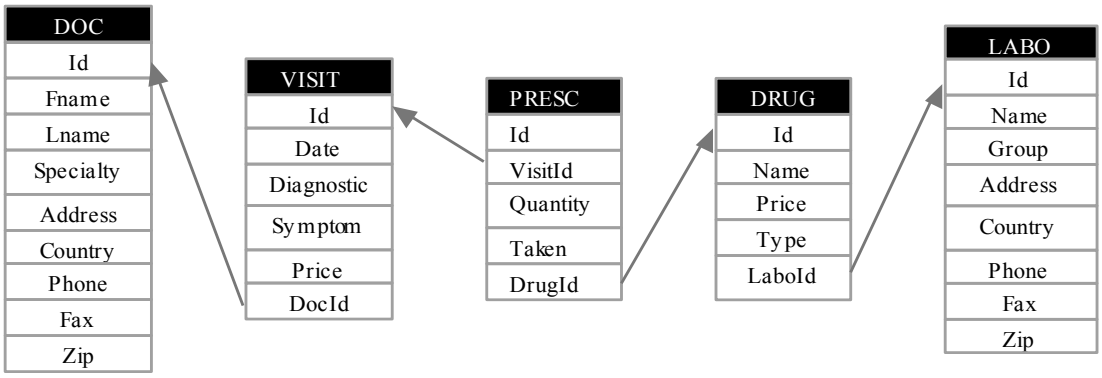
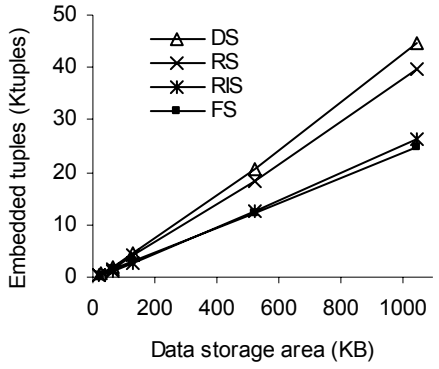
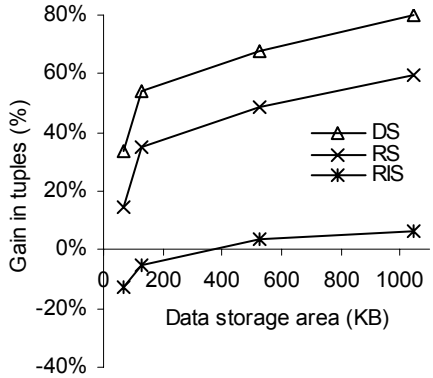



**Figure 41** *Medical Database Schema.*

Figure 42(a) pictures the maximum number of tuples that can be stored by each storage and indexation models as a function of the row data storage capacity. Figure 42(b) and Figure 42(c) plot ratios between the different models. The curves are self-explanatory. Note that RIS performs really badly (RIS is very close to FS) compared to DS and RS. Thus, RIS should be adopted only if it provides a significant performance gain at query execution time. Note also that DS and RS provide relatively close results, highlighting the high compactness of ring indices.




*(a) Embedded Tuples Amount.*




*(b) Gain Compared to FS.*

*(c) Lost Compared to DS.*

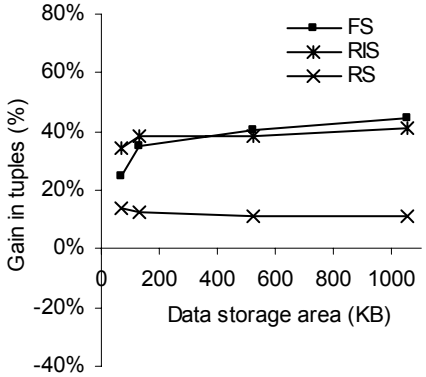



**Figure 42** *PicoDBMS Storable Tuples.*

We can also notice that the more the embedded tuples (see Figure 42(a)), the less the indexation cost. Indeed, the slope of the curve, plotting the number of embedded tuples in function of the storage capacity, is much higher for indexed models (RIS, RS) than for FS. Also, the gain compared to FS (see Figure 42(b)) increases with the database size. This remark induces that for bigger smart cards, indices comes at a lower cost. The same remark applies to DS which acts as a dictionary compression scheme. Thereby, the compression efficiency increases with the database cardinality.

Note also that the total number of embedded tuples does not exceed 50.000 for a smart card providing 1 megabytes of stable storage. Thus, we do not examine in the sequel queries involving more than this upper bound.

## 5.4 PicoDBMS Performance

This section measures the PicoDBMS performance for update (tuple insertion) and queries (view evaluation). Two different platforms have been used to conduct these experiments: a real smart card platform prototype and a hardware simulator of this smart card (both provided by Axalto), allowing to deduce the performance of PicoDBMS on a dataset that strongly exceeds the storage capacity of the real smart card. We present below these platforms, the used datasets and queries, and then the results.

While the analogy between the smart card and a magnetic disk (see Section 5.1) in terms of performance metrics is still valid, the latency parameter turns to be negligible in all situations. Indeed, the latency varies from 1 to 4 milliseconds depending on the query plan complexity. Actually, the latency strictly corresponds to the time required to build the query plan, the tuples being computed in a pure pipeline fashion. Thus, this section will concentrate on the transmission rate, a more significant metrics in our setting.

### *5.4.1 Platform and Dataset*

The smart card prototype and the hardware simulator used in the performance evaluation are 32-bit experimental platforms running at 50 MHz, with the operating system Zeplatform[32]. The smart card is equipped with 64 kilobytes of EEPROM stable memory available for the user, 96 kilobytes of ROM holding the operating system, and 4 kilobytes of RAM consumed by the operating system and the execution heap, with a hundred of bytes remaining at user disposal. An internal timer is also embedded in the smart card prototype, giving the ability to measure the response time for each incoming APDU. This platform allows a performance evaluation on tiny databases (tens of kilobytes), the PicoDBMS code being located in EEPROM in this prototype. To assess the performance on a higher amount of data, we used the simulation platform pictured in Figure 43. This platform is made of the smart card hardware simulator connected to a control PC, and linked to a client PC connected to a

[32] Announced by Schlumberger/CP8 at the cartes'2000 forum.

standard smart card reader (ISO 7816 standard). The hardware simulator can be parameterized by the control PC to emulate a smart card equipped with 1 megabytes of stable storage. This hardware simulator is cycle accurate, meaning that it monitors the exact number of CPU cycles between two breakpoints set in the embedded code. We checked the accuracy of the simulation platform comparing the simulator's measures on small databases to the one obtained using the real smart card prototype.

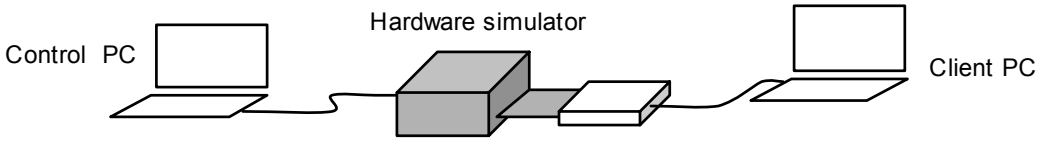



**Figure 43** *Smart Card Simulation Platform.*

Our first experiments highlighted important bottlenecks in ZePlatform when dealing with data intensive operations, a rather unusual case in traditional smart card applications. Thus, Axalto modified internals of Zeplatform, providing a much better support for PicoDBMS. Indeed, stable memory accesses usually induce costly security checks to guarantee that the embedded code remains confined in an authorized memory segment[33]. This contradicts the global design of PicoDBMS, making an intensive use of pointers and then generating a huge amount of direct accesses to the EEPROM. The PicoDBMS kernel being a priori as secure as the operating system (assuming it is developed, certified and installed in the card under the same conditions as the OS itself), the security checks have been strongly simplified, both on the smart card prototype and on the hardware simulator. The performance gain obtained thanks to this modification of ZePlatform and to the rewriting of PicoDBMS from JavaCard to C is roughly two orders of magnitude (the element of comparison being the JavaCard prototype demonstrated at VLDB'01).

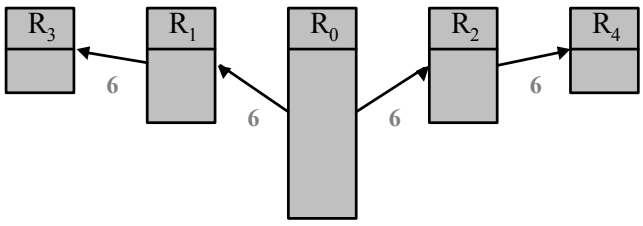



**Figure 44** *Synthetic Database Schema.*

The communication cost between the smart card and the client terminal is not taken into account in the measurement. The reason for this is twofold. First, the PicoDBMS transmission rate is the main relevant factor and must be assessed whatever be the communication setting between PicoDBMS and the application (*e.g.*, a cellular phone application communicating internally with a PicoDBMS-enabled SIM card). Second, the communication rate is not considered as a long term bottleneck: smart cards are today

[33] More than 2000 CPU cycles are required by the security checks before authorizing each access to

connected on USB port, and some prototypes delivering 1 megabytes per second are already developed by smart card manufacturers.

For the query performance evaluation, we use a synthetic dataset, thus different from the one used to observe data compactness, in order to control and vary the selectivity of each attribute. The database schema of this synthetic dataset is composed of five relations (see Figure 44). The number of involved relations is representative of a rather complex embedded database, sharing strong similarities with the one used in traditional benchmarks (*e.g.*, TCP benchmarks). Moreover, we choose relations' cardinality ratios similar to the TPC-R/H/W benchmarks, that is to say a large relation referencing two relations 6 times smaller, each one referencing in turn another relation again 6 times smaller in term of records number. The primary and foreign keys fields are numeric and each relation owns ten attributes, five of them holding numerical values, and the others holding characters strings with an average size of 20 bytes. The number of distinct values in each string attribute is fixed to 10% of the relation's cardinality, except for attributes participating in a group by clause for which this ratio is fixed to 1%. The detailed description of relation $R_0$ is given in Table 10. The other relations are built on the same schema, except that they own a single foreign key attribute $A_1$, while $A_2$ has the same properties as $A_3$ and $A_4$. Note that attribute $A_0$ is stored in FS since it is unique and cannot benefit from DS, RS, or RIS models. $A_3$ and $A_4$ are never stored in DS since their occurrences are smaller than the size of a pointer (4 bytes).

| *Attribute' name* | *Integrity constraint* | *Value type* | *storage model alternatives* | *Distinct values* | *Size (Bs)* |
|---|---|---|---|---|---|
| $A_0$ | Primary key | Numeric | FS | $\|R_0\|$ | 4 |
| $A_1$, $A_2$ | Foreign key | Numeric | FS/DS/RS/RIS | $\|R_0\|/6$ | 4 |
| $A_3$, $A_4$ | | Numeric | FS/RS/RIS | $\|R_0\|/10$ | 4 |
| $A_5$, $A_6$, $A_7$, $A_8$, $A_9$ | | Character string | FS/DS/RS/RIS | $\|R_0\|/10$ | 20 (average) |

**Table 10** *Schema of Relation $R_0$.*

Different queries are evaluated (see Table 11), each one representative of one of the access right type presented in Section 2.

In the following, each curve is plotted as a function of a given parameter. Other parameters are set to the default values given below:

- *Project list cardinality*: the number of attributes participating in a projection is set to 2 (the performance measurements will show that project is not a bottleneck).
- *Selection selectivity*: the result cardinality for a query involving selection is set to 1% of the result cardinality of the same query without selection (*e.g.*, query *SP* of Table 11 delivers 1% of $R_0$ tuples).

the EEPROM.

- *Join selectivity*: joins are equi-joins on key attributes. Thus, the number of result tuples is equal to the referencing relation's cardinality (*e.g.*, joining $R_0$ and $R_1$ gives $R_0$ tuples).

- *Grouping factor*: the result cardinality for a query involving a group by clause is set to 1% of the same query without group by. In other words, the cardinality of each group produced is 100.

| *Acronym* | *Access right description* | *Corresponding representative query* |
|---|---|---|
| *P* | Subset of attributes | Projection on $R_0$ |
| *SP* | Predicates on a single relation | Selection on $R_0$ |
| *SPJ* | Predicates on two relations (direct relationship) | Selection on $R_0$, join between $R_0$ and $R_1$ |
| $SPJ^n$ | Predicates on several relations (transitive relationship) | Multi-selection, multi-join of relations $R_0$-$R_1$-$R_2$-$R_3$-$R_4$ |
| *SPG* | Single relation, mono-attribute aggregation | Mono-attribute group by on $R_0$, one aggregation |
| *SPJG* | Several relations, mono-attribute aggregation | Join between $R_0$, $R_1$, $R_3$, Mono-attribute group by, one aggregation |
| $SPJG^n$ | Multi-attribute aggregation | Join between $R_0$, $R_1$, $R_3$, group by on two attributes of two distinct relations, one aggregation |

**Table 11** *Query Set for the Performance Evaluation.*

The attributes on which projections, selections and group by apply are not specified in Table 11. The reason for this is that the same query is measured on different attributes, allowing us to present both *best case* and *worst case* for each query (a single curve is plotted when the difference between both cases is insignificant).

### *5.4.2 Insertion*

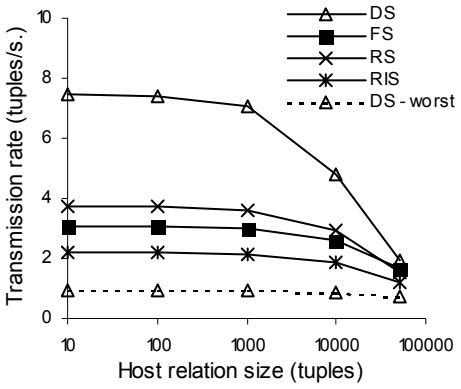
In an append-intensive database context, the performance of tuple insertion turns to be crucial. We measure the insertion rate (number of inserted tuples per second) on the synthetic database. Since insertion is a unary operation, the latency is included in the presented performance result.

In the experiment, the tuples are inserted into $R_0$. Indeed, most insertions occur in $R_0$ since this relation is the biggest in term of cardinality. Moreover, insertions in this relation represent a worst case in terms of performance since they involve the highest integrity control cost: $R_0$ owns one more foreign key than the other relations (see Table 10), the domains of its attributes (for those stored in DS) have the largest cardinality and its number of primary key occurrences is also the largest one.

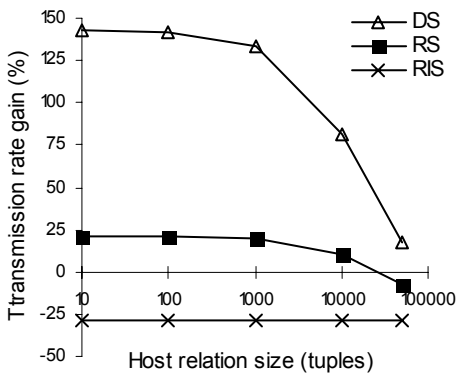
More precisely, an insert operation involves the four following main processing steps:

(Op1) *Primary and foreign key integrity constraint checking*: the insertion of a tuple requires scanning entirely the host relation (uniqueness constraint checking) as well as 1/6 (according to the relation's ratio) of each referenced relation (referential constraint checking).

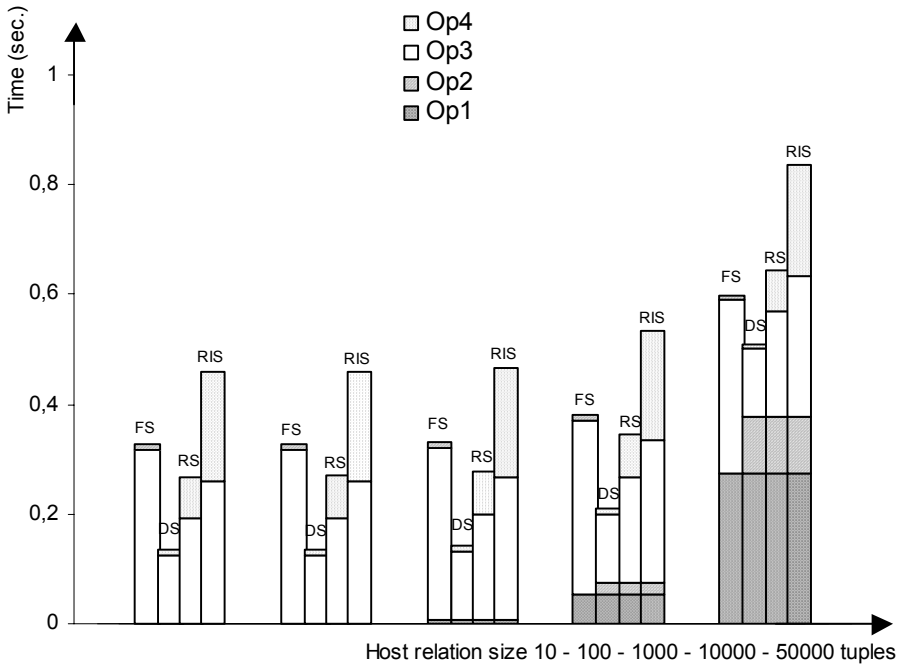
(Op2) *Update of the storage structures*: for RS, DS and RIS, the value of each newly inserted attribute has to be searched in the corresponding domain and the domain is updated if this value cannot be found. The figures below show the most frequent cost, considering that domains are either static or rather stable after a first bunch of insertions.




*(a) Transmission Rate.*




*(b) Relative Performance Gain wrt FS.*




*(c) Decomposed Costs.*

**Figure 45** *Performance of Insertions.*

(Op3) *Allocation of the inserted tuple*: this operation includes values and pointers writes inside the allocated tuple's area as well as the updates required to maintain the rings traversing the tuple.

(Op4) *Log maintenance*: this operation is related to the *Pointer-based logging* mechanism sketched in Section 4.3.

Figure 45(a) shows that the data transmission rate is acceptable (more than one tuple per second[34]) whatever be the considered storage model. The relative gains of each storage model compared to FS are presented in Figure 45(b). This figure shows that the compactness of a storage model is beneficial wrt the insertion performance mainly for a relatively small amount of data (up to a thousand of tuples in $R_0$). Indeed, and as Figure 45(c) makes clear, the high writing cost of Op3 and Op4 in FS is dominant in this case. After this threshold, the costs of Op1 and Op2, which are directly related to the relation's cardinality, flatten the gap between the different storage models. Let us note that operation Op1 could be accelerated using a B-Tree on the primary keys, but this gain would have to be balanced against the index update cost (writes in stable memory) and the storage overhead.

### *5.4.3 Projection*

Figure 46 gives the transmission rate for project queries, representative of the *P* access right type. Curves are plotted for each storage model in function of the number of attributes participating in the projection.

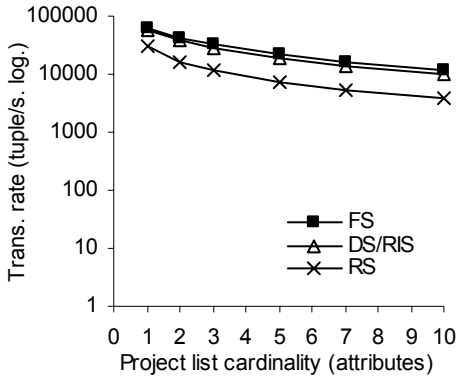



**Figure 46** *Projection Transmission Rate.*

The following conclusions can be raised:

- *The project transmission rate is independent of the relation cardinality*: obviously, the project operator requires the same time to produce each tuple.
- *RS slows down the projection*: RS induces in average the traversal of half of a ring to

[34] The limit of one second is often considered for applications involving a human interaction.

reach the value of a given attribute. In our dataset, the average length of rings is 10 pointers, thereby leading to a performance degradation by a factor of 5 (note the logarithmic scale of Figure 46).

- *The projection is not a bottleneck*: Even attributes stored in RS do not raise any problem, the throughput being over 5000 results per second whatever be the considered storage model and the number of projected attributes.

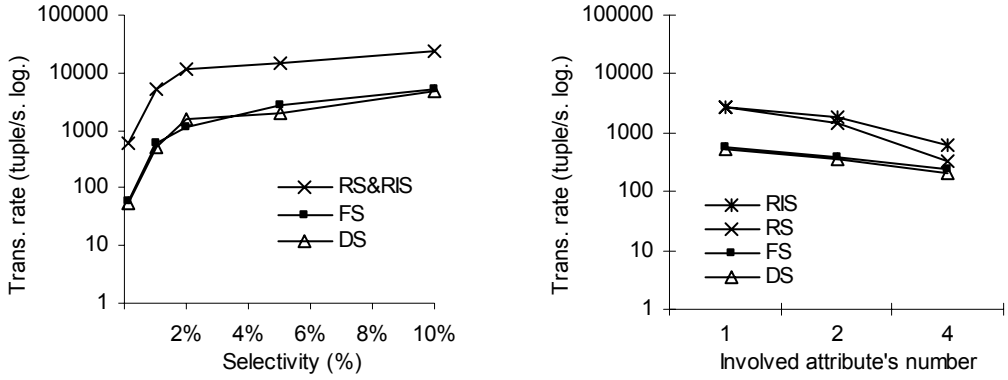
### *5.4.4 Selection*

Figure 47 presents the transmission rate for mono-relation selection queries, representative of the *SP* access right type. The transmission rate is presented as a function of the query selectivity in Figure 47(a) and as a function of the selection complexity (number of attributes involved in the qualification) in Figure 47(b). These curves lead to the following remarks.

- *The selection transmission rate is independent of the relation cardinality*: the select operator, similarly to the project operator, requires a constant time to produce each tuple.

- *The selection transmission rate increases while the query selectivity* decreases: indeed, the worse the query selectivity, the less irrelevant tuples scanned and checked before producing a matching tuple.

- *RIS and RS models outperform DS and FS:* these storage models offer natural selection indices. The performance gain is directly determined by the ratio between domains and relations cardinalities (set to 10 in our dataset). However, when the selection qualification involves several attributes, the performance gap between indexed and non-indexed storage models decreases, highlighting the fact that our execution model cannot exploit several indices for a same query.




*(a) Mono-attribute Selection.* *(b) Multi-attribute Selection.*

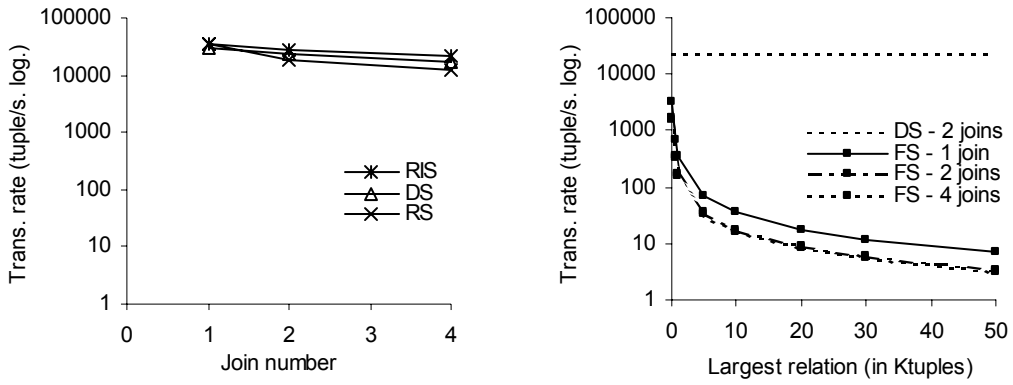
**Figure 47** *Selection Queries Transmission Rate (SP).*

### *5.4.5 Joins*

Figure 49 presents the performance of queries representative of *SPJ* and $SPJ^n$ access right types, involving both selections and joins, with different query selectivity's. Figure 48 measures the performance of pure join queries varying the join number. These two figures are necessary to capture the incidence of selection on join queries. Let us first consider the conclusions that can be drawn from Figure 48:

- *High join transmission rate for RIS, RS and DS:* models RIS, RS and DS exhibit a high throughput for pure join queries and this throughput is independent of the database size (see Figure 48(a)). Moreover, the performance is rather stable while facing an increasing number of joins. Roughly speaking, the execution of a query plan involving N joins and projects is slow down by a factor 1/(N+2) for each additional join. For example, adding a second (resp. third) join in a query reduces its performance from 30% (resp. 25%). This rather good behavior is explained by the intrinsic indexed nature of the three considered models. As stated in Section 4.1, the DS model naturally implements a unidirectional join index from S.b to R.a, where S.b represents a foreign-key and R.a the corresponding primary key (R.a is considered as a domain since it does not contain any duplicate). RS and RIS do nothing but making this index bi-directional. The best join ordering privileges traversals from foreign to primary keys, allowing to produce each result tuple of a query involving N joins by traversing only N-1 domain pointers. This explains the low performance gap between RIS/RS and DS in the figure. However, this favorable situation cannot always be reached, typically if selections impose selecting another join ordering. This point is further discussed below.

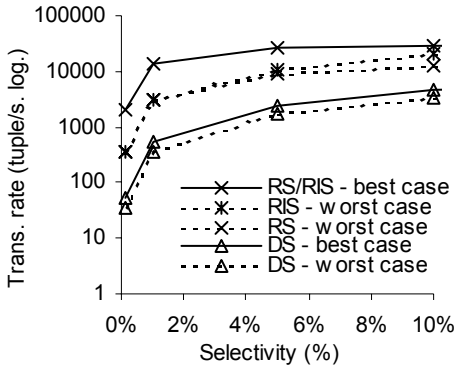
- *Low join transmission rate for FS:* not surprisingly, FS exhibits poor performance comparing to the three other models. Moreover, FS scales very badly as the database size augments (see Figure 48(b)).



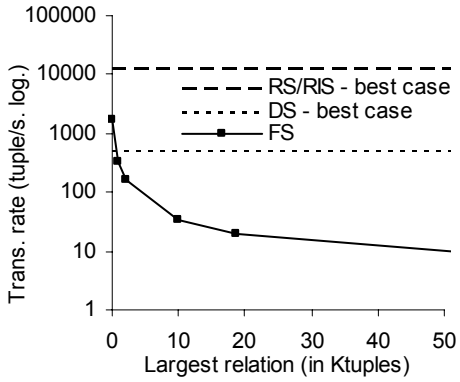

*(a) DS/RS/RIS Model.* *(b) FS Model.*

**Figure 48** *Pure Join Queries Transmission Rate.*

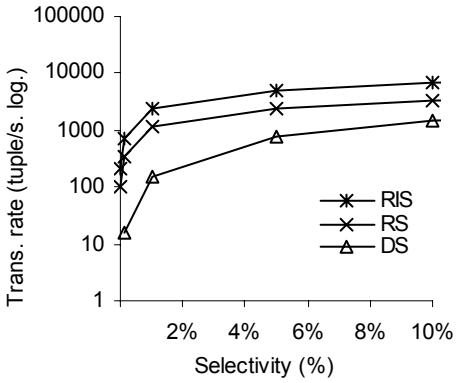
Joins and selections are usually combined in the same query. To capture this situation, Figure 49 measures the performance of SPJ and SPJ$^n$ queries. The transmission rate is plotted varying the query selectivity for RIS, RS and DS models (these models have been shown independent of the database cardinality), and varying the database cardinality for FS (with a default selectivity of 1%). As selections may apply to different relations, the query plans generated may favor different models depending on the join ordering selected. To capture this, Figure 49 plots the upper and lower bound of the transmission rate for each storage model.



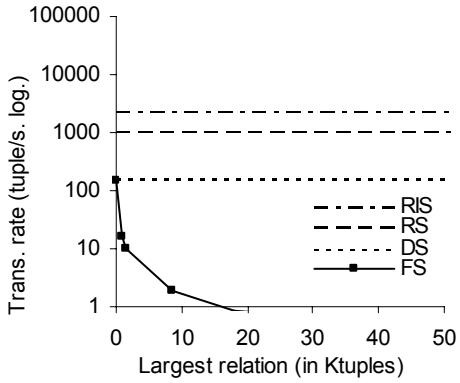

*(a) Single Join with Selection (SPJ) – DS/RS/RIS.*




*(b) Single Join with Selection (SPJ) – FS.*




*(c) Multi-joins with Selections (SPJ$^n$) – DS/RS/RIS.*



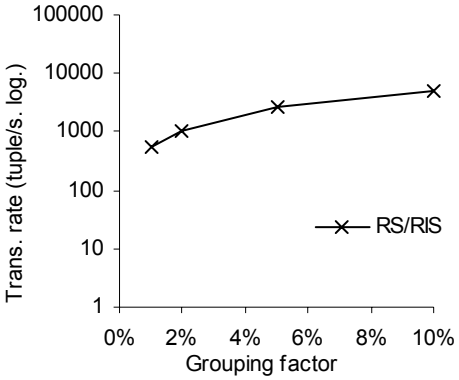

*(d) Multi-joins with Selections (SPJ$^n$) – FS.*

**Figure 49** *SPJ and SPJ$^n$ Queries Transmission Rate.*

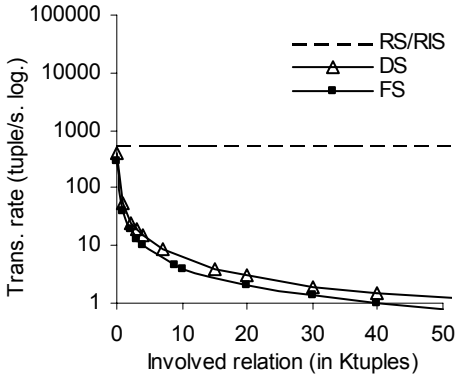
While self-explanatory, these curves deserve the following remarks. First, the performance of the join increases while the selection selectivity decreases, whatever be the storage model. As for selection queries, the worse the query selectivity, the less irrelevant tuples scanned and checked before producing a matching tuple. Second, RS and RIS clearly outperform DS when selections are considered. As stated before, DS imposes a unique join ordering in the query plan, thereby precluding the use of a selection index. For that reason, the choice of RS or RIS to store the foreign key attributes is basically preferred.

### *5.4.6 Aggregates*

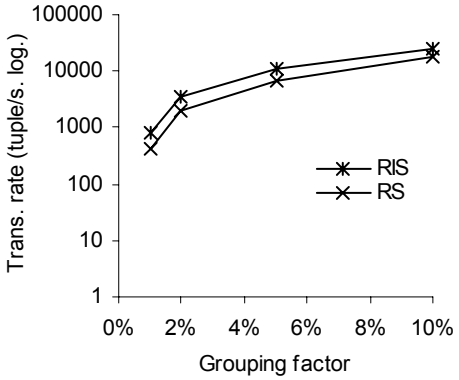
Aggregate queries implement computed data authorizations (CA), granting access to computed values without granting access to the occurrences taking part in the computation process, a rather usual and important class of authorizations. Figure 50 presents the transmission rates of aggregates queries representative of the SPG, SPJG and SPJG$^n$ access rights (see Table 11).



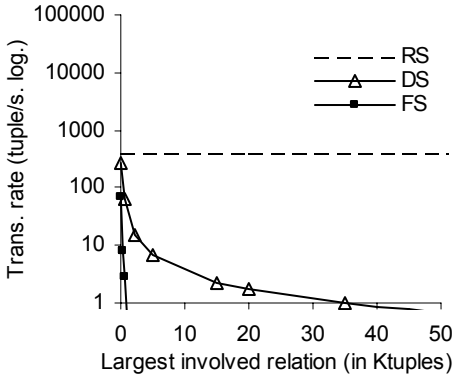

*(a) Mono-aggregation (SPG) – RS & RIS.*




*(b) Mono-aggregation (SPG) – FS/DS.*




*(c) Mono-agg., Joins (SPJG) – RS/RIS.*




*(d) Mono-agg., Joins (SPJG) – FS & DS.*

*(e) Multi-agg., Joins (SPJG$^n$).*

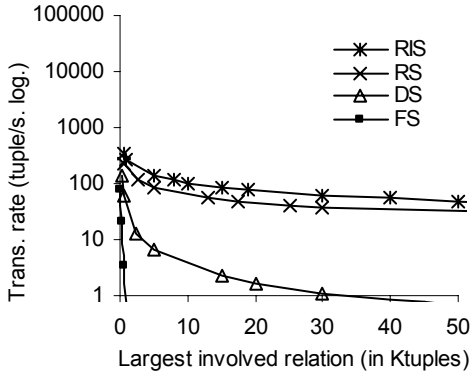


**Figure 50** *Transmission Rate of Aggregation Queries (SPG, SPJG, SPJG$^n$).*

This curves lead to the following remarks:

- *RIS and RS support mono-attribute aggregations gracefully*: the query evaluation starts from the domain containing the grouping values then traverse one or more rings (depending on the presence of joins) to get the tuples sharing the current grouping value. Thus, thanks to rings, the transmission rate for mono-aggregate queries (Figure 50(a) and (c)) is roughly comparable to non-aggregative queries. As for join queries, note that the performance is independent of the database cardinality, however the cost depends on the grouping factor. For example, a 10% grouping factor means that the query output contains 10 times less tuples after aggregation, inducing a factor 10 performance loss compared to non-aggregative query.

- *DS and FS support aggregations badly:* the transmission rate is poor for both models, even for mono-attribute aggregations, and strongly depends on the database cardinality. Indeed, DS induces a Cartesian product between the domain the aggregation applies on and the relation, while FS induces successive sequential scans of the relation to produce the result. When joins are combined in the query, the performance of DS remains stable (thanks to a high join delivery) while FS collapses (under one tuple per second).

- *RIS and RS support multi-attribute aggregation reasonably*: as pictured in Figure 50(e), the transmission rate depends now on the database cardinality for all storage models. Indeed, even with RS and RIS, only one of the attributes involved in the group by clause benefits from the ring storage. However, the performance remains acceptable with RS and RIS since more than 80 result tuples per second are still delivered.

### *5.4.7 Conclusions Drawn on Transmission Rates*

As highlighted by the preceding sections, the transmission rate that can be reached for each class of queries is highly dependent on the selected storage model. To help comparing these models, Table 12 expresses their relative performance in terms of ratio for all classes of query. In each line of Table 12, the gray cell served as a reference to establish the ratios. The transmission rates have been computed considering 1000 tuples in relation $R_0$.

DS gives the best transmission rate in terms of insertion and outperforms FS to process queries involving joins (from one to three orders of magnitude). Adding rings gives also a major benefit for join and aggregate computations (again an improvement from one to two orders of magnitude wrt DS). However, the benefit provided by RIS compared to RS is rather disappointing (at most a factor 2 when several joins participate in a query), especially when considering the loss it incurs in terms of database compactness. Nevertheless, note that for particular database schemas, like star schemas where a central large relation is referenced

by several small relations through large rings (*e.g.*, hundred of pointers), RIS model could become valuable.

| *Operation's type* \ *Storage model* | | *FS* | *DS* | *RS* | *RIS* |
|---|---|---|---|---|---|
| *Insert query* | | **1** | 4,8 | 1,1 | 0,7 |
| *Select query* | *P* | **1** | 0,9 | 0,5 | 0,9 |
| | *SP* | **1** | 0,9 | 8,7 | 8,7 |
| | *SPJ* | 0,08 | **1** | 23,4 | 23,4 |
| | $SPJ^n$ | 0,01 | **1** | 7,1 | 15,2 |
| | *SPG* | 0,7 | **1** | 136 | 136 |
| | *SPJG* | 0,002 | **1** | 118 | 228 |
| | $SPJG^n$ | 0,002 | **1** | 19 | 31 |

**Table 12** *Transmission Rate Ratios.*

# 6 Conclusive Remarks and Case Studies

This section gives conclusive remarks of the measurements. To do so, it puts into practice the conducted performance evaluations and shows how the preceding results can be interpreted to help selecting the best storage model for a given application. Two application's profiles are considered below. The first one is concerned about the total execution time for a query, an important parameter when the result tuples cannot be delivered in pure pipeline (*e.g.*, an order by clause applies to the final result). For this profile, we assume a response time requirement of one second[35]. The second profile is more concerned about the PicoDBMS throughput. Typically, this situation occurs in smart phones applications where the device display must be filled in at a minimum speed. For this profile, we assume a minimum transmission rate of 10 result tuples per second. For both profiles, we concentrate on retrieval queries since the tuple insertion rate has been shown acceptable (see Section 5.4.1) whatever be the considered storage model.

Let's first consider the application's profile concerned about the total execution time for a query. Table 13 presents the percentage of the total result that can be produced per second for each class of query, depending on the database cardinality and on the selected storage model. Note that the database cardinality is itself determined by the smart card storage capacity and by the storage model of interest. For instance, the potential database cardinality for a 1 megabytes smart card ranges from 24K tuples (with FS) to 44K tuples (with DS). The cells sustaining the application's requirement appear in gray. Table 13 shows that FS cannot fulfill the application's requirement for aggregate queries, even for a small 64 kilobytes smart card (only 11% of the result being delivered in 1 second). While DS fulfills this

[35] This is typically the case for applications involving human interactions.

requirement for 64 kilobytes smart cards, RS and RIS are necessary for larger smart cards. As mentioned in the previous section, the gap between RS and RIS is not significant in this context.

| *Card Capacity* | *Total tuples* | *SP* | *SPJ* | *SPJ$^n$* | *SPG* | *SPJG* | *SPJG$^n$* |
|---|---|---|---|---|---|---|---|
| | | | | *FS* | | | |
| 64KB | 1428 | 100% | 100% | 100% | 100% | **11%** | **11%** |
| 128KB | 2978 | 100% | 100% | **39%** | **95%** | **0%** | **0%** |
| 512KB | 12293 | 100% | **40%** | **1%** | **5%** | **0%** | **0%** |
| 1MB | 24864 | 100% | **9%** | **0%** | **1%** | **0%** | **0%** |
| | | | | *DS* | | | |
| 64KB | 1906 | 100% | 100% | 100% | 100% | 100% | 100% |
| 128KB | 4593 | 100% | 100% | 100% | **59%** | **61%** | **52%** |
| 512KB | 20590 | 100% | 100% | 100% | **2%** | **2%** | **2%** |
| 1MB | 44730 | 100% | 100% | **48%** | **1%** | **0%** | **0%** |
| | | | | *RS* | | | |
| 64KB | 1637 | 100% | 100% | 100% | 100% | 100% | 100% |
| 128KB | 4015 | 100% | 100% | 100% | 100% | 100% | 100% |
| 512KB | 18250 | 100% | 100% | 100% | 100% | 100% | **43%** |
| 1MB | 39650 | 100% | 100% | 100% | 100% | 100% | **13%** |
| | | | | *RIS* | | | |
| 64KB | 1247 | 100% | 100% | 100% | 100% | 100% | 100% |
| 128KB | 2829 | 100% | 100% | 100% | 100% | 100% | 100% |
| 512KB | 12736 | 100% | 100% | 100% | 100% | 100% | 100% |
| 1MB | 26426 | 100% | 100% | 100% | 100% | 100% | **41%** |

**Table 13** *Percentage of the Total Result Produced in 1 Second.*

Let's now consider the application's profile concerned about transmission rate. Table 14 presents the number of delivered result tuples per second for each storage model, under the same conditions as Table 13. The cells sustaining the application's requirement appear in gray[36]. The DS model fulfills the application's requirement for smart cards holding up to 128 kilobytes of stable memory. For larger databases, the RS model is required and again RIS does not provide a significant advantage.

[36] They contain the value ">10" when they comply with the requirement, even if the total number of produced result for a particular query is in reality above 10.

| *Card Capacity* | *FS* | | | | | | |
|---|---|---|---|---|---|---|---|
| | *Total tuples* | *SP* | *SPJ* | *SPJ$^n$* | *SPG* | *SPJG* | *SPJG$^n$* |
| 64KB | 1428 | >10 | >10 | >10 | >10 | **1** | **1** |
| 128KB | 2978 | >10 | >10 | >10 | >10 | **0** | **0** |
| 512KB | 12293 | >10 | >10 | **2** | **4** | **0** | **0** |
| 1MB | 24864 | >10 | >10 | **0** | **2** | **0** | **0** |
| *Card Capacity* | *DS* | | | | | | |
| | *Total tuples* | *SP* | *SPJ* | *SPJ$^n$* | *SPG* | *SPJG* | *SPJG$^n$* |
| 64KB | 1906 | >10 | >10 | >10 | >10 | >10 | >10 |
| 128KB | 4593 | >10 | >10 | >10 | >10 | >10 | >10 |
| 512KB | 20590 | >10 | >10 | >10 | **4** | **3** | **3** |
| 1MB | 44730 | >10 | >10 | >10 | **3** | **2** | **2** |
| *Card Capacity* | *RS* | | | | | | |
| | *Total tuples* | *SP* | *SPJ* | *SPJ$^n$* | *SPG* | *SPJG* | *SPJG$^n$* |
| 64KB | 1637 | >10 | >10 | >10 | >10 | >10 | >10 |
| 128KB | 4015 | >10 | >10 | >10 | >10 | >10 | >10 |
| 512KB | 18250 | >10 | >10 | >10 | >10 | >10 | >10 |
| 1MB | 39650 | >10 | >10 | >10 | >10 | >10 | >10 |
| *Card Capacity* | *RIS* | | | | | | |
| | *Total tuples* | *SP* | *SPJ* | *SPJ$^n$* | *SPG* | *SPJG* | *SPJG$^n$* |
| 64KB | 1247 | >10 | >10 | >10 | >10 | >10 | >10 |
| 128KB | 2829 | >10 | >10 | >10 | >10 | >10 | >10 |
| 512KB | 12736 | >10 | >10 | >10 | >10 | >10 | >10 |
| 1MB | 26426 | >10 | >10 | >10 | >10 | >10 | >10 |

**Table 14** *Number of Results Tuples Produced in 1 Second.*

In the light of these two tables, we can draw the following global conclusions. Selection and projection queries are never a bottleneck, whatever be the considered storage model. The main issue is thus supporting efficiently joins and aggregations. To meet this goal, RS is generally required, except for small capacity smart cards where DS can be acceptable. The final question is about the generality of these conclusions. In other words, could we devise other storage and indexation models that may drastically change these conclusions? Our conviction is that this study covers the main trade-offs in terms of storage and indexation techniques, namely direct storage (flat) vs. compressed storage (domain) and indexed vs. non-indexed structures to speed up selections and joins. Of course, variations of the studied techniques may be considered but the expected variation in terms of performance is rather low in a main memory like context. The minimum gap between RIS and RS is nothing but a confirmation of this allegation. At most, one could devise multi-attribute indexation

structures to speed-up multi-attribute aggregate queries, a particular form of pre-computed queries.

# Chapter 5 – Coping with Storage Limitation for Smart Card RDBMS

*This chapter investigates cryptographic and query processing techniques which allow keeping the same security level as PicoDBMS while externalizing the database in an untrusted storage. This work has been done in the last months of my PhD; some issues are thus partially addressed and need some further work.*

## 1 Introduction

New stable memory technologies, like Millipedes [VGD+02], could bring high storage capacities in a tiny space, thus making it a candidate of choice for smart cards. However, the integration of nanotechnologies in smart cards, and more generally in secure devices, is not expected to be available in a mid term. Indeed, these technologies are still in the development phase (not production phase). In addition, the cost limitation of smart cards and their high security requirements delay the integration of new technologies. Thus, the amount of secure stable storage will remain bounded to hundreds of kilobytes for a while, thereby limiting the database size that can be managed on-board.

Fortunately, smart cards are plugged into or linked to more powerful devices and may even embed some additional memory modules, nevertheless insecure. We can therefore envision taking advantage of these storage and processing capabilities to overcome the smart card storage limitations. Several architectural options may be considered:

(i) *Embedded:* this option includes emerging smart card based architecture where a secure smart card chip is connected to large but insecure memory modules. For instance, in the Gemplus' SUMO "smart card", FLASH memory modules are dispatched within the plastic's substrate, thus providing 224 megabytes of stable storage. Another example is the MOPASS consortium [MOP04], which proposes to combine FLASH memory cards with a smart card.

(ii) *Closely connected:* in that case, the external memory is located on the host device (e.g., cell phone, PDA, laptop, or set top boxes). To be practical, the smart card should have a high bandwidth communication link with the device (e.g., USB).

(iii) *Client / Server:* The smart card is connected, through a host device, to a remote device acting as an external storage. Compared to the previous one, this architecture allows accessing the data from any host device connected to the network and brings ubiquity and durability since the database can be accessed from anywhere and can be easily backed up.

Using the embedded, closely connected, or remote storage for storing the database removes the inherent storage limitation of smart cards and thus extends the scope of potential PicoDBMS's applications. Since the data is stored externally in an untrusted environment, its confidentiality can be compromised and its contents can be altered (even if encrypted). This motivates the investigation of cryptographic and query processing techniques that allow externalizing the database on an untrusted storage while keeping the same tamper-resistance as smart card databases.

The rest of the chapter is organized as follow. First, we present the related works in Section 2. The problem is stated in Section 3 and the approach to solve it is depicted in Section 4. Section 5 is devoted to cryptographic techniques while Section 6 focuses on query processing. Finally, Section 7 concludes and proposes a list of further issues that should be addressed soon.

## 2 Related Works

Client-based access control approaches [HIL02, BoP02, DDJ+03] have been proposed recently in the relational context. In these approaches, the data is kept encrypted at the server and decrypted on the client. In [HIL02, DDJ+03], the client is the database owner (the database is private) while [BoP02] considers data sharing and dynamic access rights. To achieve reasonable performance, most part of the query processing is performed by the server on the encrypted data. However, such strategies imposes to add fuzzy information called indices [HIL02, DVJ+03] or to rely on encryption properties (i.e., $A=B \Leftrightarrow E(A)=E(B)$) [BoP02]. These approaches are not suited to our requirements since they grant a certain trust to the database server assuming that:

(i) The server behaves correctly, i.e., returns a correct answer to a given query. An incorrect behavior, induced by a malicious user, may entail confidentiality (e.g., Bob modifies the query transmitted from the SOE to the server in order to retrieve more data than allowed);

(ii) Users may not tamper (even randomly) the database in order to access unauthorized data. A malicious user may exchange chunks of data in the database file to access unauthorized data or mislead the access right controller by inserting random values in the database (e.g., Bob, allowed to access customer with age > 20, modifies randomly the cipher text storing the age);

(iii) The database footprint may not be analyzed to infer information from the encrypted data or from the additional fuzzy information. [DVJ+03] points out this last problem and proposes to balance confidentiality requirements with efficiency by using more or less fuzziness in the additional indices.

In the Operating System community, several techniques for securing file systems on an untrusted storage have been proposed [KRS+03, SHS01, MaS02]. These studies are based on cryptographic protocols and thus are obviously related with our problem. However, the proposed solutions manage large files and do not support query processing.

Finally, [MVS00, Vin02] propose techniques for "securing database systems". However, they actually focus on transactional features (ie., atomicity, durability) and not on query processing. Moreover, their solutions do not scale to large databases. In fact, the authors envision applications that do not need query processing and work on a relatively small amount of data (e.g., DRM metadata and rules management [ShV02, VMS02]).

# 3 Problem Formulation

## 3.1 Reference Architecture

The reference architecture depicted on Figure 51 includes three elements, depending on the level of security assigned to each one. In the following, we describe these elements and then instantiate this reference architecture in some of the architectural contexts presented in the Introduction.

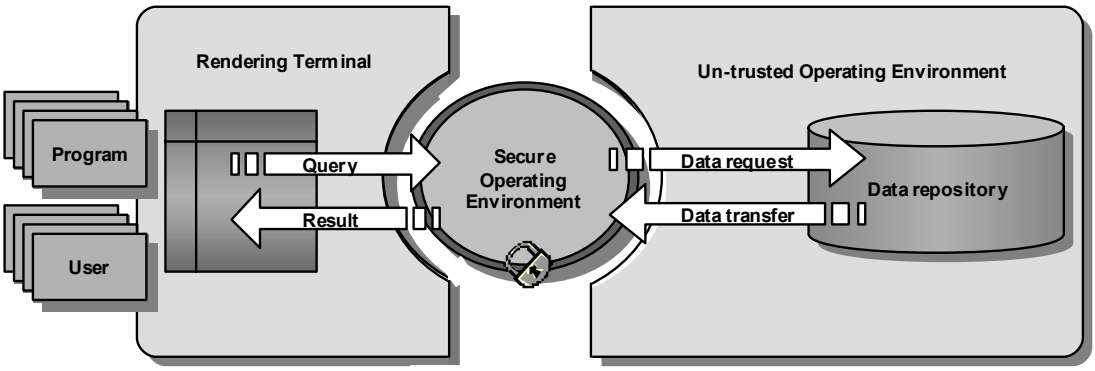


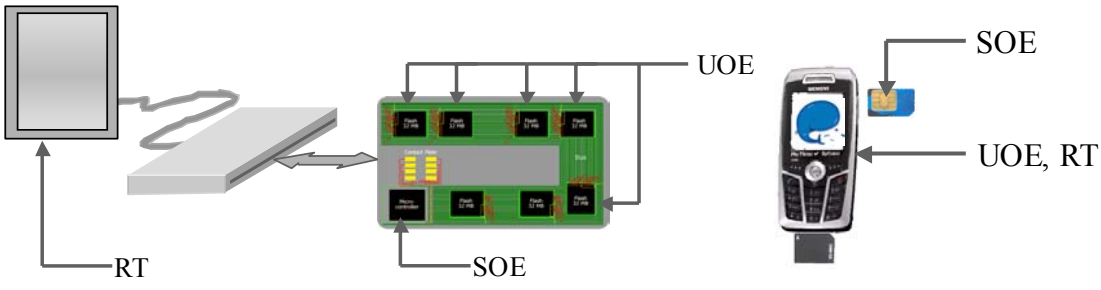

**Figure 51** *Reference Architecture.*

The *Secure Operating Environment* (SOE) is the unique element of trust. Within the SOE, processing and data are trusted, secured and hidden to any attacker. In the context of this study, the SOE is a secured chip (e.g., a smart card) endowed with a reasonable amount

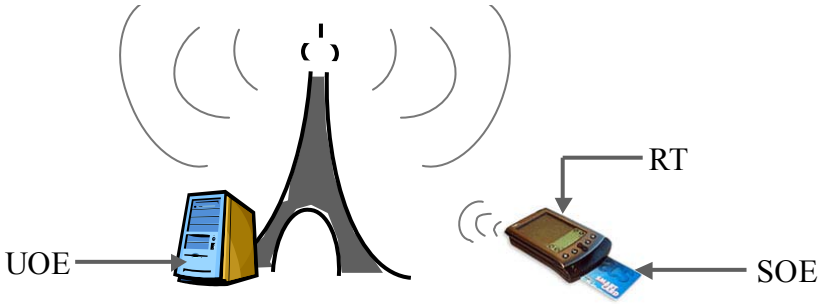
of RAM (e.g., 16 kilobytes)[37].

Conversely, the *Untrusted Operating Environment* (UOE) may be the focus of attackers. The UOE does not offer any guarantee on the data it holds, neither on the processing it does, thus requiring the techniques that will be presented in Section 5.

Finally the *Rendering Terminal* (RT) is the mean by which the query result is delivered to the user. Thus, the RT has obviously the same access rights as the user; otherwise the RT could not deliver the results. The RT should not have access to unauthorized data or temporary results since the user could tamper the RT.




*(a) Embedded.* *(b) Closely Connected.*




*(a) Client/Server.*

**Figure 52** *Architecture Instantiations.*

Let us now consider some examples illustrating this reference architecture. With the SUMO card, the UOE is constituted of disseminated insecure memory modules while the RT is the physical device where the card is plugged in (see Figure 52(a)). Considering a smart card plugged in a cell phone, the RT and the UOE may be the same physical component, i.e., the cell phone; the UOE may also be the memory card, the RT being the cell phone (see Figure 52(b)). Finally, in the Client/Server context, the UOE is the server holding the database, the RT being the physical device where the card is plugged in (see Figure 52(c)).

[37] While RAM is a scarce resource and smart card makers try to reduce it to its minimum (see chapter 3), dedicated smart cards equipped with a significant amount RAM (typically 16KB) have already been produced to face specific applications.

## 3.2 Attacks

With the PicoDBMS solution, the data and the query processing (or at least the view processing) are confined inside the smart card which confers a high tamper resistance. Indeed, the potential attackers, including the cardholder, cannot tamper the data neither corrupt the query processing. Externalizing the data on the UOE allows several attacks, directed on the data at rest in the UOE or on the externalized query processing (including the exchanged messages between the UOE and the SOE). Adapting the terminology of [BoP02], we can classify the attackers as

- *Intruder:* a person or software who infiltrates the system and tries to extract valuable information from the externalized data and/or processing.
- *Insider:* a person or software properly identified by the SOE who tries to read or modify information exceeding her own access rights.
- *UOE administrator:* a person who has all privileges to administer the UOE and thus may conduct easily any attack on the UOE.
- *Thief:* a person who steels the system (or part of it). Obviously, this is more likely to happen when dealing with small devices (e.g., SUMO, MOPASS, cell phone, PDA) than with servers.

An insider has many means and many motivations to attack the database. Let us take some examples in the medical folder context considering that the insider is the cardholder himself. These examples could be transposed to several other contexts where the "user" can be any insider (it could also be software in the case of e.g., a virtual home environment application).

*Examining the data at rest*, the malicious user may infer some information on unauthorized parts of his folder. The user may also *alter the data stored on the UOE*, for instance, to self prescribe some drugs. Remark that even if the data is encrypted, the user may alter it, even randomly. The user may as well *replace a new version of some data by an old one* in order to make the system "forget" something (e.g., to be refund several times by the mutual insurance company). Attacking the communication link, the user can *substitute the data expected by the SOE by another data* in order to gain access to unauthorized part or to modify the behavior of the medical application. As a final, and more subtle example, let us assume that the user is granted a privilege on an aggregate value without being granted the right to see the elementary data participating in this aggregation; the malicious user can *delete all but one elementary data* on the communication link, thus obtaining elementary data values one after the other. While attackers other than insiders may conduct the same kind of attacks, their motivation is generally the examination of the data at rest in order to deduce some unauthorized information.

We can classify the different means of attacks in 4 classes:

- *Data Snooping*: An attacker examines the data and deduce some information
- *Data Altering*: An attacker deletes or modifies (even randomly) some data
- *Data Substituting*: An attacker replaces valid data with another valid data
- *Data Replaying*: An attacker replaces valid data with its older version

These attacks can either be directed on the data at rest in the UOE or on the externalized processing, including on the data exchanged on the communication link between the SOE and UOE.

## 3.3 Security Objectives

As mentioned in the Introduction, our goal is to achieve a security level as high as the one obtained using PicoDBMS. However, the security level is undoubtedly reduced as soon as part of the system stands on an untrusted party (the UOE), potentially controlled by an attacker (e.g., the UOE administrator). We detail below each dimension of this objective.

Compared to the PicoDBMS, we must ensure that externalized data and externalized processing (i.e., processing done on the UOE and message exchanged between the SOE and the UOE) are *opaque* and *tamper resistant*. By opaque, we mean that an attacker cannot deduce any information observing the data or the externalized processing. By tamper-resistant, we mean that an attacker cannot alter the data or the externalized processing. As mentioned before, opacity and tamper-resistance are ensured in PicoDBMS by embedding the database and the processing in a tamper-resistant chip.

Regarding the e*xternalized data*, cryptographic techniques may provide such opacity and tamper resistance (see Section 3 of Chapter 2); however care should be taken to minimize the overhead induced by their use. This problem is addressed in the following sections.

The UOE being untrusted, the honesty of any *externalized processing* has to be checked afterward in the SOE. Actually, such verification might be as expensive as the processing itself. Some techniques indeed exist for simple selections [GKM+04] but the problem remains open for more complex operations. Thus, the tamper resistance of externalized processing seems difficult to attain. Externalized processing opacity seems also unreachable since externalized processing will always disclose some information (even if it is done on encrypted data) which may be valuable for an attacker.

## 3.4 Problem Statement

The problem can be summarized by the three following dimensions:

- *Architectural dimension*: The proposed architecture includes the SOE, the UOE and the RT. Each of these three components has a different level of confidence and different constraints. Therefore, we must precise how to split the query processing and the data storage (including temporary data) in order to reach our security objective.

- *Cryptographic dimension*: Some cryptographic techniques must be provided to protect the externalized data and processing against the four forms of attacks described above, namely data snooping, altering, substituting, and replaying, while minimizing the induced overheads.

- *Query processing dimension*: Specific query processing techniques, coping with the architectural constraints (e.g., smart card constraints), must be designed to allow the dynamic evaluation of complex view expressions (including joins and aggregates – see table 9 in Chapter 4, Section 2) on the externalized and cryptographically protected data. This processing must exhibit acceptable performance, i.e., compatible with the application's requirements.

# 4 Proposed Approach

## 4.1 Architectural Dimension

As mentioned in the previous section, externalizing processing on an un-trusted component reduces the security level since it may disclose valuable information to an attacker. Moreover, checking the integrity[38] of externalized processing is tricky (and costly). To cope with these issues, we take a drastic option which is to reduce externalized processing to its minimum, i.e., the UOE ability is reduced to retrieving data thanks to an address and a size given by the SOE. This leads us to consider a low level interface implementing a retrieve function like *GetData (address, size)→Data*.

The advantages of such an option are that it achieves the highest level of opacity and makes tractable the problem of tamper resistance. Indeed, it only discloses the accesses to the data as well as some information during communication (number and size of the exchanged messages). Providing more opacity appears to be a *Private Information Retrieval* (PIR) [CGK+95] problem, which is out of the scope of this chapter. Moreover, checking the external processing integrity simply amounts to verify that the retrieved data unit has not been altered (data integrity), substituted (originates from the expected address with the expected size), nor replayed (in its latest version).

The dynamic evaluation of the authorized views for the connected user is thus confined

[38] The UOE can give incomplete or false results to trick the access right manager.

within the SOE, except the external data retrieval which is necessarily delegated to the UOE. To do so, the SOE must store at least the database *bootstrap*, encryption keys, and some versioning information (used to protect the external data against replay). Moreover, the stable memory of the SOE could be exploited to cache some frequently accessed data ("hot" data) like the database metadata (schema, views, etc.) and access rights information.

As mentioned previously, the RT holds the same access rights than the connected user. It might therefore (i) complete the evaluation of the authorized views based on authorized data), (ii) compute the query based on the authorized views, (iii) render the final result. For instance, the join of two authorized views $V_1$ and $V_2$ can be processed on plaintext data belonging to $V_1$ and $V_2$ on the rendering terminal without hurting the data confidentiality.

## 4.2 Cryptographic and Query Processing Dimension

While the hardware constraints of the smart card (the SOE) are obviously the same as the one presented in Chapter 4, it impacts differently the design of the embedded query processor since the database is externalised on the UOE.

Indeed, the external data must be decrypted and its tamper-resistance must be checked within the SOE, thereby significantly increasing the data access cost. To give a rough idea of the involved costs on advanced smart cards (e.g., the one used in Chapter 4), the decryption cost alone represents around 50 µs for 8 bytes of data (using triple-DES). Even ignoring all the other costs (tamper-resistance checking, I/O, communication), the data access is yet several hundreds more expensive than accessing the internal memory of the smart card.

This first remark makes irrelevant the techniques proposed in Chapter 3 and part of those used in PicoDBMS (Chapter 4) to address the severe constraints on the RAM in smart cards. Indeed, techniques proposed in Chapter 3 recompute repeatedly every result that cannot be buffered in RAM, thus leading to numerous iterations on the on-board data. In PicoDBMS, aggregation or ad-hoc joins computation also induces several iterations on the on-board data. These techniques were effective because the data access cost was very small (e.g., <100ns).

In Chapter 4, the RAM constraint is also tackled by an extensive use of indices for Select-Project-Join queries. While indices are clearly required in this new context, they do not solve totally the problem unless we materialize every possible view result, thereby hurting data and access right dynamicity[39]. For instance, aggregation calculus and joins considering the result of two selections cannot make use of indices. Without RAM, the problem seems intractable.

On the other hand, the techniques proposed in Chapter 3 and 4 do not make use of any

[39] Remember that the objective is to support dynamic evaluation of authorized views as in PicoDBMS (see Chapter 4, section 2).

external resources. In our context, we may use a swapping area on the UOE or even on the RT if it has some storage capabilities. Obviously swapped data should have the same characteristics (in terms of opacity and tamper-resistance) as the base data. Thus, swapping in and out some data roughly doubles the data access cost.

As a consequence, we propose to use the small quantity of available RAM (e.g., 16 kilobytes) for query processing and to rely on swapping on the UOE or RT in order to use algorithms adapted from the state-of-the-art (e.g., sort-merge join).

To maximize the usefulness of the RAM, we also propose to adopt the following principle: at every step during query processing, the smart card should only retrieve the subset of data strictly necessary to perform that step. For instance, a sort-merge join could be performed considering only key and joining attributes rather than wasting some RAM with other attributes. Indeed, even if these extra-attributes may appear in the result, they are useless for the join operation. Keeping these attributes may incur expensive swapping while delaying their retrieval may allow retrieving less data (because of e.g., the join selectivity).

So far, our approach can be summarized by the three points below:

1. *Make extensive use of indices (selection and join indices)*
2. *Rely on state-of-the-art algorithms using the smart card reduced RAM and external (secure) swapping*
3. *At every step, retrieve the minimal data subset to perform that step*

However, this approach makes sense only if we can provide a low granularity access to the external data. Indeed, using indices for selections or joins leads to access only the interesting subset of tuples (horizontal subset) while point 3) leads to retrieve only the interesting subset of attributes (vertical subset).

We thus add a fourth point to our approach:

4. *Provide low granularity access to the external data*

*Low granularity access* depends not only on the UOE (it may be more difficult with a disk than with a FLASH memory) and the way it is connected to the SOE (i.e., USB, network or bus) but also on the granularity of the cryptographic techniques used.

# 5 Cryptographic Techniques

In the following, opaque and tamper-resistant data are called *Protected Data Units* or PDU. Also, our study often concentrates on external NAND FLASH memory (e.g., SUMO product, memory for cell phones, PDA, FLASH memory cards, etc.), which leads us to borrow FLASH technology terminology in the section. For the sake of clarity, we use in this

chapter the term *Block* to designate a set of *Pages*. To avoid confusion, we use the term *Syllable* to designate minimal unit of data processed together while enciphering/deciphering (e.g., 64 bits is the syllable of Triple-DES algorithm).

While several techniques can be used to protect the external data against attacks, the challenge is to provide methods that are compatible with our requirements in terms of opacity, tamper-resistance, and performance. Moreover, as stated in the previous section, we should provide a *low granularity access* to the external data. This means that the overhead induced by protecting an external data unit should be proportional to the size of the accessed block, i.e., the fixed cost per block should be as limited as possible. We first detail our proposition to build PDUs considering sequentially the four forms of attacks. Then we concentrate on the best way to store the timestamps needed to protect PDUs against replay attacks. To conclude this section, we analyze the cost overhead due to tamper resistance while accessing the database, and the impact on the query execution engine.

## 5.1 Data Snooping

*Protected data units* (PDUs) must be opaque, and thus are encrypted by the SOE with a traditional opaque encryption technique to prevent from data snooping. As explained in Chapter 2, Section 3.1.2, the *Cipher Block Chaining* (CBC) encryption mode enforces data opacity as soon as a unique initialization vector (IV) is used to encrypt the first encryption syllable of each PDU. The uniqueness of this initialization vector can be obtained using for example the physical memory address of the PDU on the external storage medium.

Since it may happen that we are interested only in a part of a PDU, we should be able to decrypt only a subset of a retrieved PDU. With the CBC mode, we can decrypt separately each individual syllable of a PDU supplying the key and the previous encrypted syllable (or the initialization vector for the first syllable). The CBC can be performed with any secure algorithms of the literature, as Triple DES or AES. Note that Triple DES is a good candidate since it is efficiently implemented (in hardware) in most smartcards and works with 64 bits syllables while AES uses 128 bits. Nevertheless, AES will be available soon (in hardware on smartcards) and should provide better performance (see Figure 25 in Chapter 2).

An important remark is that the cost for encrypting/decrypting is linear with the number of syllables involved in the process. Indeed, when the initialization phase of the encryption algorithm is performed (e.g., building three needed sub-keys in case of a Triple DES), the cost of the encryption/decryption function is proportional to the involved number of syllables. In our context, the encrypting/decrypting process uses a secret key stored within the SOE, thus allowing to pre-compute that initialization step.

## 5.2 Data Altering

PDU can be protected against data altering by including a block digest (or checksum) in each

external block, which will be checked when the PDU is retrieved. This digest can be either a Message Detection Code (MDC) or a Message Authentication Codes (MAC) as presented in Chapter 2, Section 3.2.2. Regarding MDC based solutions, *MDC-and-encrypt* (see Figure 26 in Chapter 2) breaks the PDU opacity for the reason that the digest of two identical plaintext blocks are identical. The block integrity must thus be provided by either the *MDC-then-encrypt* or the *encrypt-then-Encrypted MDC* process. However, the first alternative forces to decrypt the whole PDU to check its integrity (MDC is computed on the plaintext), and the second one leads to re-compute the digest plus encrypt it. The fixed extra cost of encrypting the digest can be avoided using MAC based solutions.

We turn therefore into MAC based solutions. The *encrypt-then-MAC* approach is the most appropriate because the digest can be checked directly, without decrypting the PDU. Moreover, some MAC functions like HMAC-SHA-1-96 (see Chapter 2), provide a reduced constant cost. Indeed, using this hash function, the digest of an encrypted block *EB* with a 512 bits key *k* is given by *HMAC(EB) = Truncate(SHA-1(k || SHA-1(k || EB)),96)*. Assuming *SHA-1(k)* is computed once and cached in memory, the remaining computation to perform are *SHA-1(EB)* and *SHA-1(X)*, *X* being a hash value of 160 bits (i.e., the result of *SHA-1(k || EB)*). This reduces the fixed cost per PDU to the computation of *SHA-1(X),* which is roughly 3 times less than encrypting the MDC with Triple DES (see Figure 25 in Chapter 2).

All these methods providing PDU integrity have however been designed for large data (e.g., 1 kilobytes). In case of small PDU (e.g., a 12 bytes plaintext), such techniques result in checksums larger than the data itself. To provide integrity at lower cost, we can rely on the following technique: let us consider a block cipher working on syllables of *x* bits and a plaintext *P* of *y* bits to be protected; we propose to build each plaintext syllable, before encryption, by concatenating the successive *x-r* bits of *P* with a duplication of the *r* first bits of the syllable[40]. Integrity is checked by verifying that the first r bits and the last r bits of the decrypted syllable are equal. The security achieved by this scheme depends on the size of *r*. Indeed, let us first consider a single syllable S; if an attacker modifies a single bit of the encrypted syllable, the chance that the produced syllable remains valid (i.e., that the first r bits of S are equal to the last r bits of S) is $1/2^r$. When dealing with PDU larger than one syllable, the CBC encryption mode will ensure that syllables within the same PDU cannot be reordered or suppressed since this will lead to an incorrect decryption that may be considered valid with the same probability (i.e., $1/2^r$). *r* should be chosen according to the required security level but can be kept relatively small (e.g., 16 or 32 bits) since without the encryption key, the attacker has no mean to test if a tampered PDU can be valid unless he submit it to the smart card, which may be automatically locked after a certain number of detected errors. If some extremely small data has to be stored (e.g., one byte or even less), we should obviously group together several values in one PDU in order to fill the *x-r* bytes, since we cannot decrypt less then x bits. Note also that cryptographic improvement of this integrity protocol for small data might be possible, but need the cooperation of specialists of

cryptography.

The frontier between the two mentioned strategies (adding an HMAC digest or using redundancy) can be determined experimentally, depending on the chosen cipher and hashing function.

## 5.3 Data Substituting and Data Replaying

These two attacks (substituting and replaying) can be avoided by *marking* the PDUs with additional information, which will be checked when retrieved.

To protect PDUs from substituting, we integrate some unique information in each PDU. This information could be, for instance, the physical address on the external memory. Since the SOE access the PDUs via this address, it could easily check the validity of the retrieved block.

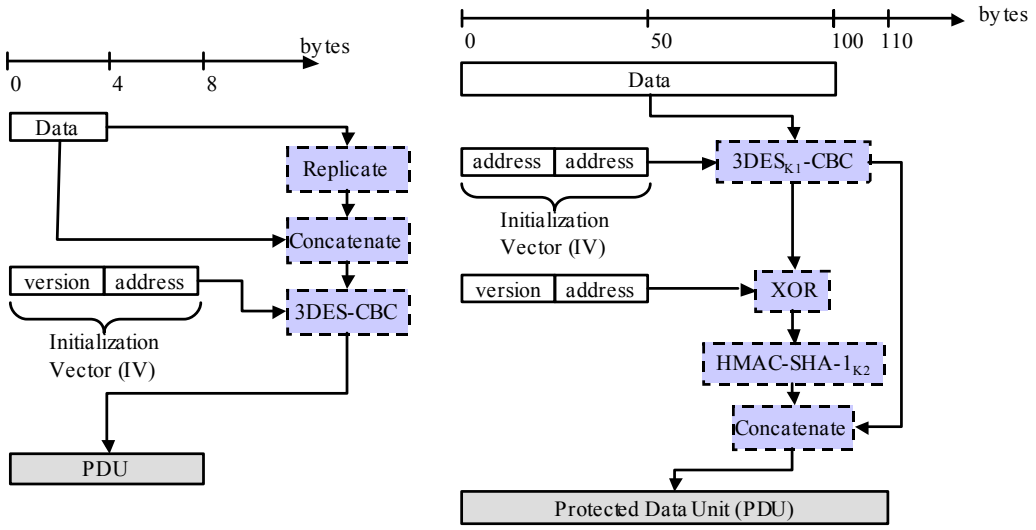
Ensuring resistance against blocks replaying induces to stamp each block with a version number (or timestamp). Unlike for substituting, the timestamp cannot be known in advance by the SOE when retrieving a given PDU. Timestamp should therefore be stored securely (this problem will be further discussed in the next section) and retrieved by the SOE for checking. The timestamp can be always increasing or generated randomly (most smart card include a hardware random number generator). An increasing timestamp will give the attacker the opportunity to replay a bloc that has been updated $2^r$ times. Thus we will prefer a random timestamp of, for instance, 16 bytes giving to the pirate a probability of $1/2^{16}$ to replay randomly an old block without being detected by the smart card. As mentioned above, such a probability can be considered secure in our context.

The way to integrate an identifier and a timestamp at a low cost into a given PDU depends on the integrity method used.

For small PDUs using the redundancy technique presented above, we concatenate the identifier and the timestamp and use it as the initialization vector of the CBC encryption (note that the initialization vector is unique when obtained in this way). Since the whole PDU is systematically decrypted by the SOE, checking the identifier and timestamp comes almost for free (ignoring the cost to retrieve the expected timestamp -see below).

For larger blocks protected with a MAC, such a strategy is inappropriate since it does not allow decrypting only part of the PDU. In this case, the additional information can better be XORed with the encrypted data before computing the MAC of the PDU. Since the HMAC is already checked to ensure the blocks integrity, this additional checking also comes almost for free.

[40] Assuming $r \leq x / 2$




*(a) Very fine grain PDU.* *(b) Larger PDU.*

**Figure 53** *PDUs Generation Process.*

Figure 53 shows the process described above to generate small and larger PDUs.

## 5.4 Timestamps Storage

The expected timestamp associated to each PDU must be known by the SOE to detect replay attacks. This timestamps information is frequently accessed since it is required at each PDU access. A straightforward approach is thus to store in the SOE the timestamp associated to each PDU. While this will provide the best reading and updating performance, it limits drastically the size of the external database, especially when small PDUs are considered.

To solve this issue, we can partition the external memory in large *freshness blocks* and store in the SOE one timestamp for each freshness block. All the PDUs contained in a freshness block share the same timestamp[41]. This strategy provides the best reading performance but imposes to modify all the PDU belonging to a given freshness block when a single byte is updated (the timestamp of all PDUs of the same freshness block evolves). Thus, updating the timestamp of the freshness block induces to re-compute the MAC of each coarse grain PDU and to decrypt/re-encrypt each fine grain PDU. Thus, either the external database is small, either the update performance is prohibitive.

To reduce the update cost while modifying a single byte of a PDU, we could construct a *Merkle Hash Tree* [Mer89] for each freshness block on top of the contained PDUs. While this strategy provides the best update performance, it leads to compute and transmit $log_2(n)$ hash values assuming the freshness block contains $n$ PDUs in addition to the expected PDU, and compute $log_2(n)$ hash operations within the SOE. This prohibitive reading cost excludes

[41] Note that a PDU can belong to several FB. In that case, its timestamp is the one of the first FB.

this solution in our context.

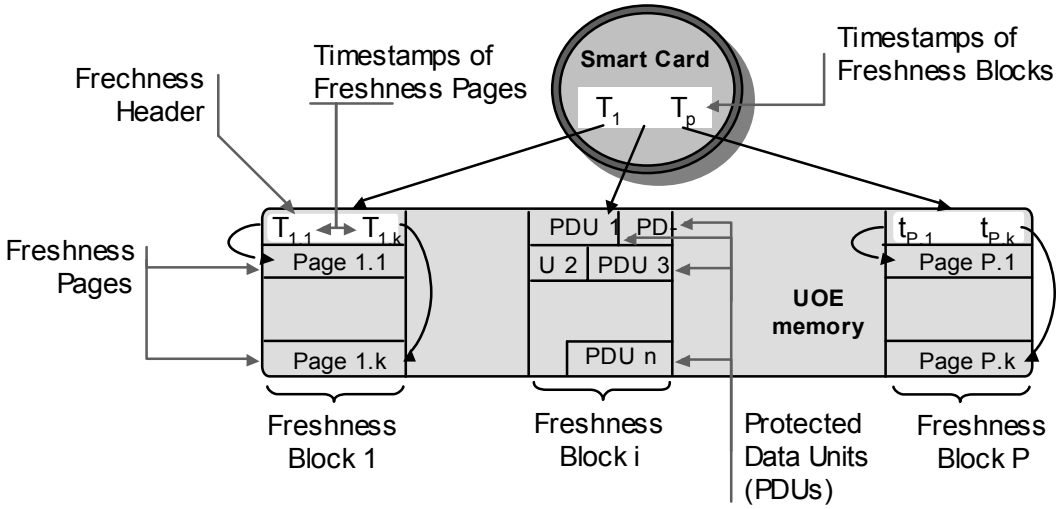



**Figure 54** *Memory Partitioning into Freshness Blocks and Pages.*

We propose a hierarchical solution to store the timestamps. Each freshness block is divided into several freshness pages. The on-board timestamp certifies the freshness of a set of timestamps stored externally in PDUs, each one certifying the freshness of all PDUs belonging to the same freshness page. This alternative is illustrated on figure 54 where all the freshness page timestamps are stored contiguously in a freshness header.

With this two level hierarchy, we can address a large database with acceptable performance. Indeed, reading a PDU requires only one additional access to its externally stored timestamp. Since all PDUs belonging to the same freshness page share the same timestamp, updating a single byte of a PDU induces the freshness modification of all PDUs on the same freshness page, the update of the externally stored timestamp and thus the freshness modification of all timestamp PDUs of the same freshness block.

## 5.5 Preliminary discussion on costs

The purpose of this section is to analyze the cost of accessing the external data in order to understand its major component and to derive some hints for designing the query processor. We will also analyze the impact of considering small PDUs on these costs. Obviously, since this preliminary study considers several hardware architectures, we cannot go through the details of each one.

Let us consider that the SOE wants to access *x* bytes from a *y* bytes PDU. The access costs can be described by the following formula; each component being described below.

*AccessCost* = *I/O cost* (Reading the PDU on the UOE storage medium) +
*Communication cost* (transmitting the PDU to the SOE) +

*Decryption cost* (decrypting the useful part of the PDU) +
*Integrity checking cost* (using one of the two proposed methods) +
*Freshness checking cost* (retrieving the associated timestamps) +
*Identifier checking cost* (checking that it corresponds to the required PDU)

At first glance, we may conclude that considering small PDUs will inevitably increase all these costs since most of them have a fixed component which has to be paid for each PDU. Conversely, the access patterns that we may have during query processing as well as the adoption of simple strategies advocate in favour of small PDUs.

As mentioned in Section 4, our evaluation strategy should make intensive use of indices since the access to external data is expensive and the SOE is constrained. Moreover, the SOE should retrieve only the interesting subset of attributes (at a given moment). Thus the accesses to the external data are *selective* and *random*.

Selective accesses will be much more efficient using small PDUs. Indeed, even considering a small selectivity factor (e.g., 20%), accessing small PDUs will lead to retrieve only the useful data (with a small integrity overhead, e.g., 25% using AES and 32 bits for redundancy) while coarse grain PDUs will induce reading, transmitting, hashing a much larger quantity of data (e.g., five times more data with a selectivity of 20%). This is true horizontally (predicate selectivity) and vertically (subset of attributes).

Random accesses are a known problem in disk based database system. Existing solutions include clustering, caching and prefetching techniques; they will benefit both from small and coarse grain PDU accesses as soon as minimal caching is done in the SOE/UOE. On the UOE, we should at least cache the last accessed *block* (the notion of block depends on the architecture) in RAM to reduce the I/O cost in case of several small PDU requests in the same block. In the SOE, the last accessed freshness value should also be cached in RAM since contiguous PDUs are likely to share freshness values.

Finally, one remark should be done on the communication component of the cost. Generally, communication has a coarse granularity (e.g., 1 KB) and transmitting less data does not diminish the cost. This remark has two impacts. First, we should not use indexation techniques as the one proposed in PicoDBMS (e.g., ring) since it would incur a large number of query/answer to the UOE, thus increasing the communication cost. As an example, the list of *visits* identifiers associated to a given *doctor* should not be stored as a chained list (e.g., rings) but in a single coarse grain PDU (since all the visits of the given doctor should be useful). Second, this could also advocate for coarse grain PDUs for index nodes. Indeed, considering B-Tree as an example, the SOE could retrieve less data if each key in the B-Tree is stored in a small PDU, however this would increase the communication cost during the index probing.

All these consideration makes clear that a careful study of each particular architecture, access patterns and data structure should be done in order to design an adequate storage

model for the external data.

# 6 Query Processing

This section presents the first step in the design of query processing in such an environment. The aim of the query processing confined in the secure operating environment is to build the authorized users views. The execution strategy must comply with the smart card constraints presented in Section 4 leading to rely on intensive use of indices and on an external swap area (on the external memory) for intermediate results. This section gives evaluation principles based on two examples: a select-project-join query and a group by query.

## 6.1 Execution Basics, Notations and Assumptions

The query execution principle is driven by the SOE limitations. Since the on-board RAM is limited, the data flow between the operators of the *Query Execution Plan* (QEP) must be reduced to the minimum. Thus, only tuples identifiers are conveyed along the whole QEP. For the sake of clarity, tuples identifiers of a relation *R* are designated by *R.id*. However, *R.id* is not stored as an attribute of the *R* relation within the database. It can be any information allowing the tuple retrieval (e.g., a tuple physical address).

When an operator of the QEP requires additional data to perform its task, the data flow should be enhanced with the strictly needed additional subset of attribute values. To this end, intermediate operators are introduced in the QEP. First, the traditional *project* operator (represented by $\pi$) removes attribute(s) value(s) from each tuple within the data flow since they are not mandatory to perform the next operation. Second, an operator that we call *positive project* (represented by $\pi$+) retrieves the needed attribute(s) value(s) from the database for each tuple of the flow since the next operation to be performed requires them. Of course this strategy could be refined in particular cases. However, we only give in this section an idea of our approach, even though further works remain to be done.

The database considered in the query examples of this section includes the following structures:

- *R* and *S* are two database relations, respectively composed of attributes *a*, *b*, *c* and *d*, *e*, *f, g*;
- $SI_1$ and $SI_2$ are two selections indices respectively built on *R.b* and *S.e*;
- *IJ* is a join index built on the join predicate *R.a* = *S.d*. We consider that *S.d* references *R.a*.

In the whole section, we also make the following assumptions:

- The data storage model allows the retrieval of any required attribute of a given tuple thanks to the identifier of this tuple;
- Join indices allow to retrieve the list of matching tuples identifiers wrt the join criteria with a given tuple, specified by its identifier;
- Selection indices give the identifiers of tuples sharing a key value in a given attribute.

## 6.2 Select-Project-Join Queries

We investigate here two possible QEP to answer the select-project-join query *Q* given below.

```
SELECT  R.c, S.f
FROM    R, S
WHERE   R.a = S.d        AND
        R.b = 'b_value'  AND
        S.e = 'e_value';
```

Assuming that the *R.b = 'b_value'* predicate selectivity is high compared to the one of the join, that is also small compared to the one of the predicate *S.e = 'e_value'*, the query optimizer might choose the QEP pictured on Figure 55 to answer the query. The squares represent the access to the database through the low level interface, the circles represent the physical involved operators, and the edge stands for the dataflow between the operators. These conventions are used all-over the section.

Thus, the following operations are performed:

1. *Selection (index)*: access the *R* tuples identifiers *R.id* matching *R.b = 'b_value'* using index $SI_1$;
2. *Join*: for each *R.id*, retrieve the joining identifiers of S *S.id* using *JI* and produce the matching pairs (*R.id*, *S.id*);
3. $\pi^+$*:* for each (*R.id*, *S.id*), access the value of the *S.e* attribute corresponding to *S.id*;
4. *Selection*: for each (*R.id*, *S.id, S.e*), evaluate the predicate *S.e = 'e_value'*;
5. $\pi$*:* for each (*R.id*, *S.id, S.e*), project (*R.id*, *S.id*);
6. $\pi^+$*:* for each (*R.id*, *S.id*), access the value of the *S.f* attribute corresponding to *S.id,* and possibly (when *R.id* changes) access the value of the *R.c* attribute corresponding to *R.id*;
7. $\pi$*:* for each (*R.id*, *S.id, R.c, S.f*), project (*R.c, S.f*) and externalize it in clear on the rendering terminal, which displays the final result.

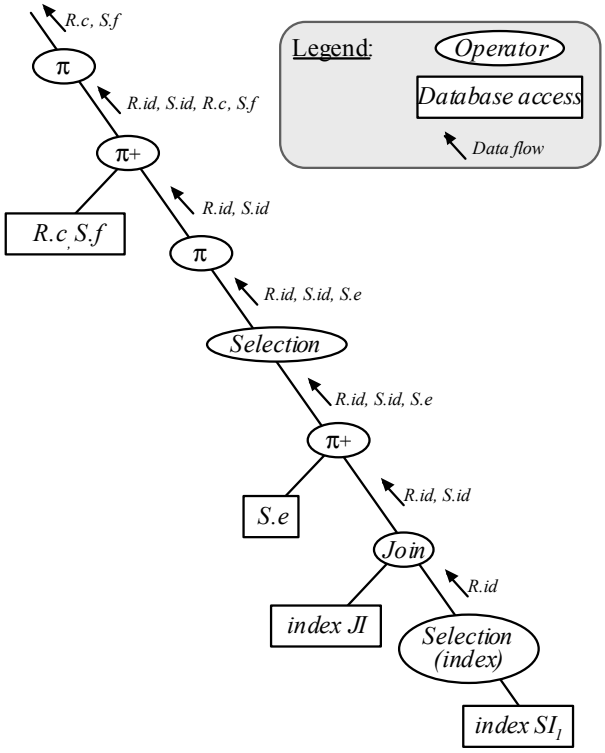



**Figure 55** *Query Execution Plan $P_1$.*

We assume now that the selectivity of the two selection predicates is equivalent and high (few result tuples are produced compared to the cardinalities of *R* and *S* base relations). The query optimizer might choose in this case[42] the QEP $P_2$ pictured on Figure 56.

Thus, the following operations are performed:

1. *Selection (index)*: access the *R* tuples identifiers *R.id* matching *R.b = 'b_value'* using index $SI_1$;

2. $\pi$*+:* for each *R.id*, access the value of the *R.a* attribute corresponding to *R.id*;

3. *Sort*: for each (*R.id, R.a*), sort it in memory on the *R.a* value. When the memory overflows, encrypt the sorted bucket with a private key $K_1$, compute its digest $H_1$ (e.g., using the SHA-1 algorithm) and keep it in the SOE, then write this encrypted bucket $B_1$ to the UOE (see Figure 57). This process is repeated until the whole input flow is consumed. At the end of the *Sort* operation, *x* partially sorted buckets $B_1$, … $B_x$ have been externalized on the UOE, and their corresponding *x* digests $H_1$, …, $H_x$ are kept in the SOE. Note that these digests can also be stored encrypted on the UOE. For the sake of conciseness, we assume here that the SOE disposes of enough

[42] This QEP might also be generated if index *IJ* does not exist.

RAM memory to keep them on-board.

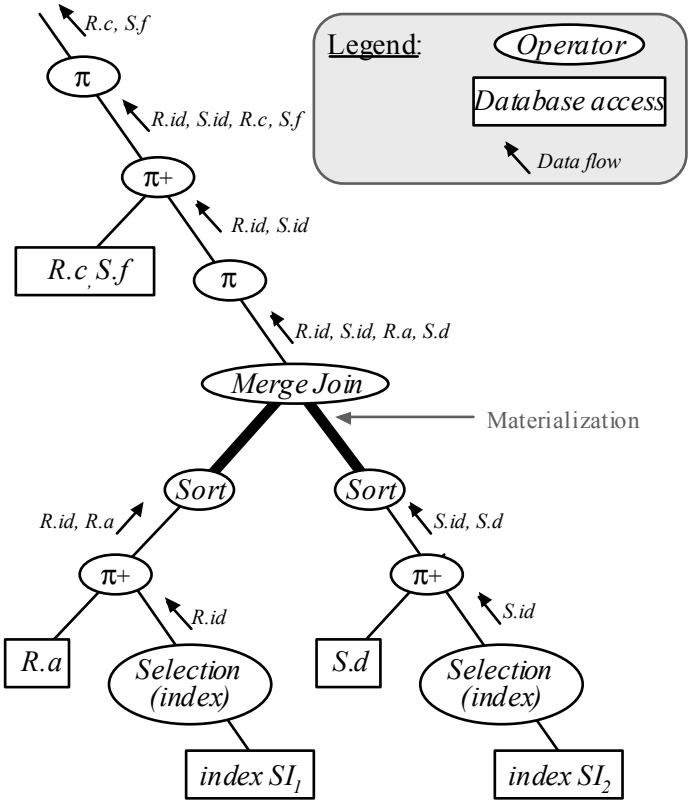



**Figure 56** *Query Execution Plan $P_2$.*

4. *Selection (index)*: access the *S* tuples identifiers *S.id* matching *S.e = 'e_value'* using index $SI_2$;

5. *π+:* for each *S.id*, access the value of the *S.d* attribute corresponding to *S.id*;

6. *Sort*: as previously, sort the input flow pairs (*S.id, S.d*) on the *S.d* value. At the end of this step, *y* buckets $B_{x+1}, \ldots, B_{x+y}$ have been constituted on the UOE and *y* digests $H_{x+1}, \ldots, H_{x+y}$ have been built in the SOE.

7. *Merge Join*: retrieve in parallel the *n* sorted buckets (assuming $n = x+y$) and process the merge join on the *R.a = S.d* predicate. The buckets are accessed one syllable at a time to re-compute incrementally their respective digest $H_1', \ldots, H_n'$. This step can be pipelined, i.e., when produced each result tuple of the join can be consumed by the next operator of the QEP. However, the final result of the query cannot be delivered in clear until the end of the merge step (see Figure 57). Indeed, the retrieved sorted bucket $B_1, \ldots, B_n$ cannot be checked against tampering since $H_1', \ldots, H_n'$ are not computed. At the end of the merge phase, $H_1', \ldots, H_n'$ are compared to their expected value $H_1', \ldots, H_n'$. The evaluation aborts in case of comparison failure.

8. $\pi$: for each (*R.id*, *S.id, R.a, S.d*), project (*R.id*, *S.id*);

9. $\pi$+: for each (*R.id*, *S.id*), access the value of the *S.f* attribute corresponding to *S.id,* and access the value of the *R.c* attribute corresponding to *R.id*;

10. $\pi$: for each (*R.id*, *S.id, R.c, S.f*), project (*R.c, S.f*). Since the merge phase is not completed, store the computed results in the SOE. When memory overflows, encrypt the results with a session key $K_2$ to form a bucket $R_i$, and externalize $R_i$ to the rendering terminal. After the merge phase ends successfully, deliver to the rendering terminal and produce the remaining results in clear. The rendering terminal is in charge of decrypting the encrypted result buckets and displaying the final result.

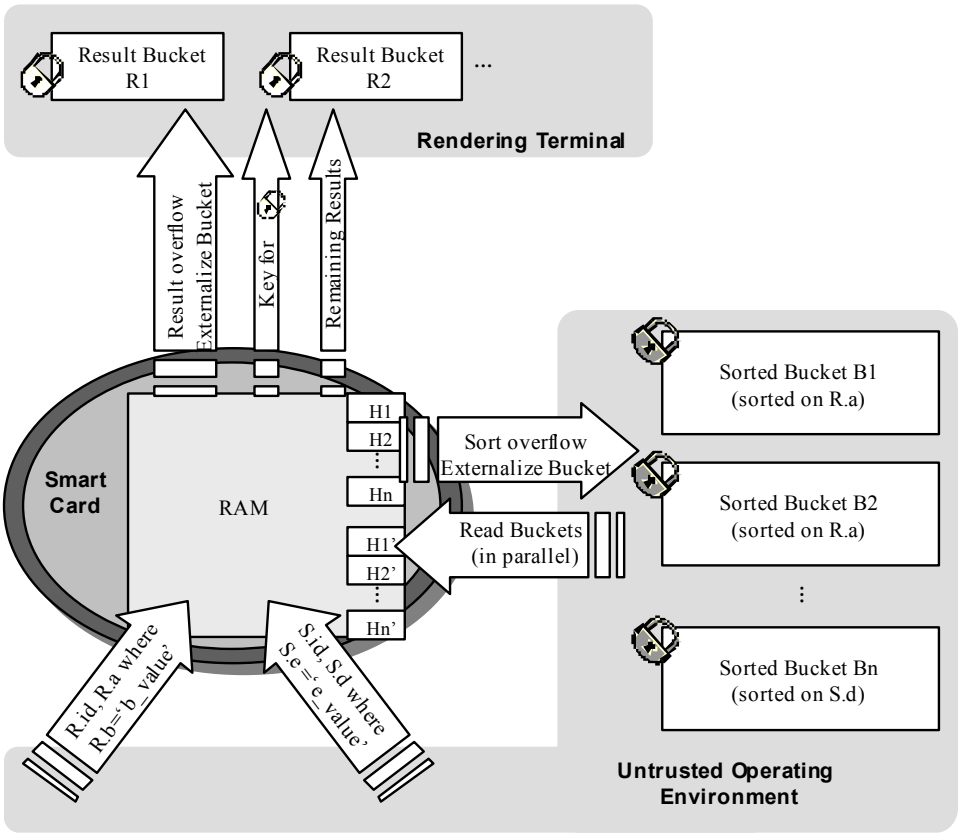



**Figure 57** *Sort Merge Join Process.*

Note that in case of $P_2$, the Sort Merge Join is performed more efficiently whether *R.id* values are kept sorted within the selection index $SI_1$ and the *R.id* values are stored in the *S.d* attribute. Then, the output of the *Selection (Index)* operator accessing $SI_1$ is already sorted on *R.id*, which avoids the following *Sort* operation. However, we do not discuss further these possible optimizations which require further research (see Section 7 for future works).

## 6.3 Group By and Aggregation Operations

We present here the operators needed to group an input flow on one or several attributes, in order to compute some aggregates on each group. We consider the previously defined SQL query *Q* with an additional group by clause and a sum computation as follows.

```
SELECT   R.c, S.f, SUM(S.g)
FROM     R, S
WHERE    R.a = S.d         AND
         R.b = 'b_value'   AND
         S.e = 'e_value'
GROUP BY R.c, S.f;
```

The QEP $P_3$ given by the optimizer to execute the query is pictured in Figure 58. $P_3$ is similar to $P_2$, slightly modified to produce the value of the attribute *S.g*, with on top two additional operators evaluating the group by and the aggregate computation. These two additional operators are discussed here; we refer to the previous section for the explanation of the $P_2$ sub-part of the QEP.

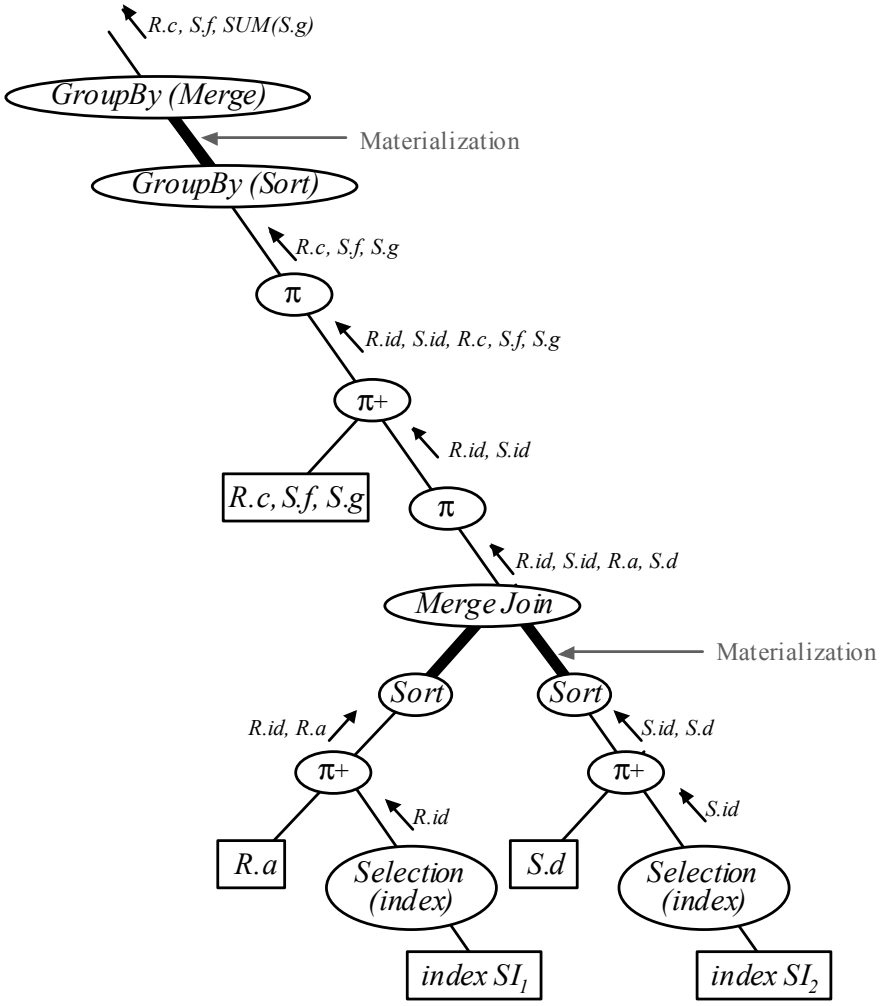



**Figure 58** *Query Execution Plan $P_3$.*

The *GroupBy (Sort)* operator input flow is constituted from triplets (*R.c*, *S.f*, *S.g*) produced by its sub-tree. The role of this operator is to sort the input flow on the grouping key, i.e., (*R.c*, *S.f*), into sorted buckets. Thus, the incoming flow is sorted in memory. To delay the memory overflow, the tuples sharing a value pair (*R.c*, *S.f*) are aggregated. This reduces the tuples in memory to a single occurrence of (*R.c*, *S.f*, *partial_sum*) per grouping key. When memory overflows, the sorted tuples are externalized in a bucket, that must be encrypted. As previously mentioned, a digest of the bucket (e.g., computed with SHA-1) must be kept in the SOE. The process can be reiterated until the whole input is consumed. Finally, $n$ sorted buckets $B_1,\ldots, B_n$ have been build and externalized, and their $n$ respective digest $H_1,\ldots, H_n$ have been computed in the SOE.

The *GroupBy (Merge)* operator retrieves in parallel the $n$ sorted buckets, one syllable at a time to re-compute incrementally their respective digest $H_1',\ldots, H_n'$. The retrieved triplets (*R.c*, *S.f*, *partial_sum*) are merged on the grouping key (*R.c*, *S.f*) and the aggregate is computed. Whether memory overflows, final results (those sharing a grouping key lower than the current processed one) are encrypted with a session key into a result bucket and externalized to the rendering terminal. Note that, as mentioned in the *Merge Join* phase, the plaintext result will be available only after the *GroupBy (Merge)* operator ends with success. Indeed, when all the sorted buckets are consumed and when their respective digests $H_1',\ldots, H_n'$ are compared without failure to their expected values $H_1,\ldots, H_n$, the session key is delivered to decrypt the result buckets and the remaining results are externalized in clear to the rendering terminal. The rendering terminal is in charge of decrypting the encrypted result buckets and displaying the final result.

# 7 Conclusion and Future Works

In this chapter, we have presented an ongoing work aiming at using an untrusted external storage in the smart card context, assuming that the smart card environment often makes it possible. We characterized the additional threats due to data and process externalization, which include data snooping, altering, substituting and replay. We also presented an approach to solve this issue, based on minimizing the processing delegated to untrusted parties, and using cryptographic methods to ensure the tamper resistance of the system. Moreover, we motivated the need for storing the database into small *Protected Data units* (PDUs) in order to minimize accesses to irrelevant sub-parts of the data and to delay the data access to the execution steps which mandatory require them. Then we showed how cryptographic techniques can be used to make the PDU tamper resistant, using techniques including CBC, HMAC, and hierarchical timestamps. Finally, we presented our execution strategy to process queries under the smart card constraints.

This study is still ongoing and many further works remains to be conducted. First, we must study the optimal partitioning of the memory into *freshness pages* and the appropriate

placement of the PDUs holding the data and holding timestamps according to the considered architecture (e.g., UOE memory technology, communication protocols between UOE and SOE, etc.).

The storage model for data, indices, and metadata should be investigated. It must be determined appropriately according to the considered UOE architecture and complying with the proposed query evaluation techniques. For example, a storage model similar to PAX, that would group attribute values in freshness pages, seems suitable (as pictured in Figure 59, with the *R* and *S* relations mentioned in Section 6).

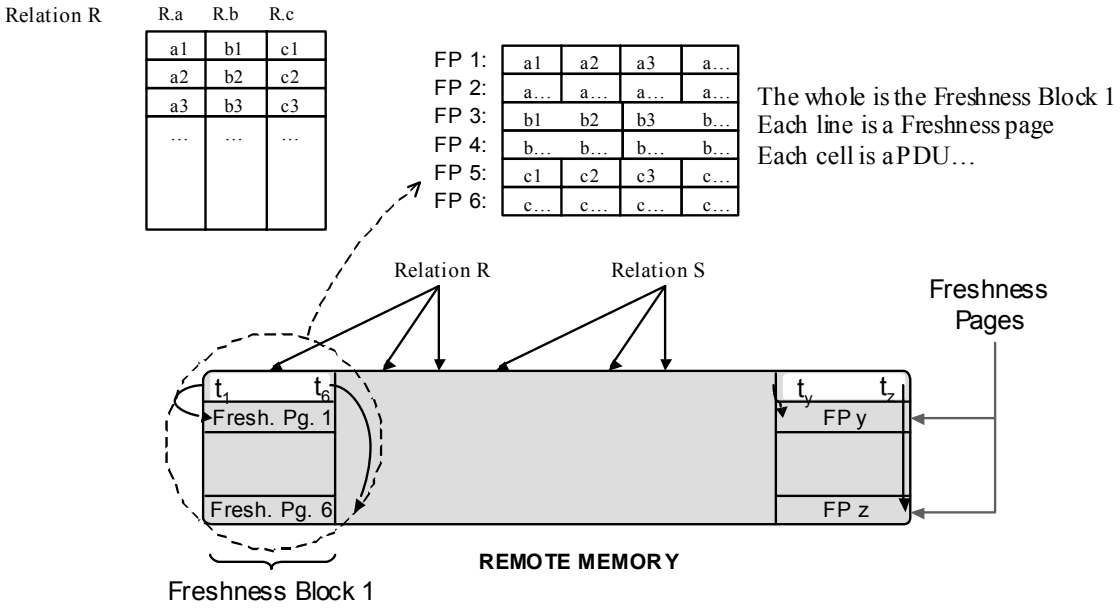



**Figure 59** *PAX as a Storage Model Example.*

When the appropriate indices will be chosen, their storage model must also be determined. For example, the join index *IJ* accelerating the join on the predicates *R.a = S.d* (see Section 6), when applied to a key/foreign key join (assuming *R.a* is the primary key of *R* and *S.d* is a foreign key in *S* referencing *R*), could be stored on the external storage medium as pictured in Figure 60. The join index is here split into three columns of pointers, the first one is stored as a regular attribute of relation R (e.g., using the previously mentioned PAX model), the second is stored as a regular attribute of relation S, and the third one is stored independently of R and S in larger PDUs (one PDU per group of pointers addressing the S tuples matching a given R tuple). Indeed, when the join is processed from *R* to *S*, all identifiers of *S* joining with a given *R* tuple will be retrieved together, which induces to store them in a single PDU.

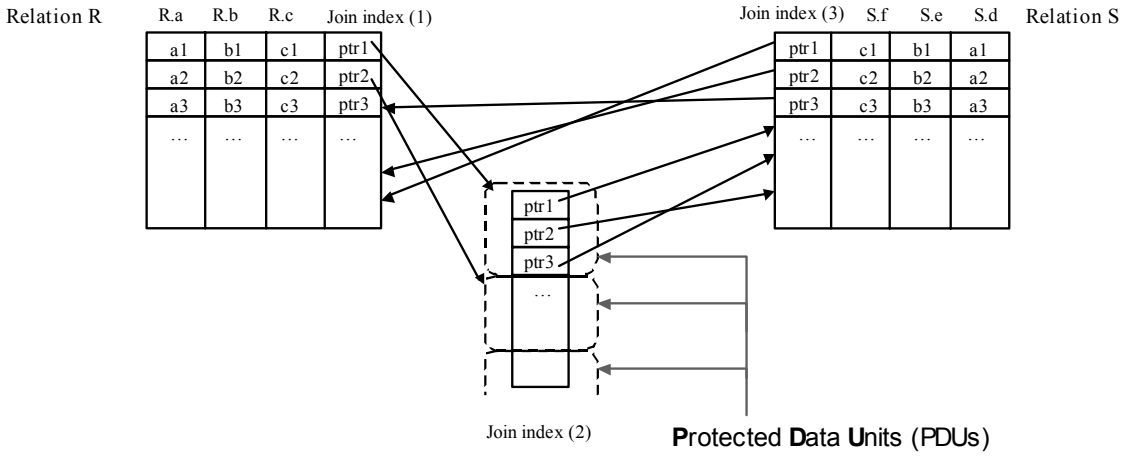



**Figure 60** *Storage Example for Join Indices.*

We should also examine some optimizations specific to our environment. For instance, as opposed to disk based DBMS, selection indices can be useful even with poor selectivity (e.g., 50%) and even if the index is non-clustered. Indeed, using the index may avoid repetitive PDU accesses to check the predicate since the test is performed on the index. For example, when evaluating the query *Select avg(salary) from client where function = "Assistant Professor"*, we can avoid accessing (i.e., loading, checking against tampering, and decrypting) all clients using an index since the qualified *function* value is accessed only once.

Storing the database values in domains can also be useful. Indeed, even if the compactness of the data in not an issue in terms of memory space, it could be helpful to accommodate with the constrained RAM of the SOE, particularly in the case of large and redundant values. With domains, the SOE could in a first step project the domain pointers. Then, before accessing the domain value, the SOE could ask the rendering terminal if it yet owns the domain value, thereby avoiding accessing it.

Finally, we must elaborate a cost model, optimization strategies and validate our choices by extensive performance evaluations on representative instantiations of the architecture mentioned in Figure 52.

# Chapter 6 – Conclusion and Research Perspectives

Pervasive computing and ambient intelligence motivate the development of new data-centric applications that must be tackled in a growing variety of ultra-light computing devices. Indeed, the need for embedded database functionalities in such environment comes from three motivations. First, the confidentiality of embedded or collected data forces query processing to be confined in secure smart objects which should only deliver the authorized subpart of the embedded data. Second, communication costs (and associated high energy consumption) leads to reduce transmitted data to its most pertinent subpart. Finally, disconnected activities require embedded data management techniques to perform computation locally without relying on remote resources.

Query execution techniques must therefore be adapted to small devices. In smart objects, RAM appears to be the most critical resource. In this document, we first propose efficient techniques to process database queries within highly RAM-constrained smart objects. Then, we concentrated on the smart card context and showed the effectiveness of ad-hoc embedded data storage and indexation models based on an extensive performance evaluation. Finally, we propose preliminary solutions to cope with smart card storage limitations, relying on remote, but insecure, data stores.

This chapter first summarizes the studies addressed in this document. Then, we present some attractive perspectives that we envision to tackle in the short or mid-term.

## 1 Summary

The first study presented in this document addresses the RAM limitation shared by most smart objects, proposing a framework that helps to design RAM-constrained query evaluators. We proposed a query execution model that reaches a lower bound in terms of RAM consumption. This model processes rather complex queries with a minimal RAM - few tens of bytes - required to organize the dataflow between operators and to store the state of aggregation or sort operators (*i.e.*, current aggregate computation or current sorting key). Minimal RAM consumption is trade for numerous iterations during query processing, thus limiting the efficiency of this execution model. Consequently, we devised a new form of optimization, called iteration filter, which drastically diminishes the prohibitive cost incurred by the preceding model, without additional RAM consumption. Thanks to iteration filters,

the dataflow between the operators of the QEP is shrunk by a factor between 5 and 100 depending on the query and on the operators' position in the tree. Iterations' filters thus reduce the query evaluation cost by one or two orders of magnitude. Finally, we proposed variations of the preceding techniques that best exploit any incremental growth of RAM. The additional RAM benefits to the query evaluation, and iteration filters still apply. Our performance evaluations show the accuracy of the proposed techniques and demonstrate that they constitute a convincing alternative to indexing techniques in a wide range of situations. Indeed, the response time shows a hyperbolic shape with the RAM growth, indicating that the operator's algorithms exploit well the additional RAM, especially for small RAM addition, which constitutes our sphere of interest. Typically, most of the queries considered in our measurements can be executed with a response time close to 1 second with only 1 kilobytes of RAM. Our performance evaluation helps finding the best compromise between RAM capacity, volume of on-board data, query complexity and response time. This performance evaluation provides guidelines that allow to calibrate the RAM resource of a hardware platform according to given application's requirements, as well as to adapt an application to an existing hardware platform.

In the second study, we considered applications dealing with confidential data, accessed by several users with different privileges (*e.g.*, medical folder, user profile). PicoDBMS [PBV+01] allows to embed confidential data in a smart card and to check the users' privileges by embedding a query engine as well. We go one step further than the PicoDBMS study: we categorize the required access rights in a pico-database contexts and evaluate the performance of different storage and indexation techniques to fulfill these requirements. To this end, we define three main categories of authorizations covering a wide range of situations: schema, occurrence and computed data authorizations. We then derive seven access rights types derived from these three categories that should be supported by a PicoDBMS-like system. Several queries, representative of the access right types are use in an extensive performance evaluation conducted on an advanced smart card prototype and on a hardware simulator. These evaluations show the effectiveness of each proposed data and index structures according to the access rights required by the application and the amount of involved data. The conducted analysis also helps selecting the appropriate storage structures for a given application according to the volume of embedded data, the required access rights and the required response time. For instance, considering an application with a total execution time requirement of 1 second, the domain storage (DS) will satisfy this requirement for 64 kilobytes smart cards but Ring Storage (RS) and Ring Inverse Storage (RIS) will be necessary for larger smart cards. Similar conclusions can be raised for any application requirements. Finally, the performance measurements can be used to determine the limit of such a "*totally embedded*" approach (*i.e.*, data and query engine are embedded) in terms of response time and amount of embedded data. For instance, when several megabytes of data need to be secured, the proposed query execution models cannot fulfill severe application requirements. In this case, other techniques have to be envisioned (see next section).

The third study addresses secure storage delegation. Given the smartcard tiny die size, the amount of stable storage will remain limited to few megabytes for a while. Indeed, highly dense stable memory technologies, like Millipedes, are not expected to be available in a short or mid term. Thus, we proposed to rely on remote storage resources to increase smart cards abilities. Such a proposal fits emerging smart card based architecture where the secure chip is connected to large (but insecure) memory modules. Remote storage media might also be available when smart cards are plugged into a more powerful device or connected to a network. Our goal is thus to store the data on an insecure store while keeping the same tamper resistance as provided by the PicoDBMS solution. First, we stated that, even if the remote resource provides powerful processing capabilities, the delegated processing should be reduced to its minimum for security reasons. The remote storage interface is thus reduced to a low level I/O API, accessing "files". Then, we showed how to set cryptographic techniques to make the remote "files" tamper-resistant. Finally, we explained how to organize and partition data and indices into these secured "files" to allow efficient query processing.

## 2 Research Perspectives

This section describes some interesting ideas that could not be tackled during the thesis and that we would like to study thoroughly.

*Contactless environments:* Although our study focuses on the contact smart card context, contactless interfaces are more and more pushed by constructors and governmental organizations [SIN02b] for its ease of use. While PicoDBMS could run "as is" on contactless smart cards, whether the applications' requirements remain the same is an important question. For example, the target in terms of response time induced by the interaction with human users (the limit of one second is often considered) may be reconsidered. This could conduct to revise the way queries are evaluated and may lead to more specific query engines. For instance, a query engine providing an approximate answer could be a solution to match severe response time requirements (*e.g.*, in order avoiding the card holder to stop when passing near to a card reader).

*Cache conscious execution strategies:* Another interesting research issue in smart card would be to improve the processor data cache usage when processing queries. Indeed, advanced smart cards are now endowed with a cache holding up to 1 kilobytes [Phi03]. Cache aware strategies had been envisioned on traditional and main-memory DBMS, which led to reorganize indices' storage [RaR99] and data layout [ADH+01, ADH02]. Recently, cryptographic algorithms requiring high throughput have been implemented in such a cache conscious manner [ABM04]. Cache conscious algorithms could bring great performance improvements in the RAM constrained context since RAM lower bound execution models lead to numerous iterations which may benefit from the cache. Moreover, cache conscious processing could lead to rethink data and index storage organization. The problems becomes

more complex when we consider the data compactness requirement which may contradict some cache conscious storage organizations.

*New stable memory technologies:* In this document, we focus on EEPROM technology because current smart cards are endowed with this technology. As stated in Chapter 2, Section 1.2.4, Flash memory will become more popular in future smart card platform. Flash memory has different properties that should be taken into account for the design of embedded database components. The challenging perspective of adapting database component to Flash memory has been studied in [BSS+03] considering only writes with a small amount of data (hundreds tuples). A study considering a large amount of data and oriented to complex access rights evaluation would be very valuable. In the same spirit, long term memory alternatives could also be envisioned, including technologies like PCM, OUM and Millipedes. These memories also provide very specific and interesting access properties which have a direct impact on the database storage and indexation model. There are already some works on Millipedes [YAE03] in a general context. Combining the memory properties with the other the hardware constraints will undoubtedly generate interesting problems.

*Securing large outsourced databases:* Chapter 5 proposes to rely on remote storage resources to increase smart cards abilities while considering PicoDBMS-like applications. An important assumption is that the database is always accessed using the same physical card. This means that (i) there is no concurrent access and (ii) sharing the data means sharing the smart card. Assuming the database is stored on an insecure server and removing the previous assumption leads to another spectrum of applications, *i.e.*, in the *Database Service Provider* Model [HIL+02], where users would be endowed with smart cards securing the access to the remote encrypted data. This hot topic already generated several papers including [HIL+02, BoP02, DDJ+03, IMM+04], however these papers focus only on the confidentiality of the remote data and not on tamper-resistance (including opacity, integrity and freshness). Extending our preliminary study of Chapter 5 could thus be a real challenge but raises some new issues. Indeed, when several users access the protected database, each one holding its own smart card, classical transaction management issues occur and are complicated by the fact that the data is encrypted and that freshness must be ensured.

*Remote processing:* The techniques proposed in Chapter 5 can also be viewed as a first step to cope with smart card constraints using remote (and thus insecure) resources. We took a somehow radical approach, reducing the remote processing to its minimum. However, it is an interesting perspective to further investigate remote (and secure) processing delegation. This could increase the query processing performance on a large amount of secure data, and could also be envisioned to save energy (since data is stored remotely). This raises some interesting issues. First, care should be taken to avoid information disclosure by using specific encryption scheme [BoP02] or by adding indices to the encrypted data [HIL+02]. This is a difficult problem since, as soon as the insecure device has the possibility to make some processing on the encrypted data, a potential pirate may exploit this possibility to extract unauthorized information. Second, the honesty of remote processing will have to be

checked since the external device is insecure. Some techniques indeed exist for simple selections [GKM+04] but the problem remains open for more complex queries. In that context, the security could also be balanced with efficiency and power consumption.

*XML data model:* All the studies made in this manuscript were done in the relational context. Considering an XML PicoDBMS is obviously an interesting direction. For instance, XML may be more appropriate for personal profile (or VHE - Virtual Home Environment). Considering the XML data model raises different concerns: (i) XML storage and indexation: while many proposals for storing and indexing XML data have been made, we are not aware of XML storage and indexation models for smart cards. (ii) XML access control: XML access rights [BCF01, GaB01, DDP+02] are based on XPATH and [BDP04] proposed an efficient streaming access rights evaluator embedded in a smart card. Checking access rights on local data changes significantly the problem; moreover this issue is highly related with the previous one. Finally, considering XML and new memory architectures may bring interesting contributions.

# Résumé en Français – French Summary

## 1 Introduction

L'évolution actuelle de l'informatique conduit à un environnement composé d'un grand nombre de petits dispositifs spécialisés, servant les individus. Mark Weiser, le pionnier de l'informatique ubiquitaire, dépeint cette vision du monde informatique au début des années 90 [Wei91], qu'il définit ainsi, à l'inverse de la réalité virtuelle : "la réalité virtuelle mets les gens au cœur d'un monde généré par l'ordinateur, alors que l'informatique ubiquitaire force l'ordinateur à vivre dans le monde réel des gens." Dans cette vision, les individus sont entourés d'une pléiade d'objets intelligents. Ces dispositifs communicants sont conscients des individus, à l'écoute de leurs besoins, et ajustent leurs actions en fonction.

Le concept d'objet intelligent apparaît dans les années 90, au carrefour de la miniaturisation des composants, des technologies de traitement de l'information et des communications sans fil. Dans cette thèse, nous voyons les objets intelligents comme des dispositifs dotés de capacités d'acquisition de données, de stockage, de traitement et de communication. Les cartes à puce (cartes Java multi-applications, cartes bancaires, cartes SIM, carte d'indenté électronique, etc.), les capteurs surveillant leur environnement (collectant des informations météorologiques, de composition de l'air, de trafic routier, etc.) et les dispositifs embarqués dans des appareils domestiques [BID+99, KOA+99, Wei96] (puce embarquée dans un décodeur de télévision) sont des exemples représentatifs des objets intelligents. Les chercheurs en médecine (projet Smartdust [KKP]) comptent aussi sur les puces pour être inhalées ou ingérées (pour remplacer les radiographies, automatiser la régulation du taux de sucre chez les diabétiques, etc.).

Dès lors que les objets intelligents acquièrent et traitent des données, des composants base de données embarqués peuvent être nécessaires. Par exemple, dans [BGS01], des réseaux de capteurs collectant des données environnementales sont comparés à des bases de données distribuées, chaque capteur agissant comme un micro-serveur de données répondant à des requêtes. [MaF02] souligne le besoin d'effectuer des traitements locaux sur les données accumulées, telles que des calculs d'agrégats, pour réduire la quantité de données à émettre et ainsi économiser de l'énergie. De plus, la protection de données contextuelles ou de dossiers portables (comme un dossier médical, un carnet d'adresse, ou un agenda) stockés dans des objets intelligents conduit à embarquer des moteurs d'exécution de requêtes sophistiqués

pour éviter de divulguer des informations sensibles [PBV+01]. Ces exemples sont représentatifs du besoin grandissant de composants base de données embarqués.

Cette thèse adresse particulièrement l'impact des contraintes relatives aux objets intelligents sur les techniques base de données à embarquer dans les calculateurs. L'introduction est organisée comme suit. La première section de l'introduction met en évidence ces contraintes et motive le besoin de composants base de données embarqués. La section suivante expose les trois contributions majeures de la thèse et la dernière section présente l'organisation de ce résumé.

## 1.1 Contexte de l'Etude

Les objets intelligents disséminés disposent de ressources fortement contraintes, d'une part pour des raisons techniques, et d'autre part pour faire face à des réalités économiques.

En effet, de nombreux domaines applicatifs contraignent la taille du dispositif pour lui permettre d'être embarqué (dans les appareil domestiques), disséminés dans l'environnement (capteurs), transportés (cartes à puce), ingérés (usage médical), etc. La taille réduite du dispositif impose la réduction des ressources embarquées, notamment en terme de mémoires persistante et volatile (RAM) disponibles. De plus, les objets intelligents sont autonomes dans de nombreux cas, ce qui pousse à limiter les ressources embarquées, notamment en terme de puissance du processeur et de bande passante de communication, pour une question de durée de vie (sauvegarde d'énergie). Les aspects sécuritaires exacerbent toutes ces considérations, la taille du dispositif étant d'autant plus réduite (par exemple dans les cartes à puce) pour compliquer l'observation de l'activité des composants (écouter le bus de données) et rendre les attaques très coûteuses. Enfin, les considérations économiques conduisent à réduire le coût de production au minimum pour les dispositifs dédiés aux marchés de masse. Pour un processus de fabrication donné (par exemple, avec la technologie 0.18 microns), le coût du dispositif est directement lié à sa taille, ce qui conduit à calibrer soigneusement les ressources.

La conception de composants base de données embarqués est un enjeu majeur [ABP03c] pour les raisons suivantes.

Les objets intelligents peuvent amasser de l'information relative à des individus, ce qui entraîne des problèmes de sécurité. Dès lors que les données accumulées sont sensibles, le moteur d'exécution de requêtes doit rester confiné dans le dispositif intelligent pour construire et délivrer le sous-ensemble autorisé des données embarquées. Ainsi, la complexité du moteur embarqué dépend de la sophistication des droits d'accès à pourvoir. Dans les environnements futurs [BID+99, KOA+99], de nombreux programmes et personnes physiques pourraient interagir avec les objets disséminés, chacun selon ses propres droits d'accès.

La réduction des coûts de communication constitue une autre motivation des composants base de données embarqués. En effet, dans un contexte sans contact, l'économie d'énergie est liée directement à la quantité de données transmises. Alors, les composants base de données de filtration et d'agrégation, et plus généralement les composants permettant de réduire la quantité de données transmises sont un partenaire indispensables des dispositifs légers. Dès lors que la réduction des communications permet d'économiser de l'énergie [MFH+03], cette motivation subsistera à long terme.

Enfin, la gestion de données embarquées est nécessaire lorsque les traitements doivent être effectués de façon déconnectée. Par exemple, les objets intelligents embarqués dans le corps humain, pour mesurer les pulsations cardiaques et délivrer rapidement une prescription médicamenteuse en cas d'activité suspecte, doivent analyser les données localement sans dépendre de ressources distantes.

En conséquence, la confidentialité, les communications et l'économie d'énergie, ainsi que les activités autonomes nécessitent d'embarquer des composants base de données dans les dispositifs intelligents. La conception de ces composants est donc un problème important, qui se trouve compliqué par le fait que chaque dispositif présente des spécificités propres (consommation d'énergie réduite, résistance aux attaques) et des contraintes matérielles spécifiques (dé de silicium limité en volume, asymétrie des ressources).

## 1.2 Contributions de la Thèse

Cette thèse apporte les trois contributions suivantes :

Comme dans tout dispositif ultra-léger, la mémoire volatile (RAM) est une ressource cruciale. Dans le contexte d'étude de la thèse, les techniques classiques d'évaluation de requêtes, basées sur l'exploitation de grandes quantités de mémoire RAM, sont inapplicables. Des techniques nouvelles, qui prennent en compte la limitation forte de la mémoire RAM, sont alors indispensables. Le chapitre 3 de la thèse donne une solution à ce problème. L'étude du traitement de requêtes présentée s'applique à tout environnement disposant d'une mémoire RAM limitée. La solution est à base de réitérations successives sur les données de base et utilise des filtres d'itération. Un prototype basé sur ces techniques a été réalisé. L'évaluation de performances montre la pertinence des techniques proposées et conduit à exprimer des règles de co-conception pour trouver le compromis adéquat entre capacité RAM, complexité des requêtes (ou des droits d'accès supportés) et temps de réponse. Il s'agit d'une contribution majeure de la thèse publiée dans [ABP03].

La seconde contribution présentée dans la thèse répond au problème de confidentialité des données embarquées dans une carte à puce. En effet, la sécurité est un avantage essentiel de la carte à puce mais l'approche triviale pour gérer des droits d'accès sur une carte (i.e., une vue matérialisée par utilisateur) conduit à une réplication massive des données et à des droits figés (très peu dynamiques). Pour éviter ces désavantages majeurs, il faut embarquer

des composants permettant de calculer dynamiquement la vue autorisée des données stockées. Le chapitre 4 de la thèse donne une solution à ce problème en étendant le travail initial sur la conception du PicoDBMS [PBV+01]. Une analyse complète de la gestion de données embarquées sur une carte à puce est conduite, généralisant ainsi l'approche PicoDBMS. Les différents droits d'accès qui peuvent être définis sur n'importe quelle donnée embarquée sont classifiés, et différentes techniques de stockage et d'indexation sont envisagées. Un prototype complet du PicoDBMS a été réalisé, fournissant ainsi un contexte d'expérimentation réaliste. L'évaluation de performances permet de conclure sur le choix d'une structure de données pour une application selon le volume de données embarquées, les droits d'accès et les temps de réponse requis.

La troisième contribution adresse la limitation de mémoire de stockage inhérente à la carte à puce, qui constitue un problème à long terme. Cette entrave impose, pour certaines applications dont les données ne peuvent être entièrement embarquées, de recourir à un système externe, typiquement moins sûr. Les techniques de gestion de données doivent alors pouvoir résister à la falsification des données et des traitements délégués à l'extérieur pour assurer une sécurité aussi proche que possible de celle offerte par la technologie PicoDBMS. Le chapitre 5 de la thèse donne une solution à ce problème en proposant d'utiliser un système de stockage externe non sûr  tout en garantissant les mêmes propriétés de résistance aux attaques que la carte à puce. Ce chapitre présente les menaces supplémentaires induites par l'utilisation d'un système de stockage externe, ainsi que l'ébauche d'une solution. La solution combine des techniques de traitement de requêtes et de cryptographie, et met en œuvre une stratégie d'exécution compatible avec la faible quantité de RAM disponible dans la puce.

## 1.3 Organisation du Résumé

Dans ce résumé, nous détaillons la seconde contribution, puis décrivons plus brièvement la première et la troisième. Ainsi, la partie la plus importante de résumé est dédiée à l'étude des puces électroniques et notamment à la technologie PicoDBMS.

D'abord, nous donnons une vision de l'évolution profonde des technologies et des usages de la puce sur les dernières années. Nous présentons ainsi des perspectives nouvelles et une motivation plus forte aux bases de données embarquées dans des puces sécurisées. De plus, cette première partie nous permet d'actualiser et de redéfinir le problème des SGBD embarqués en conséquence.

Ensuite, nous présentons le design de PicoDBMS, en terme de modèle de stockage et d'accès aux données stockées dans la puce, puis en terme d'exécution de requêtes.

Puis, nous validons PicoDBMS du point de vue des performances et tirons un certain nombre d'enseignements sur son design initial. Pour conduire cette évaluation de performances, nous catégorisons les droits d'accès nécessaires à définir sur les données

embarquées, nous définissons une métrique propre au contexte embarqué, puis nous mesurons précisément les performances des différents aspects de l'évaluation de requêtes (stockage, indexation, opérateurs, et plans d'exécution) permettant le support des droits d'accès définis précédemment.

Enfin, nous traçons de nouvelles orientations de recherche autours des bases de données embarquées dans les puces. Nous montrons ainsi la pertinence et la pérennité de cette problématique, en terme de capacité à générer de nouveaux problèmes de recherche. C'est dans cette dernière partie que nous résumons aussi les contributions de la thèse présentées dans les chapitres 3 et 5.

# 2 Bases de Données Sécurisées dans une Puce

## 2.1 Evolution des Technologies et des Usages

La puce sécurisée est à la croisée des chemins de deux chantiers majeurs de l'informatique moderne : (i) la mise en œuvre de systèmes disséminés intelligents (réseaux de capteurs, intelligence ambiante, etc.), et (ii) la sécurisation des systèmes, allant des simples calculateurs (PDA, téléphones) aux systèmes d'information d'entreprise [Can04, TCG04].

Ces dernières années, un nouveau pan de recherche est apparut, consacré à l'étude des systèmes intelligents disséminés. L'apparition de cette problématique dans les conférences majeures du domaine et le nombre grandissant de nouvelles conférences dédiées à cette problématique attestent de sa pertinence. Ce domaine va des réseaux de capteurs aux dispositifs d'intelligence ambiante basée sur des profiles utilisateurs et réagissant à l'activité alentour [FAJ04]. Les composants impliqués dans un tel environnement sont très proches des cartes à puce. De plus, ils acquièrent des données (environnementales, évènementielles), ont des capacités de traitement et communiquent entre eux (avec éventuellement des politiques de droits). Ceci ouvre à de nouvelles applications base de données embarquées, renforçant l'intérêt de l'étude de PicoDBMS.

En outre, les systèmes informatiques évoluent vers des solutions sécurisées. Des projets comme *Palladium* [Mic02] ou *Trusting Computing Platform* [TCP04, IBM] adressent spécifiquement la sécurité de plates-formes informatiques personnelles à l'aide de matériel sécurisé et de programmes de confiance (basés sur du chiffrement, des signatures digitales et des droits digitaux). Ces efforts initiaux ont fusionné, formant un large projet conduit par le *Trusting Computing Group* (TCG) [TCG04], impliquant les leaders mondiaux de solutions matérielles et logicielles tels que Intel, IBM, HP, Microsoft, AMD, Sony, etc. L'objectif est de sécuriser des réseaux et applications [Did04] composées de simples calculateurs comme les assistants personnels, les téléphones cellulaires et les PC [Mic04], et allant jusqu'aux systèmes d'information d'entreprise [Can04, TCG04]. Simultanément, des organisations

gouvernementales [FTC02] étudient la faisabilité des systèmes sécurisés en terme de liberté individuelle, et certains mouvements réagissent à cette forme de contrôle de sécurité imposée par les constructeurs [NoT04]. Cet intérêt global pour la sécurité conduit les constructeurs à offrir de plus en plus de supports informatiques sécurisés. Les ibuttons [Ibut00, HWH01], cartes mémoires sécurisées (type MMC sécurisée), composants sécurisés reliés aux ports série/parallèle/USB (clé sécurisée), *Crypto-Tockens* et bagues magnétiques (comme la *JavaRing*) sont des exemples représentatif de cette offre nouvelle. La plupart de ces composants sécurisés sont proche des cartes à puce vu qu'ils disposent de microcontrôleurs 32-bit garantissant la sécurité, dotés de mémoire électronique interne et de RAM limitée, se différenciant simplement des puces dans la façon de s'interfacer avec leur hôte [VRK03]. Ainsi, la plupart des constructeurs de solutions carte à puce suivent avec intérêt cette évolution vers la sécurité vu son potentiel à générer de nouvelles applications [Yos03] et des bénéfices financiers importants (comme le contrôle du partage ou de la copie de données multimédias, ou la limitation du vol de données d'entreprise). En effet, la sécurité matérielle inhérente aux cartes à puce et leur puissance de calcul en font un maillon de sécurité idéal [DhF01]. De plus, à plus grande échelle (sécurité d'un PDA, PC, réseaux…), dès lors que la sécurité est gérée par une puce, elle n'est plus contrôlée par les programmes ou le système d'exploitation, comme dans la proposition de *ST Microelectronics* à base de puce *ST19* compatible avec les spécifications du TCG [STM04]. La politique de sécurité est ainsi paramétrée par le porteur de la carte, et non par les constructeurs, faisant de la carte à puce la solution de sécurité admissible par tous et aisément contrôlable par l'utilisateur.

## 2.2 Exemples d'Applications

Bien sur, les classes d'applications identifiées dans [PBV+01] ont été renforcées par l'intérêt grandissant des organisations gouvernementales pour la sécurité. Par exemple, la loi Gramm-Leach-Bliley (GLB Act) régissant la confidentialité des données personnelles accumulées par les prestataires de produits financiers (banques, compagnies d'assurance) ou les spécifications HIPAA [OCR02] en vigueur depuis 2003 aux USA, régissant les principes de sécurité protégeant toute information personnelle (données médicales, d'assurance, etc.), rendent les cartes à puce de plus en plus attractives pour embarquer les données médicales des citoyens [Sma03a, Sma03c], prendre part à des transactions financiers sécurisées [SPE03, Sma03b], et servir de support aux données d'identification (voir [Acc02], les initiatives électroniques des gouvernements). De plus, l'évolution décrite précédemment, poussant à plus de sécurité, ainsi que l'apparition de nombreux composants intelligents disséminés, ouvre de nouveaux champs d'applications des puces. A la lumière de ces considérations, nous pouvons établire les classes d'applications principales suivantes, les 4 premières ayant déjà été identifiées dans [PBV+01] :

- *Identification et outil transactionnels déconnectés* : applications correspondant au marché traditionnel des cartes à puce (carte de crédit, porte-monnaie électronique, carte SIM), et se trouve renforcé par l'apparition dans de nombreux pays de cartes d'identité

électronique comme les cartes nationales d'identité, de transport, ou d'accès à des zones physiques sécurisée (personnel d'aéroport). Le volume de données embarqué est faible (identifiant du porteur et informations précisant son statut), et l'évaluation de requêtes et droits d'accès sont inutiles, un PIN-code suffisant à assurer la confidentialité du contenu.

- *Bases de données téléchargeables :* consistent en des packages de données (liste de restaurants, hôtels et cites touristiques, catalogues, etc.) téléchargés dans la carte et accédés via n'importe quel terminal. Le volume de données peut être important et les requêtes complexes, mais les droits d'accès sont inutiles (1 seul utilisateur, données en général publiques et en lecture seule).

- *Environnement utilisateur :* stocke dans la carte le profile du porteur [Pot02], constitué de données propres au porteur (des informations bio-métriques [She03]), à son environnement (données de configuration, mots de passe, cookies, licences de logiciels, etc.), et des données personnelles (carnet d'adresse, agenda, signets, etc.). Les requêtes restent simples, vu que les données ne sont pas reliées. Cependant, elles doivent être protégées par des droits d'accès sophistiqués, car la puce est accédée par différents utilisateurs (programmes ou humains) avec leurs propres droits.

- *Dossiers personnels :* peuvent être de nature médicale, scolaire, d'assurance, et maintiennent un historique d'évènements (prescription médicales, cotisations d'assurance, etc.). Ces données sont hautement confidentielles et peuvent être accédées par plusieurs utilisateurs (médecin, pharmacien, ou organismes gouvernementaux pouvant interroger des populations de cartes et des dossiers de type différent pour chercher par exemple les liens possibles entre certaines maladies et le niveau d'éducation du porteur). Les droits d'accès nécessitent d'être définis sur des vues complexes pour assurer la confidentialité des dossiers.

- *Surveillance environnementale, autonome et sécurisée :* les réseaux domestiques, de capteurs, et le contexte de l'intelligence ambiante impliquent de nombreux composants disséminés qui acquièrent des données de façon autonome. De manière analogue, le standard *MasterCard Open Data Storage* (MODS) [Mas03] permet aux vendeurs, aux banques et autres organisations de stocker et accéder des informations sur les cartes de crédits. Les données accumulées peuvent être confidentielles, dès lors qu'elles concernent l'historique des évènements reliés à l'espace de vie d'un individu. Pour assurer la confidentialité des données, le propriétaire des données accumulées doit attribuer des droits éventuellement complexes à chaque utilisateur potentiel de son environnement (autres composant autonome, organisations, applications fabriquant un profile, etc.). Ce domaine d'application naissant et les exemples de scénarios d'utilisations possibles sont décrits dans [FAJ04].

- *Sécurisation de flots et de supports de stockage :* les informations payantes (régies par

des droits digitaux) distribuées en flots à l'ensemble des utilisateurs [vldb04], les données confidentielles stockées sur un support non sécurisé, les informations sensibles circulant sur un réseau peuvent être protégées par des composant de confiance (puces sécurisées, cartes cryptographiques, co-processeurs sécurisés, etc.). Cette classe d'applications correspond à de l'embarquement de traitement dans un composant sûr pour protéger des données externes. Dès lors que des puces sécurisées sont attribuées à de nombreux utilisateurs (carte nationale d'identité électronique, cartes SIM, etc.) comme cela est envisagé dans de nombreux projets (projets de cartes multi-usages [MTS+04, SIN02b] utilisées comme passeport, permis de conduire, carte d'électeur, de transport, etc. lancés dans de nombreux pays d'Europe, aux Etats-Unis [USG03, OHT03, Kim04], au Japon [Dep01] et en Chine [APE03]), ce type d'applications nécessitant le contrôle de l'identité vont bénéficier des technologies carte [All95, Per02a]. Les droits d'accès sur les données externes peuvent être sophistiqués, la puissance de calcul requise et le débit de la puce peuvent être fortement sollicités (les cartes cryptographiques chiffrent des messages électroniques à la volée).

| *Applications* | | *Volume de données* | *Sélection Projection* | *Jointure* | *Groupby, Distinct* | *Droits d'accès, vues* | *Atomicité* | *Durabilité* | *Statistiques* |
|---|---|---|---|---|---|---|---|---|---|
| *Identification et outil transactionnel* | | Faible | | | | | √ | | |
| *Orientées données* | *Base téléchargeable* | Large | √ | √ | √ | | | | |
| | *Environnement utilisateur* | Moyen | √ | | | √ | √ | √ | |
| | *Dossier personnel* | Large | √ | √ | √ | √ | √ | √ | √ |
| | *Surveillance intelligente* | Large | √ | √ | √ | √ | √ | √ | √ |
| | *Sécurisation de données externe* | Large | √ | √ | √ | √ | √ | √ | √ |

**Table 15** *Profiles des applications des cartes à puce.*

La Table 15 résume les caractéristiques des composants base de données embarqués dans les puces pour répondre à ces différentes applications. On peut noter que les cartes multi-usages peuvent abriter plusieurs classes d'applications, poussant ainsi à embarquer par défaut un moteur de SGBD puissant.

## 2.3 Formulation du Problème

Dans le contexte de PicoDBMS, les données confidentielles sont confinées dans la puce et bénéficient de son niveau (très élevé) de résistance aux attaques. Toutefois, bien que les propriétés matérielles de la puce protége les données des attaques physiques, cet environnement force aussi à embarquer un code spécifique permettant de n'externaliser à l'utilisateur que des données autorisées. Ce code peut être vu comme le gardien authentifiant

l'utilisateur et ne délivrant que les données correspondant à ses droits d'accès.

Dans la suite, nous considérons que l'identification/authentification de l'utilisateur sont assurés de façon classique (nom/mot de passe), de même que l'attribution des permissions (droits sur des vues de la base de données). Nous devons maintenant clarifier les droits d'accès à supporter par les applications de type PicoDBMS. Pour cela, nous définissons trois classes principales d'autorisations, de complexité croissante, couvrant un large panel de situations. Par soucis de généralité, ces catégories d'autorisation sont exprimées sur un schéma entité associations: (i) les *Autorisations sur le Schéma* (SA) sont définies sur l'intention de la base (son schéma) sans considérer l'extension de la base (occurrences); (ii) les *Autorisations sur les Occurrences* (OA) donnent accès à certaines occurrences, sur prédicat(s) sur leurs propriétés ou sur existence d'une association (directe ou transitive) à une autre occurrence autorisée; et (iii) les *Autorisations sur des données Calculées* (CA) donnent droit à des valeurs calculées sans permettre l'accès aux occurrences participant au calcul (par exemple, des valeurs moyennes sont révélées mais pas les valeurs individuelles utilisées dans le calcul).

Dans une base relationnelle, et prenant en compte les besoins des applications visées par PicoDBMS, nous distinguons sept types de droits d'accès dérivés des classes principales SA, OA et CA. Ils sont détaillés dans la Table 9. Le SA peut être implémenté en utilisant les instructions SQL classiques grant/revoke sur les relations de la base et des vues basées sur l'opérateur de projection (accès P dans la Table 9). Le prédicat du OA peut s'appliquer sur des attributs d'une relation (accès SP), de deux relations (association directe, accès SPJ), ou de plusieurs relation (association transitive, accès $SPJ^n$). Enfin, le CA est implémenté en SQL en utilisant des vues basées sur des calculs d'agrégats. L'agrégation peut être mono-attribut (groupement sur un seul attribut) et engager une seule relation (accès SPG), mono-attribut et engager plusieurs relations (accès SPJG), ou multi-attributs engageant potentiellement plusieurs relations (accès $SPJG^n$).

Une solution triviale permettant d'assurer les droits d'accès décrits précédemment serait d'embarquer un système de gestion de fichier. Chaque vue serait matérialisée dans un fichier, et des droits sur le fichier seraient attribués aux utilisateurs. Cette solution serait basée sur un code très simple. Cependant, la dynamicité des données de base et des droits d'accès serait entravée, et la solution entraînerait inévitablement beaucoup de redondance entre les fichiers, ce qui n'est pas concevable dans un environnement disposant de capacité de stockage restreinte. Nous considérons donc que l'évaluation dynamique des vues doit être considérée comme un pré-requis du problème, ce qui suppose d'embarquer des opérateurs base de données et la possibilité d'évaluer des requêtes.

| *Acro-nyme* | *Autorisa°* | *Type de droit d'accès* | *Opérateurs nécessaires* | *Exemple de vue* |
|---|---|---|---|---|
| P | SA | Sous-ensemble des attributs | Projection | Noms des docteurs |
| SP | OA | Prédicats sur une relation | Select, Project | Nom des docteurs pédiatres |
| SPJ | OA | Prédicats sur deux relations (association directe) | Select, Project, Jointure | Nom des docteurs vus le mois dernier |
| SPJ$^n$ | OA | Prédicats sur plusieurs relations (association transitive) | Select, Project, Jointure | Nom des docteurs ayant prescrit des antibiotiques |
| SPG | CA | Une relation, agrégat mono-attribut | Select, Project, GroupBy (agrégat) | Nombre de docteurs par spécialité |
| SPJG | CA | Plusieurs relations, agrégat mono-attribut | Select, Project, Jointure, GroupBy (agrégat) | Nombre de visites par spécialité de docteur |
| SPJG$^n$ | CA | Agrégat multi-attributs | Select, Project, Join, GroupBy (agrégat) | Nombre de visites par spécialité de docteur et par an |

**Table 16** *Description des Droits d'Accès sur les Données Embarquées.*

Le code embarqué dans la puce doit satisfaire les règles de conception suivantes, déduites des propriétés de la carte à puce:

- *Règle de compacité*: minimiser la taille des structures de données et du code du PicoDBMS pour s'adapter à la quantité réduite de mémoire persistante;
- *Règle de la RAM*: minimiser l'utilisation de la RAM vu sa taille extrêmement limitée;
- *Règle d'écriture*: minimiser les écritures en mémoire persistante vu leur coût très élevé (≅10 ms/mot);
- *Règle de lecture*: tirer parti des lectures rapides (≅100 ns/mot);
- *Règle d'accès*: tirer parti de la fine granularité et de l'accès direct à la mémoire persistante pour les opérations de lectures et les écritures;
- *Règle de sécurité*: ne jamais externaliser de donnée sensible hors de la puce et minimiser la complexité algorithmique du code embarqué pour éviter les trous de sécurité;
- *Règle du CPU*: tirer parti de la puissance surdimensionnée du CPU, comparée au volume de données embarquées.

Le problème est donc de soumettre à ces règles la conception d'un PicoDBMS embarqué capable d'évaluer dynamiquement, sur les données embarquées, les sept types d'expression de vue listés dans la Table 9.

Dans la suite, nous adressons les différents aspects de l'évaluation de vue dans une puce: le stockage, l'indexation, les opérateurs et le plan d'exécution. Nous ne considérons pas les autres aspects, sauf lorsqu'ils interfèrent avec le processus d'évaluation (comme le support transactionnel), auquel cas nous donnons le minimum d'informations nécessaire à la compréhension. Remarquons aussi qu'un utilisateur peut poser une requête portant sur

plusieurs vues. Dans ce cas, l'évaluation de la requête et la composition des vues peut être réalisée hors carte (sur le terminal) sans nuire à la règle de sécurité.

## 2.4 Travaux Existant

Dès les années 90, le besoin de gestion de données apparut dans plusieurs applications tournant sur des calculateurs légers, allant des contrôleurs industriels et des décodeurs de télévision câblée aux systèmes d'imagerie médicale et aux contrôleurs de vol des avions. Cela fut pour les fabricants de solutions embarquées une motivation suffisante pour s'intéresser au développement de SGBD embarqués. L'objectif est alors d'offrir un accès efficace aux données [SKW92, Hey94], des fonctionnalités transactionnelles (ACID), tout en cohabitant avec d'autres applications internes au calculateur (notamment celle utilisant la base de données). La conception du SGBD embarqué est alors guidée par la minimisation de l'empreinte du code (obtenu par simplification et componentisation), la portabilité sur différents calculateurs / systèmes d'exploitation, et l'auto administration [ChW00]. Généralement, le SGBD embarqué est dynamiquement lié à l'application embarquée. Quelques études académiques ont été menées dans ce domaine [BaT95, Ols00, Ort00], et de nombreux produits commerciaux existent comme *Pervasive SQL 2000* [Per02b], *Empress Database* [Emp00, Emp03], et *Berkeley DB* [SeO99, OBS99].

Parallèlement, l'augmentation rapide du nombre de téléphones cellulaires, assistants personnels (PDA), et d'autres calculateurs portables de grande consommation, pousse les principaux éditeurs de SGBD à proposer des versions légères de leur produit sur ce marché prometteur [Ses99, Sel99]. Ainsi, *Sybase SQL Anywhere Studio* offre une option de déploiement ultra-léger [Syb99, Syb00] de son produit phare *Adaptive Server Anywhere*. De même, des versions légères des SGBD principaux comme *IBM DB2 Everyplace* [IBM99, KLL+01], *Oracle Lite* [Ora02a, Ora02b] et *Microsoft SQL Server for Windows CE* [Ses00] ont été développées. Les éditeurs de SGBD porte une attention particulière à la réplication de données et à la synchronisation avec une base centrale, pour permettre l'exécution en mode déconnecté et la synchronisation lors de la re-connexion. A nouveau, l'empreinte du code est réduite par simplification du noyau du SGBD et en liant le code du SGBD de façon sélective à l'application. Les SGBD légers sont portables sur une grande variété de calculateurs et de systèmes d'exploitation, comme *PalmOs*, *WinCE*, *QNX Neutrino* et *Embedded Linux*. Bien qu'adressant des marchés différents, il est maintenant difficile de distinguer clairement les SGBD embarqués des SGBD légers (*Sybase Anywhere* semble même appartenir aux deux classes) vu que les uns ajoutent peu à peu des fonctionnalités base de données (support transactionnel, interrogation SQL, etc.) pendant que les autres retirent les fonctions non indispensables pour minimiser l'empreinte du code.

Le fait de considérer les puces comme un calculateur traditionnel pourrait conduire à utiliser un SGBD embarqué ou léger dans cet environnement. Cependant, les puces ont des composants matériels très spécifiques (compatibles avec un haut niveau de sécurité), qui

présentent des asymétries originales conduisant à repenser entièrement les techniques utilisées dans les SGBD embarqués/légers.

Les réseaux de capteurs amassant des informations météorologiques ou de trafic routier ont motivé plusieurs études récentes liées à la gestion de données dans les puces. *Directed Diffusion* [IGE00], *TinyDB* [MHH04], et *Cougar* [YaG02] concernent l'acquisition et le traitement de flots continus de données collectées. Des techniques d'optimisation de requêtes distribuées sur les capteurs sont étudiées dans [BGS01, BoS00]. La réduction de la consommation d'énergie liée aux communications par ondes radios, l'une des contraintes principales de cet environnement, est proposée dans [MaH02]. Bien que les capteurs et les cartes à puce partagent certaines contraintes matérielles, les objectifs des applications sont différents de même que les techniques utilisées pour atteindre ces objectifs. Dans un capteur, les fonctionnalités base de données nécessaires sont réduites au filtrage de données et à l'agrégation pour réduire le flot de sortie et ainsi économiser de l'énergie [MFH+02]. L'exécution des requêtes plus complexes mentionnées Table 9 est inutile. De plus, les contraintes sur le temps de réponse sont différentes dans les deux cas: selon l'utilisation du réseau de capteurs, les contraintes de temps peuvent être inexistantes ou temps réel alors qu'elles sont liées à l'interaction avec l'utilisateur dans le cas des cartes à puce. Enfin, les capteurs exécutent souvent des requêtes sur des flots de données, ou sur une seule table embarquée, alors que les cartes gèrent plusieurs relations.

Un nombre réduit d'études adressent spécifiquement la gestion des données dans les cartes à puce. Les premières propositions de SGBD pour carte à puce furent ISOL's SQLJava Machine [Car99] et le standard ISO SCQL [IOSp7] spécifiant un langage base de données pour carte à puce. Toutes deux adressent une génération de cartes disposant de 8 kilobytes de mémoire persistante. Bien que leur conception fut limitée à des requêtes mono-tables, ces propositions illustrent l'intérêt de concevoir un SGBD dédié pour carte à puce. Plus récemment, l'initiative de Mastercard, *MasterCard Open Data Storage* (MODS) [Mas03], offre une API commune permettant aux commerçants, banques et autres organisations d'accéder et de stocker des données sur les cartes à puces des utilisateurs avec une sécurité accrue pour le porteur de carte. Cependant, MODS est basé sur des fichiers plats, des droits d'accès grossiers (au niveau du fichier), et donc propose un modèle non compact et n'offre aucune possibilité d'exécution de requête. Néanmoins, cette initiative montre l'intérêt du développement de techniques base de données pour ces environnements.

Une étude récente propose des techniques de stockage spécifiques pour gérer des données dans une puce contenant de la mémoire flash. Là encore, la conception se limite à des requêtes mono-tables et est fortement orientée par l'architecture du type de puce ciblé. Ainsi, ce travail propose de stocker les données dans de la mémoire flash de type NOR, généralement substituée à la ROM et dédiée au stockage des programmes. Vu que les modifications dans une flash NOR sont très coûteuses (une simple modification impose l'effacement très coûteux d'un large block de données), les techniques envisagées sont orientée vers la minimisation du coût de modification (basée sur le marquage de tuple

effacés avec un *deleted bit* et de la pré allocation de tuple *dummy records* [BSS+03]). Bien que cette étude montre l'impact des propriétés matérielles sur le noyau du SGBD, elle ne satisfait pas les contraintes des cartes et ne traite pas l'exécution des requêtes complexes, nécessaires à la construction de la partie autorisée des données.

Enfin, remarquons que *GnatDB* [Vin02], présentée comme un système de gestion de base de données ultra-léger et sécurisé peut tenir dans une puce (son empreinte est de 11 kilobytes). En effet, GnatDB est conçu pour gérer une petite quantité de données externes, la puce étant utilisée pour protéger ces données contre les altérations accidentelles ou malicieuses. GnatDB n'adresse pas l'exécution de requête ni le stockage interne et l'indexation.

# 3 Le SGBD Embarqué PicoDBMS

PicoDBMS est doté des composants suivants, stockés dans la puce : un module de stockage organisant les données de la base et les index associés dans la mémoire persistante de la puce, un évaluateur de requêtes capable de construire et d'exécuter des plans d'exécution (composées d'opérateurs de sélection, projection, jointure et calcul d'agrégat), un moteur transactionnel assurant l'atomicité de séquences de modifications, et un module de droits d'accès offrant la possibilité de donner/retirer des droits sur les vues définies, évitant ainsi d'externaliser des données non autorisées. D'autres modules sont placés sur le terminal, dans le driver JDBC interfacant l'application utilisateur avec la base de donnée embarquée. Le schéma de la base autorisée à l'utilisateur est externalisé à la connexion, permettant au composant JDBC de parser la requête et d'en vérifier la syntaxe. La clause order by est évaluée si nécessaire sur le terminal. On peut noter que ces traitements externes satisfont à la règle de sécurité vu qu'ils ne concernent que des données autorisées. Ainsi, PicoDBMS permet le partage de données confidentielles entre plusieurs utilisateurs se connectant alternativement à la base. La Figure 61 illustre cette architecture.

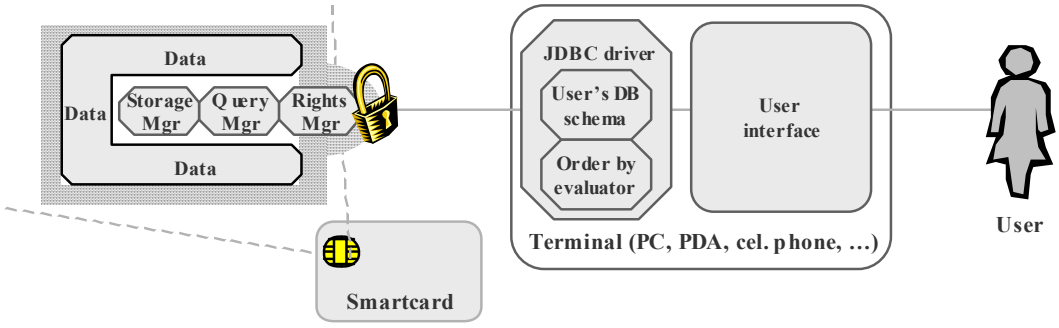



**Figure 61** *Architecture de PicoDBMS.*

Dans la suite, nous présentons les concepts strictement nécessaires à la compréhension du processus d'évaluation des données autorisées, à savoir le modèle de stockage et

d'indexation, les opérateurs de l'algèbre relationnelle et l'évaluation des plans d'exécution. En outre, nous décrivons brièvement les techniques transactionnelles (bien qu'elles soient hors du sujet) dans la mesure où elles interviennent dans les mesures de performance et dans la taille du code. Ainsi, les aspects transactionnels doivent être pris en compte dans l'évaluation des performances globales et la faisabilité de PicoDBMS. Pour une description plus détaillée de ces aspects, nous revoyons le lecteur à [PBV+01].

## 3.1 Modèle de Stockage et d'Indexation

La façon la plus simple d'organiser les données est le *stockage à plat* (FS), dans des enregistrements placés séquentiellement sur le support de stockage, contenant les valeurs de chacun des attributs qui les composent. Le principal avantage de cette organisation est la bonne localité des données. Dans notre contexte, cette alternative présente cependant deux inconvénients majeurs. D'une part, elle est très consommatrice en espace de stockage, n'éliminant pas les doublons de valeurs. D'autre part, elle implique une exécution inefficace basée sur des opérations séquentielles, convenable uniquement pour les premières générations de puces disposant de quelques kilobytes de données. L'ajout de structures accélératrices peut résoudre ce problème d'exécution, mais aggrave encore la consommation mémoire. Ce type d'organisation est donc inapproprié pour un PicoDBMS[43].

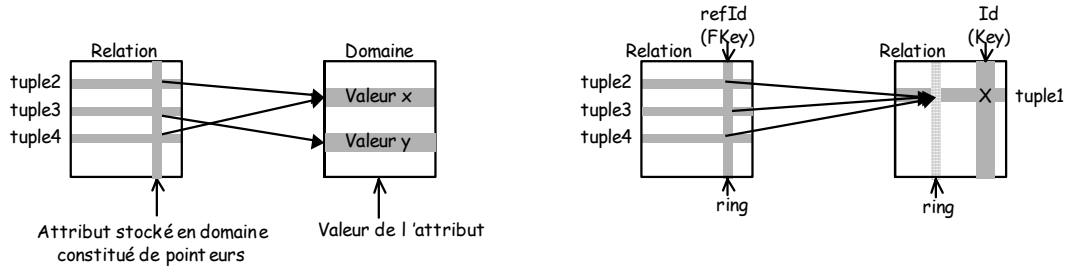
Le modèle de stockage d'un PicoDBMS doit garantir au mieux la compacité des données et des index. Dès lors que la localité des données n'est pas un problème dans notre contexte (règle d'accès), un modèle de stockage basé sur l'utilisation intensive de pointeurs inspiré des SGBD grande mémoire [MiS83, AHK85, PTV90] peut favoriser la compacité. Pour éliminer les doublons, il est possible de regrouper les valeurs dans des domaines (ensemble de valeurs uniques), en utilisant le *stockage en domaine* (DS) comme le montre la Figure 62. Les enregistrements référencent leurs valeurs d'attributs avec des pointeurs. Un domaine peut être partagé par plusieurs attributs. Ce mode de stockage est particulièrement compact pour les types énumérés, qui varient sur un ensemble réduit et déterminé de valeurs[44].

La mise à jour des enregistrements dont les attributs sont en domaine est plus complexe que pour des valeurs stockées à plat, mais leur coût est moindre dans notre contexte simplement parce qu'elles génèrent moins d'écritures. Pour atténuer le léger surcoût du stockage en domaine, nous utilisons ce type de stockage uniquement pour les attributs larges (plus grands que la taille d'un pointeur) contenant des doublons. Naturellement, les attributs sans doublons (les clés) sont stockés à plat. Les attributs de taille variable (généralement plus grand qu'un pointeur) peuvent aussi bénéficier du stockage en domaine même s'ils ne présentent pas de doublons. Dans ce cas, le gain est du à la simplification de la gestion de la

43 Il se peut que l'on ne puisse cependant pas éviter ce type d'organisation.

44 Des techniques de compression peuvent être utilisées de manière complémentaires pour améliorer encore la compacité [Gra93].

mémoire (les tuples des relations sont de taille fixe) et à la compacité des journaux.



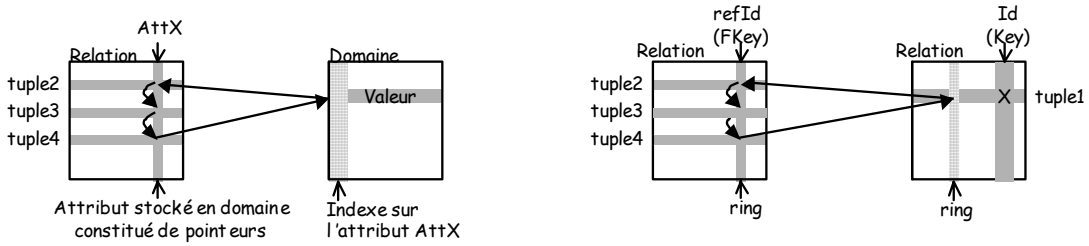

*(a) Stockage d'un attribut classique.* *(b) Stockage d'une clé étrangère.*

**Figure 62** *Stockage en domaine.*

Les index doivent eux aussi être aussi compacts que possible dans la puce. Contrairement aux bases de données traditionnelles stockées sur disque favorisant la localité des données, les puces peuvent utiliser les index secondaires (basés sur des pointeurs) de manière intensive. Tout d'abord, considérons les index de sélection. Un tel index est habituellement composé de deux parties : une collection de valeurs et une collection de pointeurs reliant chaque valeur à tous les enregistrements partageant cette valeur. En supposant que l'attribut indexé prenne valeur sur un domaine, la collection de valeurs correspond au domaine de la base utilisé pour stocker les valeurs distinctes. L'idée est de stocker la collection de pointeurs reliant ces valeurs aux enregistrements concernés en lieu et place des pointeurs reliant l'attribut de chaque enregistrement à la valeur de domaine qu'il prend. Cela nous conduit à construire une structure accélératrice en forme d'anneau partant de la valeur de domaine vers les enregistrements, comme le montre la Figure 63. L'anneau de pointeurs formant l'index de sélection peut aussi être utilisé pour retrouver la valeur d'attribut de chaque enregistrement, constituant ainsi un modèle de stockage des données. Nous appelons ce modèle *stockage en anneau* (RS). Le coût de ce stockage indexé est réduit à son minimum, c'est à dire un pointeur par valeur de domaine, quelle que soit la cardinalité de la relation indexée. Cette économie en stockage est obtenue au prix d'un surcoût de projection des enregistrements, due au parcours de la moitié de la chaîne de pointeurs en moyenne pour retrouver la valeur d'attribut.

Les index de jointure [Val87] peuvent être construits d'une manière similaire. Un prédicat de jointure du type *(R.a=S.b)* sous-entend que *R.a* et *S.b* varient sur le même domaine. Stocker *R.a* et *R.b* en anneau défini un index de jointure. De cette manière, chaque valeur de domaine est reliée via deux anneaux séparés aux tuples de R et de S partageant cette valeur. Comme la plupart des jointures sont effectuées sur un attribut clé, *R.a* étant la clé primaire référencée par la clé étrangère *S.b*, dans notre modèle, les clés primaires sont stockées à plat. Cependant, *R.a* forme précisément un domaine, même s'il n'est pas stocké à l'extérieur de la relation. Comme l'attribut *R.b* prend valeur dans le domaine *R.a*, il référence les valeurs de *R.a* avec des pointeurs. Ainsi, le stockage en domaine offre un index de jointure unidirectionnel naturel (Figure 62.b). Si la jointure doit être optimisée à partir de *R.a* vers *S.b*, un index bi-directionnel est nécessaire. Celui-ci peut être réalisé en définissant un

anneau sur *S.b*, comme le montre la Figure 63. Le coût de cet index de jointure bi-directionnel est réduit à un pointeur par tuple de *R*, quelle que soit la cardinalité de *S*.




*(a) Anneau sur un attribut régulier.* *(b) Anneau sur un attribut clé étrangère.*

**Figure 63** *Stockage en anneau.*

Un autre type d'index plus efficace peut être conçu en combinant RS et DS. Cela conduit à un index de jointure bi-directionnel offrant un suivi de pointeur direct (un seul suivi est nécessaire) de *S.b* vers *R.a*. Ce modèle de stockage amélioré, appelé *stockage en anneau avec pointeur inverse* (RIS), nécessite un pointeur additionnel par tuple de *R* et *S*. Le coût en stockage est beaucoup plus important, le gain obtenu sur les performances doit donc être très significatif pour que ce modèle soit adopté. Une analyse précise du coût de stockage des différents modèles peut être trouvée dans [PBV+01].

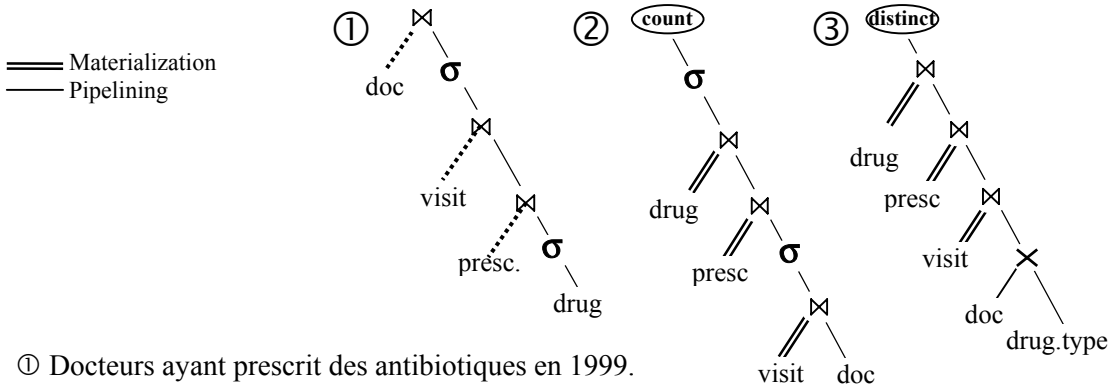
## 3.2 Exécution de Requêtes

L'exécution de requêtes traditionnelle exploite la mémoire de travail pour stocker des structures temporaires (des tables de hachage par exemple) et des résultats intermédiaires. Lorsque la mémoire primaire n'est pas suffisante pour stocker ces données, les algorithmes de l'état de l'art matérialisent ces résultats intermédiaires sur disque. Ces algorithmes ne peuvent pas être utilisés dans un PicoDBMS. D'une part, la règle des écritures proscrit l'utilisation de la mémoire stable pour la matérialisation temporaire. D'autre part, une quantité de mémoire RAM qu'on ne peut estimer a priori ne permet pas d'exécuter les requêtes : avec une petite quantité de RAM on risque le débordement impliquant des écritures en mémoire stable, et avec une grande quantité de RAM on limite fortement la quantité de mémoire stable déjà limitée dans la puce, sans non plus garantir le non-débordement [BKV98]. Enfin, les algorithmes de l'état de l'art sont assez élaborés, ce qui proscrit leur implémentation dans une puce supportant du code simple, compact et compatible avec la règle de sécurité. Pour résoudre ce problème, on peut envisager des techniques d'exécution n'utilisant pas de mémoire de travail et proscrivant les écritures en mémoire stable. Nous décrivons ici les techniques d'évaluation de requêtes simples à complexes, supportant les calculs d'agrégat et l'élimination des doublons.

D'abord, considérons l'exécution des requêtes de sélection–projection–jointure. L'évaluation de la requête se fait habituellement en deux étapes. L'optimiseur génère un plan d'exécution de la requête "optimal". Le plan est ensuite exécuté par un moteur qui implémente un modèle d'exécution et utilise une librairie d'opérateurs relationnels [Gra93].

L'optimiseur peut considérer des plans d'exécution de différentes formes : *arbre gauche*, *arbre droit* ou *arbre bushy*. Les opérateurs des arbres gauches sont exécutés séquentiellement, et chaque résultat intermédiaire est matérialisé. A l'inverse, les arbres droits exécutent les opérateurs en pipeline, évitant ainsi la matérialisation de résultats intermédiaires. Cependant, ils nécessitent de matérialiser en mémoire toutes les relations de gauche. Le arbres bushy offrent l'opportunité de gérer la taille des résultats intermédiaires et la consommation mémoire [SYT93].

Dans un PicoDBMS, l'optimiseur ne considère aucun de ces arbres vu qu'ils nécessitent de la matérialisation. La solution est d'utiliser uniquement l'approche pipeline des arbres extrêmes droits, exécutant en pipeline tous les opérateurs (même la sélection). Les opérandes de gauche étant toujours les relations de base, elles sont déjà matérialisées en mémoire persistante, ce qui permet d'exécuter la requête sans consommer de RAM. L'exécution pipeline peut être réalisée en utilisant le model itérateur [Gra93], où chaque opérateur supporte trois appels de procédures : *open* pour préparer l'opérateur à produire un résultat, *next* pour produire un résultat, et *close* pour effectuer le nettoyage final. L'exécution commence à la racine de l'arbre et se propage jusqu'aux feuilles, le flux de données étant constitué des tuples passés par un nœud fils à son père en réponse à un appel *next*.




① Docteurs ayant prescrit des antibiotiques en 1999.
② Nombre d'antibiotiques prescrits par docteur en 1999.
③ Docteurs et types de drogue prescrits distincts.

**Figure 64** *Plan d'exécution de requêtes complexes*

Les opérateurs de sélection–projection–jointure peuvent être exécutés séquentiellement ou sur un index (anneau). L'utilisation d'index en anneau est toujours la manière la plus efficace d'accéder aux données, mais dans un arbre extrême droit, seule une sélection sur la première relation de base accédée peut être accélérée. Les opérateurs de projection doivent être positionnés à la fin pour éviter toute matérialisation. Sans index, les jointures sont effectuées par boucle imbriquée, la seule technique permettant de se passer de structures ad-hoc (tables de hachage) et/ou de mémoire de travail (tri en mémoire). Avec RS, le coût de la jointure dépend du sens de traversée de l'index. Considérons la jointure indexées de *Doctor(n tuples)* avec *Visit(m tuples)*. Le coût de la jointure partant de *Doctor* est

proportionnel à $n+m$ alors qu'il est proportionnel à $m^2/2n$ en partant de *Visit* (il faut parcourir la moitié de l'anneau en moyenne pour retrouver le docteur associé à chaque visite). La jointure partant de *Visit* peut être plus efficace par boucle imbriquée dans des cas très particuliers (où $m>n^2$). Alors, un index unidirectionnel seul est utile.

Ensuite, considérons l'exécution de requêtes contenant des opérateurs de calculs d'agrégats, tris, et élimination de doublons. L'exécution pipeline parait incompatible avec ces opérateurs, s'exécutant traditionnellement sur des résultats intermédiaires matérialisés. Cette matérialisation est proscrite dans la puce par la règle de la RAM, et sur le terminal par la règle de sécurité. Nous proposons une solution au problème exploitant deux propriétés : (i) les calculs d'agrégat et l'élimination des doublons peuvent être réalisés en pipeline sur un flot de tuples groupés par valeur distincte, et (ii) les opérateurs fonctionnant en pipeline préservent l'ordre des tuples puisqu'ils consomment (et produisent) les tuples dans l'ordre d'arrivée. Ainsi, la consommation dans un ordre adéquat des tuples aux feuilles de l'arbre permet d'effectuer les calculs d'agrégat et l'élimination des doublons en pipeline. Par exemple, la Figure 64 présente des plans d'exécution délivrant des tuples naturellement groupés par *Drug.id* (plan 1), par *Doc.id* (plan 2), par *Drug.type, Doc.id* (plan 3) permettant éventuellement des calculs d'agrégat ou l'élimination des doublons sur ces attributs.

Concernant l'optimisation des requêtes, la fabrication d'arbres extrêmes droits est assez immédiate. L'implémentation d'un optimiseur complexe n'est donc pas requise. Cependant, des heuristiques simples peuvent produire des améliorations significatives. Par exemple, les sélections doivent être appliquées au plus tôt, et l'ordonnancement des accès peut favoriser la traversée des index de jointure dans le sens le plus favorable (une seule possibilité en cas d'index unidirectionnel DS, et de la clé de la relation référencée vers la clé étrangère de la relation référençant en cas d'index bi-directionnel RS/RIS).

## 3.3 Aspects Transactionnels

Comme tout serveur de données, un PicoDBMS doit assurer les propriétés ACID des transactions [BHG87] pour garantir la cohérence des données locales et pouvoir participer à des transactions distribuées. Bien que cela soit hors du sujet de cet article, nous décrivons rapidement les mécanismes transactionnels présentés dans [BPV+01] pour les prendre en compte dans les mesures de performances (section suivante). Ainsi, cette section résume les aspects de contrôle d'intégrité, de concurrence et de coût de journalisation.

Le contrôle d'intégrité peut être assuré par des techniques classiques et ne mérite aucun commentaire particulier. Seules les contraintes d'unicité et d'intégrité référentielle sont considérées dans l'étude de performance. En effet, elles doivent être prises en compte en tant que composante inhérente au model relationnel. Les autres contraintes d'intégrité ne sont pas considérées, vu qu'elles dépendent de l'application.

Concernant le contrôle de concurrence, PicoDBMS a été développé sur des plates-

formes mono-tâches, et ne supporte qu'une connexion à la fois. Dans ce contexte, la concurrence n'est évidement pas un problème. Bien qu'on puisse imaginer des puces multi-tâches (non encore disponibles), cette perspective est clairement hors du sujet de cette étude.

Le procédé de journalisation mérite de plus amples commentaires. La journalisation est nécessaire pour assurer l'atomicité locale, l'atomicité globale (dans un contexte distribué) et la durabilité. Cependant, comme cela est expliqué dans [PBV+01], la durabilité ne peut être assurée dans la puce par PicoDBMS car le support peut être volé ou détruit très facilement. En effet, la petite taille et la mobilité jouent contre la durabilité. La durabilité doit donc être assurée par le réseau. Comme cela est expliqué dans [AGP02], le coût de la durabilité ainsi que celui de l'atomicité local peuvent être entièrement délégués au réseau grâce au protocole *unilatéral de validation atomique* (UCM). Reste le coût induit par l'atomicité locale, décrit dans la suite.

L'atomicité locale peut être assurée de deux manières dans un SGBD. La première technique consiste à mettre à jour *des données ombres* reportées dans la base de façon atomique lors de la validation. Cette technique est adaptée à des puces équipées de mémoire flash de petite taille [Lec97]. Cependant, c'est inadapté à un modèle de stockage basé sur des pointeurs comme RS, car la position des données en mémoire changent à chaque mise à jour. De plus, le coût de cette technique est proportionnel à la taille de la mémoire. Ainsi, soit la granularité des données ombres augmente avec la quantité de données, soit c'est le chemin dupliqué dans le catalogue. Ainsi, le coût en écriture (qui est le facteur dominant) augmente.

La seconde technique assurant l'atomicité locale réalise les mises à jour *en place* [BHG87] dans la base de données, et sauvegarde les informations nécessaires (*Write-Ahead Logging : WAL*) pour défaire la mise à jour en cas d'abandon. Cependant, le coût de cette technique est beaucoup plus important dans un PicoDBMS que dans un système classique, qui bufferise les éléments *WAL* à journaliser pour minimiser les entrées sorties. Dans une puce, le journal doit être écrit à chaque mise à jour, vu que chaque modification devient immédiatement persistante. Cela double environ le coût d'écriture. Malgré ses inconvénients, la seconde technique est plus adaptée aux puces, s'accommodant bien d'un modèle de stockage basé sur des pointeurs et restant insensible à l'augmentation de taille de la mémoire stable. Les éléments *WAL* traditionnels journalisent les valeurs de toutes les données mises à jour. RS permet de journaliser des pointeurs à la place des valeurs, réduisant la taille des éléments à journaliser, et par la même le coût de la technique. Les enregistrements du journal doivent contenir l'adresse du tuple et les images avant des valeurs d'attributs, c'est à dire un pointeur pour chaque attribut stocké en RS, RIS ou DS et une valeur classique dans le cas d'attributs stockés en FS. On peut noter que l'anneau est gratuit en terme de journalisation vu que l'on stocke un pointeur vers la valeur partagée par l'anneau, donnant la possibilité de chaîner les tuples dans l'anneau correspondant au moment de la reprise. Dans le cas de l'insertion ou de la suppression d'un tuple, en supposant que chaque entête de tuple contienne un bit donnant son statut (présent ou effacé), seule l'adresse du tuple doit être inscrite dans le journal pour permettre la reprise. Cela induit un coût supplémentaire pour modifier les

anneaux traversant le tuple. On peut remarquer que l'insertion et la suppression de valeurs de domaines (les valeurs de domaine n'étant jamais modifiées) doivent être journalisées comme toute autre modification. Ce surcoût peut être évité en implémentant un ramasse miettes supprimant les valeurs de domaine non référencées.

Pour conclure, la modification en place avec journalisation des pointeurs et un ramasse miette réduit le coût de journalisation au minimum, c'est à dire à une adresse par tuples inséré ou supprimé et aux valeurs des attributs modifiés (qui là aussi est un pointeur en cas d'attribut stocké en DS, RS, et RIS).

# 4 Mesures de Performances et Enseignements

Cette section évalue les performances de PicoDBMS doté des différentes techniques de stockages et d'indexation proposées, et tire les enseignements de ce travail. Nous déterminons tout d'abord la métrique des évaluations. Ensuite, nous présentons la taille du code du SGBD embarqué. Puis nous évaluons la compacité des données selon le modèle de stockage et d'indexation considéré, et nous conduisons une évaluation complète des performances d'exécution de requêtes combinant des résultats de mesures et de simulations. Enfin, nous jugeons des performances et de l'applicabilité de PicoDBMS sur deux catégories courantes d'applications encartées.

## 4.1 Objectif et Métrique des Evaluations

La métrique et la méthode utilisées pour mener l'évaluation du système sont induites par les caractéristiques suivantes de l'environnement de travail considéré :

(C5) La capacité de stockage est réduite : de quelques dizaines à quelques centaines de kilobytes;

(C6) Les applications visées présentent exclusivement des mises à jour de type "append" : des données historiques sont insérées fréquemment, alors que les modifications et suppressions sont plus rares;

(C7) Les droits d'accès peuvent être sophistiqués : les vues autorisées peuvent combiner projections, sélections, jointures et calculs d'agrégats (cf. Table 9);

(C8) La puce est vue comme un support de stockage intelligent : l'utilisateur (logiciel ou personne physique) se connecte à la puce pour accéder (selon ses droits) aux données embarquées, de façon analogue à tout support de stockage (disque magnétique, clé USB, etc.).

La première caractéristique (C1) souligne l'importance d'évaluer la capacité de stockage potentielle du SGBD, c'est à dire son aptitude à la compression. La question revient à déterminer si (et comment) l'empreinte du code doit être prise en compte dans cette analyse. Dans un cas réel, le code du SGBD serait stocké dans la ROM de la puce, et ainsi ne concurrencerait pas directement le volume de données embarquées. Dans cette optique, seule

l'empreinte totale du code nécessite d'être évaluée pour assurer qu'elle peut être embarquée dans la ROM disponible. Cependant, les fabricants de cartes à puce minimisent les ressources matérielles pour réduire les coûts de production de masse (des millions d'unités sont produites). Ainsi, nous présentons dans les deux sections suivantes la taille de différentes implémentations de PicoDBMS, d'une version épurée stockant les données à plat (sans compression ni index) à une version complète, ainsi que la quantité maximale de tuples embarqués. Les deux types d'évaluations (empreinte du code et des données) sont donc présentés.

La caractéristique (C2) témoigne de l'importance de l'opération d'insertion. Les coûts de mise à jour et de suppression doivent être acceptables, mais sont moins significatifs pour les applications visées. La caractéristique (C3) impose de mesurer la performance d'un large panel de requêtes (*i.e.*, vues), de la plus simple à la plus complexe.

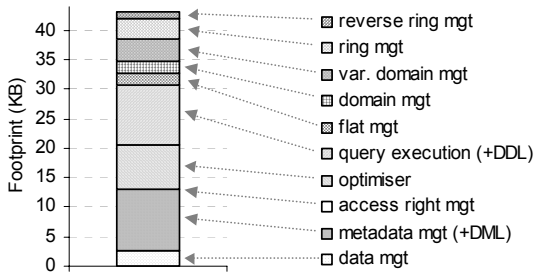
La caractéristique (C4) suggère de présenter des mesures comparables à celles évaluant traditionnellement les supports de stockage, c'est à dire la latence et le taux de transmission. La latence correspond au temps nécessaire à la production du premier tuple résultat. Le taux de transmission peut être exprimé en tuples résultat produits par seconde. Cette métrique est pertinente pour une utilisation du système s'intéressant au taux de transmission (par exemple, pour remplir la fenêtre d'affichage d'un téléphone cellulaire en un temps donné), et aussi au temps total d'exécution de la requête (par exemple, pour trier le résultat avant affichage). Notons que le taux de transmission du dispositif est bien sûr borné par celui de la puce et du lecteur, tout comme la bande passante du bus de données limite le débit véritable des disques magnétiques. Cependant, les lecteurs de cartes sont maintenant connectés au port USB, ce qui laisse augurer un gain de connectivité important à court terme.

## 4.2 Empreinte du Code

Le diagramme Figure 65(a) présente la taille respective des modules composant PicoDBMS et la Figure 65(b) donne l'empreinte totale des principales alternatives d'implémentation. Ces figures ne nécessitent pas d'explications complémentaires. Cependant, nous pouvons en tirer deux conclusions intéressantes, la première sur l'empreinte du code embarqué, la seconde sur le module de gestion des métadonnées.

L'empreinte du code est un défit commun à de nombreux concepteurs de SGBD légers et embarqués. Les éditeurs de SGBD légers exploitent cet argument dans un but commercial. De même, des travaux académiques essayent de gagner sur l'empreinte du SGBD. Notre pensons cependant que la minimisation de l'empreinte du code est moins significative que la compacité des données, ceci pour plusieurs raisons. D'abord, la différence en terme d'empreinte entre la version complète (*Full-fledged*) de PicoDBMS et la version épurée (*Flat-based*) est moins importante que prévu. De plus, la ROM est 4 fois plus dense que l'EEPROM, reduisant d'autant le bénéfice de la réduction de l'empreinte du code vis à vis de l'empreinte des données. Enfin, viser la borne inférieure en terme d'empreinte de code conduit à sélectionner la version la plus épurée, ce qui augmente considérablement

l'empreinte des données (cf. section suivante).




*(a) Version Full-Fledged de PicoDBMS.*

| *version de PicoDBMS* | *Model de Stockage* | *Indexation* | *Empreinte (KB)* |
|---|---|---|---|
| Flat-Based | FS | No | 26 |
| Domain-Based | FS/DS | Jointure | 30 |
| Index-Based | FS/DS/RS | Jointure & sélection | 39 |
| Full-Fledged | FS/DS/RS/RIS | Jointure & sélection | 42 |

*(a) Différentes Variations de PicoDBMS.*

**Figure 65** *Taille du code de PicoDBMS.*

Remarquons que la taille du module de gestion des métadonnées (cf. Figure 65(a)) est relativement importante comparée aux autres. En effet, plusieurs plans d'exécution ne peuvent tenir en RAM simultanément. Ceci exclue le stockage des métadonnées dans des relations de la base, accédées par des requêtes. Les métadonnées sont stockées dans des structures ad-hoc, et sont accédées par des appels de procédures ad-hoc, ce qui explique la lourdeur de l'empreinte du module correspondant.

## 4.3 Empreinte des Données

Cette section évalue la compacité de la base obtenue avec les quatre schémas de stockage et d'indexation candidats (FS, DS, RS, et RIS). L'évaluation ne peut être menée en utilisant un jeu de données synthétique, incapable de retranscrire la distribution et les doublons présents dans les cas réels. Grâce à nos contacts avec des partenaires médicaux, nous avons utilisé un jeu de données médical réel, stocké dans le schéma de base de données médical simplifié présenté Figure 66.

La Figure 67(a) présente le nombre maximal de tuples qui peuvent être stockés grâce à chaque modèle de stockage et d'indexation en fonction de la taille de stockage brute. Les Figure 67(b) et Figure 67(c) traduisent les rapports de capacité entre les différents modèles. Les courbes ne nécessitent pas d'explications supplémentaires. On peut remarquer que RIS est moins efficace en terme de compacité (très proche de FS) que DS et RS. Ainsi, ce modèle sera adopté seulement si le gain obtenu sur l'exécution de requêtes est très significatif. On peut noter aussi que DS et RS présentent des résultats proches, ce qui souligne l'extrême

compacité des index en anneau.

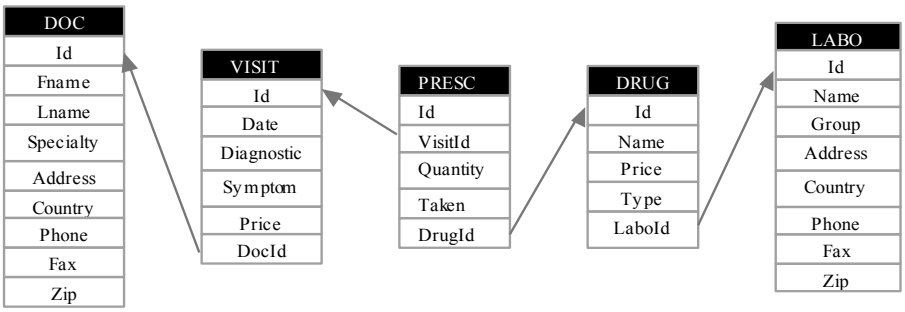
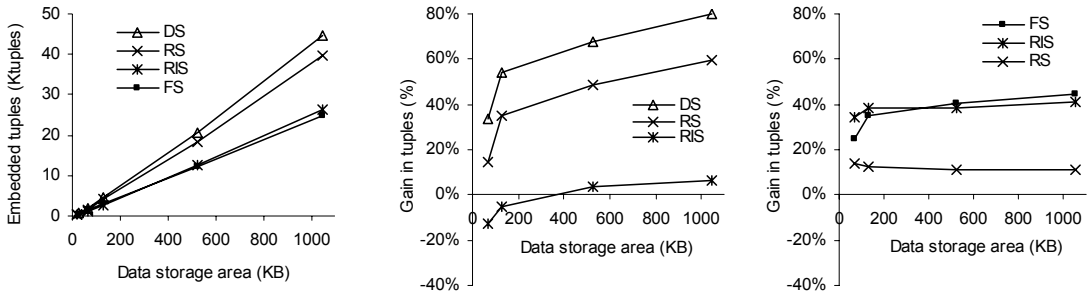



**Figure 66** *Schéma de la base de données.*

On remarque aussi que plus le nombre de tuples embarqué est important (cf. Figure 67(a)), moins le coût de l'indexation est significatif. En effet, la pente de la courbe indiquant le nombre de tuples embarqués en fonction de la capacité de stockage brute est beaucoup plus forte pour les modèles indexés (RS et RIS) que pour FS. De plus, comparé à FS, le gain augment avec la taille de la base (cf. Figure 67(b)). Cette remarque indique que pour de cartes de grande capacité, le coût des index est moindre. La même remarque s'applique à DS, qui agit comme une technique de compression par dictionnaire. L'efficacité de la compression augmente ainsi avec la cardinalité de la base de données.




*(a) Quantité de tuples embarqués.* *(b) Gain par rapport à FS.* *(c) Perte par rapport à DS*

**Figure 67** *Capacité de Stockage de PicoDBMS.*

On remarque aussi que le nombre total de tuples embarqués ne dépasse pas 50.000 pour une carte de capacité de stockage persistant de 1 mégabytes. Nous n'examinons donc pas dans la suite les requêtes traitant plus de données.

## 4.4 Performances de PicoDBMS

Cette section mesure les performances du PicoDBMS en mise à jour (insertion de tuples) et en interrogation (évaluation de vues). Deux plates-formes différentes ont été utilisées pour mener ces expérimentations : un prototype de carte réelle et le simulateur matériel de cette carte (tous deux fournis par Axalto), permettant de mesurer les performances de PicoDBMS sur un jeu de données dépassant largement les capacités de stockage de la carte réelle. Nous

présentons ci dessous ces plates-formes, le jeu de données utilisé et les requêtes, et ensuite les résultats obtenus.

Bien que l'analogie entre la carte à puce et un disque magnétique (cf. Section 4.1) en terme de métrique soit toujours valide, la latence s'avère être négligeable dans tous les cas de figure. En effet, la latence varie de 1 à 4 millisecondes selon la complexité du plan d'exécution. En fait, la latence correspond strictement au temps nécessaire à la construction du plan d'exécution, les tuples étant traités de façon purement pipeline. Ainsi, cette section se concentre sur le taux de transmission, métrique plus significative dans notre contexte.

### *4.4.1 Plate-forme et Jeu de Données*

Le prototype de carte à puce et le simulateur hardware que nous utilisons pour les expérimentations sont des plates-formes 32 bits cadencées à 50MHz, équipées du système d'exploitation Zeplatform[45]. La carte est dotée de 64Ko d'EEPROM qui constituent la mémoire stable disponible à l'utilisateur, 96 kilobytes de ROM stockant le système d'exploitation, et 4 kilobytes de RAM dont quelques centaines d'octets restent libres à l'utilisateur. De plus, elle est dotée d'un timer interne capable de mesurer le temps de réponse à chaque APDU envoyée. Cette plate-forme nous permet d'évaluer les performances sur de petites bases de données (dizaines de kilobytes), le code du PicoDBMS étant stocké en EEPROM dans ce prototype. Pour apprécier les performances sur de plus gros volumes de données, nous utilisons la plate-forme de simulation présentée Figure 68. Elle est constituée d'un simulateur hardware de carte à puce connecté un PC de contrôle, et relié à un PC client connecté à un lecteur de carte (standard ISO 7816). Le simulateur hardware est paramétré par le PC de contrôle avec les caractéristiques de la carte précédemment décrite, et peut émuler une mémoire stable allant jusqu'à 1 mégabytes. Il donne le nombre exact de cycles effectués par le processeur entre deux points d'arrêts du code embarqué. Nous avons validé l'exactitude des mesures du simulateur sur de petits volumes de donnée par confrontation avec les résultats de la carte réelle.

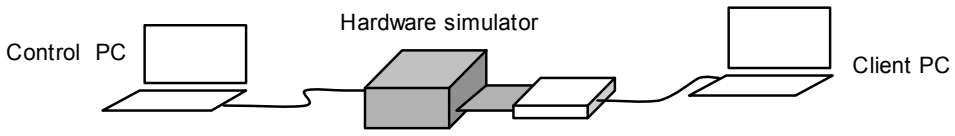



**Figure 68** *Plate-forme de simulation de carte à puce*

Nos expérimentations ont mis en évidence d'importantes entraves aux performances dans ZePlateform en cas d'accès ponctuels intensifs à la mémoire stable (ce cas se présentant assez rarement pour des applications embarquées traditionnelles). Axalto a modifié le cœur de ZePlateform en conséquence, fournissant un bien meilleur support à PicoDBMS. En effet, les accès à la mémoire persistante nécessitent habituellement de coûteuses vérifications de

[45] ZePlateform a été annoncé par Schlumberger/CP8 au salon cartes'2000

sécurité[46], qui s'assurent que l'application embarquée accède une zone autorisée. Cela va à l'encontre du design global de PicoDBMS, basé sur l'utilisation intensive de pointeurs et engendrant ainsi beaucoup d'accès à l'EEPROM. Le noyau du PicoDBMS étant à priori aussi sûr que le système d'exploitation (en supposant qu'il est développé, certifié et installé sur la puce dans les même conditions que le système d'exploitation), les contrôles de sécurité ont été simplifiés, à la fois sur la carte réelle et sur le simulateur hardware. Le gain obtenu sur les performances après cette modification de Zeplatform et le portage de PicoDBMS de JavaCard en C est d'environ deux ordres de magnitude (l'élément de comparaison étant le prototype JavaCard démontré à VLDB'01).

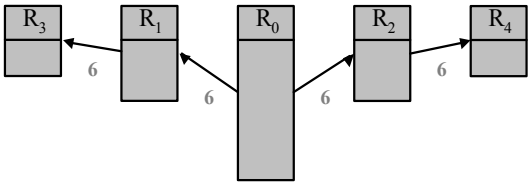



**Figure 69** *Schéma de base de données synthétique.*

Les coûts de communication entre la carte et le terminal ne sont pas pris en compte dans les mesures pour deux raisons. D'une part, les performances du SGBD sont intéressantes indépendamment des caractéristiques de la connectique. D'autre part, les communications constituent un goulot d'étranglement à très court terme : la tendance est actuellement aux cartes connectées par port USB, et des prototypes de carte délivrant 1 mégabytes de données par seconde sont déjà disponibles chez les constructeurs.

Pour évaluer les performances de PicoDBMS, nous utilisons le schéma de base de donnée synthétique constitué de 5 tables présentées Figure 69. Le nombre de tables impliqué est représentatif d'une base de données embarquée d'une certaine complexité, et est relativement proche de celui des benchmarks classiques. De plus, nous avons choisi des ratios de cardinalités semblables au benchmarks TPC-R/H/W, c'est à dire une grosse table référençant deux tables 6 fois plus petites[47], chacune référençant une autre table 6 fois moins peuplée. Les clés primaires et étrangères sont de type numérique et chaque relation possède 10 attributs, 5 de type numérique, et 5 de type chaîne de caractère de taille moyenne 20 octets. Le nombre de valeurs distinctes des attributs de type chaîne de caractère est pris à 1/10 de la taille de la relation, sauf pour les attributs participant à des clauses de groupement pour lesquels le taux est fixé à 1%. La description détaillée de la table $R_0$ est donnée Table 17. Les autres relations sont construites sur le même modèle, avec une seule clé étrangère $A_1$, tandis que $A_2$ a les mêmes propriétés que $A_3$ et $A_4$. Notons que l'attribut $A_0$ est stocké en FS vu qu'il est unique et ne peut bénéficier des modèles DS, RS ou RIS. $A_3$ et $A_4$ ne sont jamais

[46] plus de 2000 cycles processeur sont nécessaires aux dispositif de sécurité pour autoriser chaque accès à l'EEPROM.

[47] TPC-W (resp. TPC-R/H): 2 tables référencent la plus grosse de cardinalité moyenne 3 et 7.8 (resp. 4 et 7.5) fois supérieure à celles-ci.

stockés en DS vu que les occurrences sont plus petites que la taille d'un pointeur (4 bytes).

| *Nom des attributs* | *Contrainte* | *Type du contenu* | *Type de stockage* | *Valeurs distinctes* | *Taille (Bytes)* |
|---|---|---|---|---|---|
| $A_0$ | clé primaire | numérique | FS | $\|R_0\|$ | 4 |
| $A_1$, $A_2$ | clés étrangères | numérique | FS/DS/RS/RIS | $\|R_0\|/6$ | 4 |
| $A_3$, $A_4$ | | numérique | FS/RS/RIS | $\|R_0\|/10$ | 4 |
| $A_5$, $A_6$, $A_7$, $A_8$, $A_9$ | | alpha- numérique | FS/DS/RS/RIS | $\|R_0\|/10$ | 20 (moyenne) |

**Table 17** *Schéma de la Relation $R_0$.*

Différentes requêtes sont évaluées (cf. Table 18), chacune étant représentative de l'un des droits d'accès présentés Section 2.3.

| *Acronyme* | *Description du droit d'accès* | *Requête représentative correspondante* |
|---|---|---|
| *P* | Sous-ensemble des attributs | Projection de $R_0$ |
| *SP* | Prédicats sur une seule relation | Sélection de $R_0$ |
| *SPJ* | Prédicats sur deux relations (association directe) | Sélection de $R_0$, jointure de $R_0$ et $R_1$ |
| $SPJ^n$ | Prédicats sur plusieurs relations (association transitive) | Multi-sélections, jointure des relations $R_0$-$R_1$-$R_2$-$R_3$-$R_4$ |
| *SPG* | Une relation, groupement et calcul d'agrégat mono-attribut | Groupement mono-attribut de $R_0$, un calcul d'agrégat |
| *SPJG* | N relations, groupement et calcul d'agrégat mono-attribut | Jointure de $R_0$, $R_1$, $R_3$, groupement mono-attribut, un calcul d'agrégat |
| $SPJG^n$ | groupement et calcul d'agrégat multi-attributs | Jointure de $R_0$, $R_1$, $R_3$, groupement sur deux attributs de deux relations distinctes, un calcul d'agrégat |

**Table 18** *Jeu de requêtes des évaluations de performance.*

Dans la suite, chaque courbe est fonction d'un paramètre donné. D'autres paramètres sont considérés comme ayant les valeurs par défaut ci-dessous.

- *Cardinalité de la liste des projections* : le nombre d'attributs à projeter est de 2 (les mesures montrent que cela n'est pas crucial pour les performances).
- *Sélectivité des sélections* : la cardinalité du résultat d'une requête contenant une sélection est fixée à 1% de celle de la même requête sans sélection (par exemple, la requêtes SP de la Table 18 délivre 1% des tuples de $R_0$).
- *Sélectivité des jointures* : les jointures sont des equi-jointures sur clé. Ainsi, le nombre de tuple résultat est égal à la cardinalité de la relation référençant (joindre $R_0$ et $R_1$ donne $R_0$ tuples).

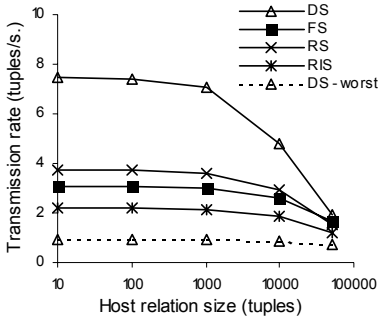
  *Facteur de groupement* : La cardinalité du résultat d'une requête de groupement est fixée à 1% de celle de la même requête sans groupement. En d'autres termes, la cardinalité de chaque groupe produit est 100.

Les attributs sur lesquels les projections, les sélections et les groupements sont appliqués ne sont pas spécifiés Table 18. En effet, la même requête est mesurée sur différents

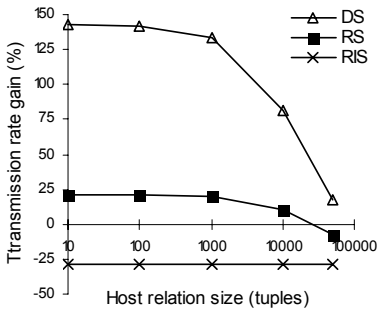
attributs, nous permettant de présenter le meilleur et le pire des cas (*best case* et *worst case*). Une seule courbe est représentée lorsque la différence entre les deux n'est pas significative.

### *4.4.2 Insertions*

Dans un contexte de base de données append, les performances d'insertion sont cruciales. Nous évaluons le taux d'insertions (nombre de tuples insérés par seconde) dans la base synthétique. L'insertion étant une opération unitaire, la latence est comptabilisée dans cette mesure.




*(a) Taux de Transmission.*




*(b) Gain de Performance Relatif versus FS.*

*(c) Décomposition des coûts.*

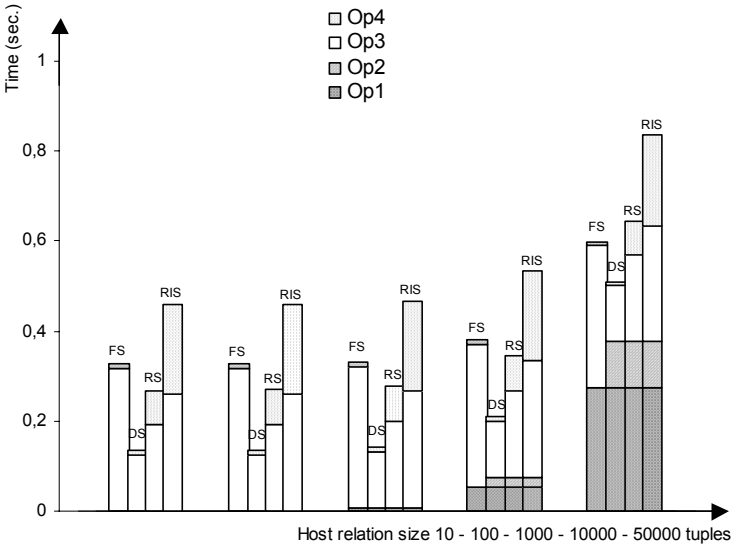



**Figure 70** *Performances d'insertion.*

Dans les expérimentations, les tuples sont insérés dans la relation $R_0$. En effet, $R_0$ étant la plus grosse relation du schéma en terme de cardinalité, c'est celle dans laquelle on insère le plus. De plus, les insertions dans cette relation représentent le pire des cas en terme de performance vu que cela implique le coût de contrôle d'intégrité le plus grand : $R_0$ possède une clé étrangère de plus que les autres relations (cf. Table 17), les domaines de ses attributs (pour ceux stockés en domaine) ont la plus grande cardinalité et le nombre d'occurrences de

clé primaire est aussi le plus grand. Plus précisément, l'insertion met en jeu les quatre opérations principales suivantes :

(Op5) *Vérification des contraintes d'intégrité des clés primaires et étrangères* : l'insertion d'un tuple impose le parcours de la relation hôte (vérification de la contrainte d'unicité) et le parcours de 1/6 (d'après les rapports de cardinalités des relations) de chaque relation référencée.

(Op6) *Mise à jour des structures de stockage* : pour RS, DS et RIS, la valeur de chaque attribut nouvellement inséré doit être recherchée dans le domaine correspondant, et la valeur est ajoutée au domaine si elle n'y existe pas encore. La Figure 70 montre le coût le plus fréquent, considérant que les domaines sont statiques ou du moins assez stables après un certain nombre d'insertions.

(Op7) *Allocation du tuple inséré* : cette opération inclue l'écriture des valeurs et des pointeurs dans l'espace alloué au nouveau tuple, et éventuellement les mises à jour nécessaires à la maintenance des anneaux traversant le tuple.

(Op8) *Maintient du journal* : cette opération correspond au coût de la journalisation basée sur des pointeurs comme nous l'avons présentée Section 3.3.

La Figure 70(a) montre que le taux de transmission est convenable pour chaque modèle de stockage (plus d'un tuple par seconde[48]). Les gains relatifs de chaque modèle de stockage par rapport à FS sont explicités Figure 70(b). Cette figure montre que la compacité du modèle de stockage améliore les performances d'insertion pour de petits volumes de données (jusqu'à 1000 tuples dans $R_0$). En effet, comme clarifié Figure 70(c), le coût élevé des opérations d'écriture Op3 et Op4 pour FS est dans ce cas dominant. Après ce seuil, les coûts de Op1 et Op2, dépendant directement de la cardinalité des relations, et écrasent les écarts entre les différents modèles de stockage. Notons que l'opération Op1 pourrait être accélérée en utilisant un arbre-B sur les clés primaires, mais ce gain doit être pondéré par le coût de maintient de l'index (écritures en mémoire persistante) et le surcoût en espace de stockage.

### *4.4.3 Projections*

La Figure 71 donne le taux de transmission des requêtes de projection représentatives du droit d'accès *P*. Les courbes sont obtenues pour chaque modèle de stockage en fonction du nombre d'attributs projetés. Les conclusions suivantes peuvent être tirées :

- *Le taux de transmission est indépendant de la cardinalité de la relation* : bien sûr, l'opérateur de projection requière le même temps pour produire chaque tuple.

[48] La limite d'une seconde est souvent considérée pour des applications nécessitant une interaction humaine.

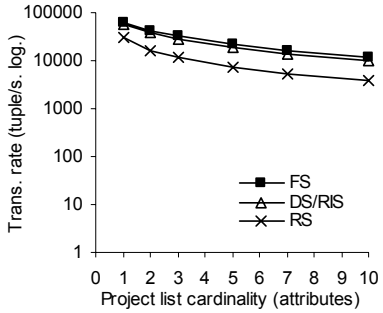



**Figure 71** *Performance des requêtes de projection (P).*

- *RS ralentit la projection* : RS induit en moyenne le parcours d'un demi-anneau pour atteindre la valeur d'un attribut donné. Dans notre jeu de données, la taille moyenne des anneaux est de 10 pointeurs, ce qui conduit à la dégradation des performances d'un facteur 5 (une échelle logarithmique est utilisée Figure 71).

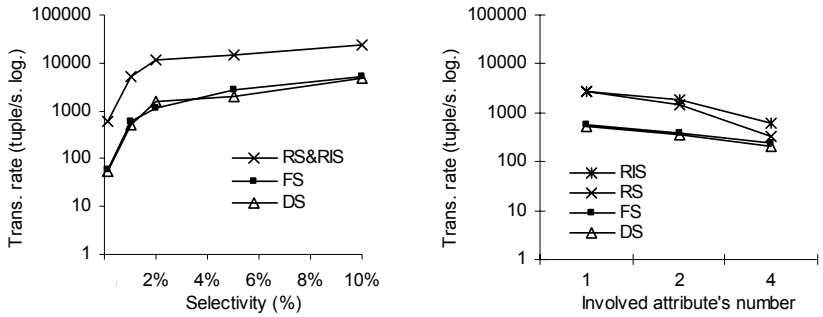
- *La projection n'est pas une entrave aux performances* : même dans le cas d'attributs stockés en RS il n'y a pas de problème, le débit étant supérieur à 5000 tuples par seconde quel que soit le model de stockage considéré et le nombre d'attributs projetés.

### *4.4.4 Sélections*

La Figure 72 présente le taux de transmission des requêtes de sélection mono-relation, représentatives du droit d'accès *SP*. Le taux de transmission est présenté en fonction de la sélectivité de la requête sur la Figure 72(a) et en fonction de la complexité de la sélection (nombre d'attributs impliqués dans le prédicat) sur la Figure 72(b). Ces courbes permettent de faire les remarques suivantes :

- *Le taux de transmission des sélections est indépendant de la cardinalité de la relation* : l'opérateur de sélection, comme pour la projection, nécessite un temps constant pour produire chaque tuple.

- *Le taux de transmission augmente avec la diminution de complexité de la requête* : en effet, plus la sélectivité est faible, moins nombreux sont les tuples disqualifiés traités par l'opérateur avant de produire un résultat.




*(a) Sélection mono-attribut.* *(b) Sélection multi-attributs.*

**Figure 72** *Performances des requêtes de sélection (SP).*

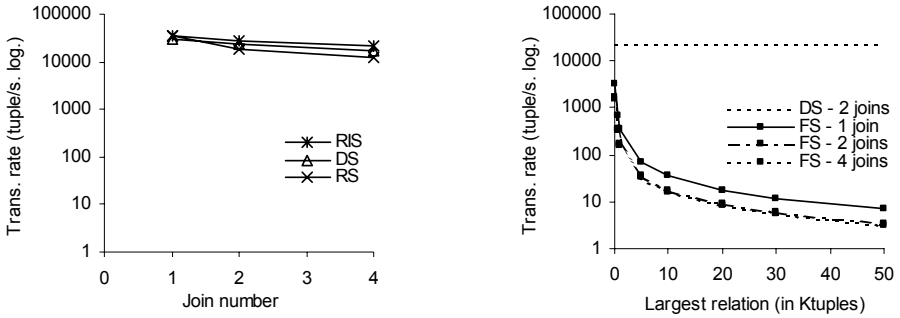
- *Les models RIS et RS sont plus efficaces que DS et FS:* ces modèles de stockage offrent un index de sélection naturel. Le gain obtenu sur la performance est déterminé par le rapport de cardinalité du domaine et de la relation (fixé à 10 dans notre jeu de données). Cependant, lorsque plusieurs attributs sont impliqués dans le critère de sélection, la différence de performance entre les modèles de stockage indexés et non-indexés s'atténue, soulignant le fait que notre modèle d'exécution ne puisse exploiter plusieurs index pour une même requête.

### *4.4.5 Jointures*

La Figure 74 présente la performance des requêtes représentatives des droits *SPJ* et $SPJ^n$, basées sur des sélections et des jointures, avec différentes sélectivités. La Figure 73 montre la performance de requêtes de jointures pures selon le nombre de relations impliquées. Ces deux figures sont nécessaires pour analyser l'incidence de la sélection sur les requêtes de jointure. Considérons d'abord les conclusions qui peuvent être tirées de la Figure 73 :

- *Le taux de transmission de la jointure est élevé pour RIS, RS et DS:* les modèles de stockage RIS, RS et DS présentent un débit élevé pour l'exécution de requêtes de jointures pures, indépendamment de la taille de la base (cf. Figure 73(a)). De plus, la performance est assez stable lorsque le nombre de jointures augmente. L'exécution d'un plan impliquant *N* opérateurs de jointure et de projection est ralentie grosso modo d'un facteur *1/(N+2)* pour chaque jointure supplémentaire. Par exemple, ajouter une deuxième (respectivement, troisième) jointure ralentit les performances de la requêtes de 30% (respectivement 25%). Ce comportement s'explique par la nature même des trois modèles considérés. Comme cela est présenté Section 3.1, le modèle DS offre naturellement un index de jointure unidirectionnel de *S.b* vers *R.a*, où *S.b* représente une clé étrangère et *R.a* la clé primaires correspondante (*R.a* est considéré comme un domaine vu qu'il ne contient pas de doublon). RS et RIS rendent l'index bidirectionnel. Le meilleur ordonnancement des jointures privilégie le parcours de la clé primaire vers la clé étrangère, permettant de produire chaque tuple résultat d'une requête effectuant *N* jointures en parcourant *N* pointeurs de domaine. Ceci explique la faible différence de

performance entre RS, RIS et DS sur la figure. Cependant, cette situation favorable est remise en cause dès lors que les sélections imposent un autre ordonnancement des jointures. Ce point est discuté en détail ci-dessous.



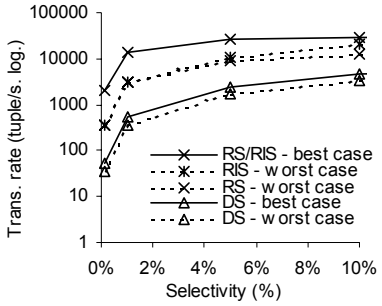

*(a) Taux de transmission DS/RS/RIS* *(b) Taux de transmission FS*

**Figure 73** *Performances des requêtes de jointures pures.*

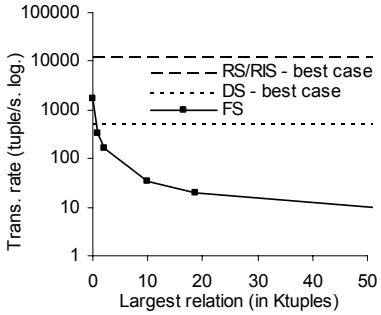
- *Le taux de transmission est faible pour FS :* FS présente de faibles performances par rapport aux trois autres modèles. De plus, FS entraîne une dégradation rapide des performances avec l'augmentation de la taille de la base (cf. Figure 73(b)).

Jointures et sélections sont souvent combinées dans une même requête. Pour prendre en compte cette situation, la Figure 74 mesure les performances des requêtes SPJ et SPJ$^n$. Le taux de transmission est présenté en fonction de la sélectivité pour les modèle RIS, RS et DS (que se montrent indépendant de la cardinalité de la base), et en fonction de la cardinalité de la base pour FS (avec une sélectivité par défaut fixée à 1%). Dès lors que des sélections sont appliquées sur différentes relations, le plan d'exécution généré peut favoriser différents modèles selon l'ordonnancement des jointures. Pour illustrer cela, nous présentons Figure 74 les bornes maximales et minimales du taux de transmission pour chaque modèle de stockage.

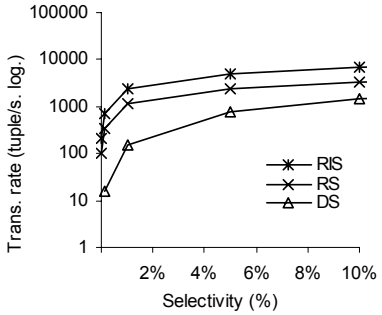
Bien qu'elles ne nécessitent pas d'explications supplémentaires, ces courbes induisent les remarques suivantes. D'abord, la performance de la jointure augmente lorsque la sélectivité de la sélection diminue, quel que soit le modèle de stockage considéré. Comme pour les requêtes de sélection, plus faible est la sélectivité, moins important est le nombre de tuples à écarter avant de produire un tuple qualifié. Ensuite, RS et RIS sont clairement meilleurs que DS lorsque l'on considère des sélections. Comme nous l'avons mentionné précédemment, DS impose un ordonnancement unique des jointures dans le plan d'exécution, rendant ainsi impossible l'utilisation d'un index de sélection. Pour cela, le choix de RS ou de RIS pour stocker les attributs clé étrangère est le plus pertinent.



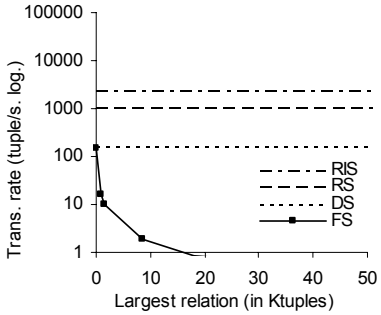

*(a) Jointure avec Sélection (SPJ) – DS/RS/RIS.*




*(b) Jointure avec Sélection (SPJ) – FS.*




*(c) Multi-Jointures avec Sélection (SPJ$^n$) – DS/RS/RIS.*




*(d) Multi-Jointures avec Sélection (SPJ$^n$) – FS.*

**Figure 74** *Performance des requêtes de jointure avec sélection (SPJ, SPJ$^n$).*

### *4.4.6 Agrégats*

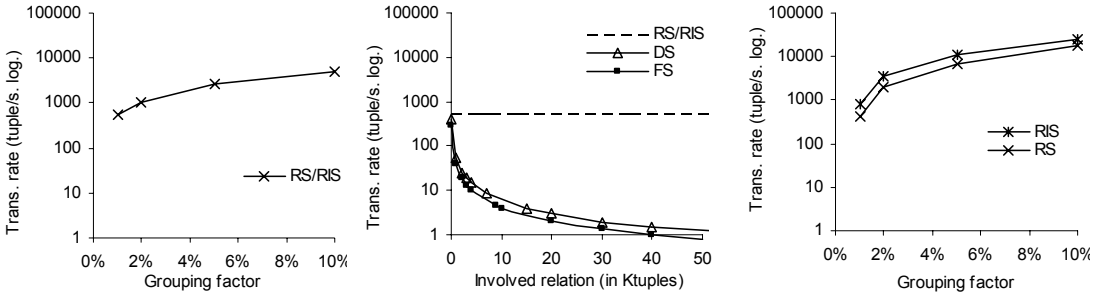
Les requêtes d'agrégats calculent des données autorisées (CA), donnant accès à des valeurs calculées sans permettre l'accès aux occurrences entrant dans le calcul, ce qui constitue une classe d'autorisations assez fréquente et importante. La Figure 75 présente le taux de transmission des requêtes d'agrégat, représentatives des droits SPG, SPJG et SPJG$^n$ (cf. Table 18). Ces courbes conduisent aux remarques suivantes:

- *RIS et RS supportent aisément les agrégats mono-attribut* : L'évaluation de la requête part du domaine contenant les valeurs de groupement et parcours ensuite un ou plusieurs anneaux (selon les jointures impliquées) pour atteindre les tuples partageant la valeur de groupement courante. Ainsi, grâce aux anneaux, le taux de transmission des requêtes d'agrégation mono-attribut est relativement similaire à celui des requêtes non-agrégatives (cf. Figure 75(a) et (c)). On remarque que la performance est indépendante de la cardinalité de la base comme pour les requêtes de jointure, mais que le coût dépend toutefois du facteur de groupement. Par exemple, un facteur de groupement de 10% signifie que le résultat de la requête contient 10 fois moins de tuples après l'agrégation, conduisant à une réduction des performances d'un facteur 10.

- *DS et FS supportent mal l'agrégation :* le taux de transmission est faible pour ces

deux modèles, même dans le cas d'agrégation mono-attribut, et dépend fortement de la cardinalité de la base. En effet, DS induit un produit cartésien entre le domaine contenant la clé de groupement et la relation, et FS nécessite des parcours séquentiels successifs des relations pour produire le résultat. Lorsque des jointures sont introduites dans la requête, la performance de DS reste stable (grâce à un débit de jointure élevé) alors que celle de FS s'effondre (moins d'un tuple produit par seconde).

- *RIS et RS supportent convenablement l'agrégation multi-attributs* : comme le montre la Figure 75(e), le taux de transmission dépend alors de la cardinalité de la base quel que soit le modèle de stockage. En effet, bien qu'avec RS et RIS, seul un attribut impliqué dans la clause *group by* peut bénéficier du stockage en anneau. Cependant, la performance reste acceptable avec RS et RIS avec encore 80 tuples résultats produits chaque seconde.




*(a) Mono-agrégation (SPG) – RS/RIS.* *(b) Mono-agrégation (SPG) – FS/DS.* *(c) Mono-agr., Jointures (SPJG) – RS/RIS.*

*(d) Mono-agrégation, Jointures (SPJG) – FS & DS.* *(e) Multi-agrégations, Jointures (SPJG$^n$).*

**Figure 75** *Performances des requêtes d'agrégation (SPG, SPJG, SPJG$^n$).*

### *4.4.7 Conclusion sur les Taux de Transfert*

Comme le montrent les sections précédentes, le taux de transmission atteint pour chaque classe de requêtes est fortement dépendant du modèle de stockage choisi. Pour aider à comparer ces modèles, la Table 19 exprime leur performance relative en terme de rapports

pour chaque classe de requête. Pour chaque ligne de la table, la cellule grisée sert de référence à l'établissement des rapports. Les taux de transmission ont été calculés en fixant la cardinalité de la relation $R_0$ à 1000.

| *Type d'opération* \ *Modèle de stockage* | | *FS* | *DS* | *RS* | *RIS* |
|---|---|---|---|---|---|
| *Insertion* | | **1** | 4,8 | 1,1 | 0,7 |
| *Interrogation* | *P* | **1** | 0,9 | 0,5 | 0,9 |
| | *SP* | **1** | 0,9 | 8,7 | 8,7 |
| | *SPJ* | 0,08 | **1** | 23,4 | 23,4 |
| | $SPJ^n$ | 0,01 | **1** | 7,1 | 15,2 |
| | *SPG* | 0,7 | **1** | 136 | 136 |
| | *SPJG* | 0,002 | **1** | 118 | 228 |
| | $SPJG^n$ | 0,002 | **1** | 19 | 31 |

**Table 19** *Rapports des taux de transmission.*

DS donne le meilleur taux de transmission en insertion et est meilleur que FS lorsque des jointures doivent être opérées dans la requête (d'un à trois ordres de magnitude). L'ajout d'anneaux apporte un bénéfice majeur pour les évaluations de jointures et les calculs d'agrégats (amélioration d'un à deux ordres de magnitude par rapport à DS). Cependant, le gain obtenu grâce à RIS comparé à RS est relativement décevant (au plus un facteur 2 lorsque plusieurs jointures sont impliquées), surtout lorsque l'on considère la perte qu'il engendre en terme de compacité de la base. Néanmoins, on peut noter que pour des schémas de bases de données particuliers, comme les schémas en étoile où une grande relation centrale référence de nombreuses petites relation par l'intermédiaires de grand anneaux de pointeurs (par exemple, une centaine de pointeurs), le modèle de stockage RIS peut devenir intéressant.

## 4.5 Conclusion et Etudes de Cas

Cette section donne des remarques qui concluent les mesures. Pour cela, nous mettons en pratique les évaluations de performances pour montrer comment ces résultats peuvent permettre de sélectionner le modèle de stockage le plus adapté à une application donnée. Nous considérons dans la suite deux profiles d'applications. Le premier s'intéresse au temps d'exécution totale des requêtes, cas fréquent lorsque les tuples résultats ne peuvent être délivrés en pipeline pure (par exemple, lorsque qu'une clause *order by* est appliquée au résultat final). Nous supposons que le temps de réponse requis dans ce profile d'application est d'une seconde. Le second profile d'applications considéré s'intéresse plutôt au débit de PicoDBMS. Ce cas est représentatif des applications tournant sur un téléphone cellulaire devant afficher le résultat à l'utilisateur avec une certaine rapidité. Nous supposons pour ce profile d'application que le taux de transmission minimal nécessaire est de 10 tuples par seconde. Pour ces deux profiles, nous nous concentrons sur les requêtes d'interrogation vu que les insertions sont acceptables (cf. Section 4.4.1) quel que soit le modèle de stockage.

Considérons d'abord le premier profile d'application, s'intéressant au temps total d'exécution de la requête. La Table 20 présente le pourcentage du résultat total produits en une seconde pour chaque classe de requête, selon la cardinalité de la base et le modèle de stockage sélectionnés. Notons que la cardinalité de la base est elle-même déterminée par la capacité de stockage de la carte à puce et par le modèle de stockage choisi. Par exemple, la cardinalité potentielle de la base pour une carte d'un mégabytes est de 24K tuples (avec FS) à 44K tuples (avec DS). Les cellules répondant au besoin de l'application sont grisées. La Table 20 montre que FS ne peut satisfaire aux besoins de l'application pour les requêtes d'agrégat, même dans de petites cartes disposant de 64 kilobytes (seul 11% du résultat est délivré dans la seconde). Alors que DS satisfait les besoins pour des cartes de 64 kilobytes, RS et RIS sont nécessaires pour des cartes plus grosses. Comme nous l'avons mentionné précédemment, la différence entre RS et RIS n'est pas significative dans ce contexte.

| *Capa-cité de la puce* | *Tuples stockés* | *FS* | | | | | | *Tuples stockés* | *DS* | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | *SP* | *SPJ* | *SPJ$^n$* | *SPG* | *SPJG* | *SPJG$^n$* | | *SP* | *SPJ* | *SPJ$^n$* | *SPG* | *SPJG* | *SPJG$^n$* |
| 64KB | 1428 | 100 | 100 | 100 | 100 | **11** | **11** | 1906 | 100 | 100 | 100 | 100 | 100 | 100 |
| 128KB | 2978 | 100 | 100 | **39** | **95** | **0** | **0** | 4593 | 100 | 100 | 100 | **59** | **61** | **52** |
| 512KB | 12293 | 100 | **40** | **1** | **5** | **0** | **0** | 20590 | 100 | 100 | 100 | **2** | **2** | **2** |
| 1MB | 24864 | 100 | **9** | **0** | **1** | **0** | **0** | 44730 | 100 | 100 | **48** | **1** | **0** | **0** |
| | | *RS* | | | | | | | *RIS* | | | | | |
| 64KB | 1637 | 100 | 100 | 100 | 100 | 100 | 100 | 1247 | 100 | 100 | 100 | 100 | 100 | 100 |
| 128KB | 4015 | 100 | 100 | 100 | 100 | 100 | 100 | 2829 | 100 | 100 | 100 | 100 | 100 | 100 |
| 512KB | 18250 | 100 | 100 | 100 | 100 | 100 | **43** | 12736 | 100 | 100 | 100 | 100 | 100 | 100 |
| 1MB | 39650 | 100 | 100 | 100 | 100 | 100 | **13** | 26426 | 100 | 100 | 100 | 100 | 100 | **41** |

**Table 20** *Pourcentage de tuples du résultat transférés en 1 seconde.*

Considérons maintenant que le profile de l'application cible nécessite un taux de transmission minimal. La Table 21 présente le nombre de tuples délivrés en résultat par seconde pour chaque modèle de stockage, sous les mêmes conditions que les résultats présentés Table 20. Les cellules satisfaisant les besoins de l'application sont grisées. Le modèle DS satisfait aux besoins pour des cartes disposant de moins de 128 kilobytes de mémoire persistante. Pour des bases plus grandes, le modèle RS est nécessaire, et là encore RIS ne présente pas d'avantage significatif.

A la lumière de ces deux tables, nous pouvons tirer les conclusions globales suivantes. Les requêtes de sélection et de projection ne posent jamais de problème, quel que soit le modèle de stockage considéré. La difficulté principale est donc de supporter efficacement les jointures et les agrégations. Pour atteindre cet objectif, RS est généralement requis, sauf dans le cas d'une carte de faible capacité où DS peut être acceptable. La question finale posée concerne alors la généralité de ces conclusions. En d'autres termes, pouvons nous imaginer d'autres modèles de stockage et d'indexation ? Nous avons la conviction que cette étude

couvre les principaux compromis existant en terme de techniques de stockage et d'indexation, à savoir le mode de stockage brute (plat) vis à vis du modèle compressé (domaine) et le modèle non-indexé vis à vis des structures indexées accélérant les sélections et les jointures. Bien sûr, différentes variations des techniques étudiées peuvent être envisagées, mais les écarts de performance entre nos techniques et leurs éventuelles variantes restent faibles dans un contexte proche de celui des bases grande mémoire. La différence de performance entre les modèles RS et RIS n'est que la confirmation de cette allégation. Au mieux, des structures d'indexation multi-attributs pourraient être conçues pour accélérer les requêtes d'agrégation multi-attributs, ce qui constitue une forme particulière de requête pré-calculée.

| *Capacité de la puce* | *Tuples stockés* | *FS* | | | | | | *Tuples stockés* | *DS* | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | SP | SPJ | SPJ$^n$ | SPG | SPJG | SPJG$^n$ | | SP | SPJ | SPJ$^n$ | SPG | SPJG | SPJG$^n$ |
| 64KB | 1428 | >10 | >10 | >10 | >10 | **1** | **1** | 1906 | >10 | >10 | >10 | >10 | >10 | >10 |
| 128KB | 2978 | >10 | >10 | >10 | >10 | **0** | **0** | 4593 | >10 | >10 | >10 | >10 | >10 | >10 |
| 512KB | 12293 | >10 | >10 | **2** | **4** | **0** | **0** | 20590 | >10 | >10 | >10 | **4** | **3** | **3** |
| 1MB | 24864 | >10 | >10 | **0** | **2** | **0** | **0** | 44730 | >10 | >10 | >10 | **3** | **2** | **2** |
| | *RS* | | | | | | | *RIS* | | | | | | |
| 64KB | 1637 | >10 | >10 | >10 | >10 | >10 | >10 | 1247 | >10 | >10 | >10 | >10 | >10 | >10 |
| 128KB | 4015 | >10 | >10 | >10 | >10 | >10 | >10 | 2829 | >10 | >10 | >10 | >10 | >10 | >10 |
| 512KB | 18250 | >10 | >10 | >10 | >10 | >10 | >10 | 12736 | >10 | >10 | >10 | >10 | >10 | >10 |
| 1MB | 39650 | >10 | >10 | >10 | >10 | >10 | >10 | 26426 | >10 | >10 | >10 | >10 | >10 | >10 |

**Table 21** *Nombre de tuples minimal transférés par seconde.*

# 5 Perspectives de Recherche

De nombreux problèmes de recherche s'inscrivent dans la suite de l'étude PicoDBMS. Cette section présente le sous-ensemble de ces problèmes nous apparaissant comme générateurs des problèmes de recherches futurs les plus intéressants. Nous avons abordé deux de ces problèmes, que nous détaillons donc plus largement, l'un d'entre eux ayant été à l'origine d'une publication [ABP03].

Cette section décrit les principales perspectives de cette étude et, plus généralement, du contexte base de données embarquées.

*Environnement sans contact* : les constructeurs et les organisations gouvernementales [SIN02b] portent un intérêt grandissant aux puces sans contact pour leur facilité d'utilisation. Bien que PicoDBMS puisse être intégré tel quel dans une carte sans contact, l'environnement diffère par de multiples aspects, ce qui peut amener à une conception différente. Par exemple, contrairement aux technologies sans contacts, le temps d'exposition au lecteur est relativement long dans le cadre d'une carte avec contact (un dossier médical peut rester plusieurs dizaines de minutes dans un lecteur, tout comme une carte stockant un environnement utilisateur peut être connecté longtemps à un terminal ou à un PC). La remise

en cause de l'objectif en terme de temps de réponse, induite par l'interaction avec une personne physique et souvent considérée comme étant de l'ordre de la seconde, impose de nouvelles techniques d'exécution. Ainsi, les requêtes interrogeant des populations de carte (menant par exemple une étude épidémiologique pendant que la carte d'un patient est insérée dans le lecteur d'un médecin) doivent aussi s'accommoder de ce temps d'exposition réduit. Ceci peut conduire à d'intéressantes problématiques visant à exécuter des requêtes très rapidement, quitte à fournir un résultat approché (évitant au porteur de stationner devant le lecteur).

*Utilisation de nouvelles technologies de mémoire persistante* : nous nous sommes concentrés sur la technologie EEPROM lors de l'étude de PicoDBMS. Cependant, l'EEPROM ayant atteint sa limite de scalabilité d'après certains constructeurs (la ligne de gravure ne peut plus être affinée), de nouvelles technologies comme la FLASH font leur apparition dans les puces. La mémoire de type FLASH présente ses propres caractéristiques qui doivent être prises en compte dans le design des composants base de données embarqués. La perspective intéressante consistant à adapter les composants embarqués à la mémoire FLASH a fait l'objet d'une première étude [BSS+03], s'intéressant exclusivement aux écritures sur un petit volume de données (centaines de tuples). Une étude considérant de plus larges volumes et orientée vers l'évaluation de droits d'accès complexes serait très bénéfique. De même, les technologies alternatives à long terme, comme PCM, OUM ou Millipèdes, pourraient être envisagées. Ces mémoires exhibent aussi des propriétés d'accès spécifiques ayant un impact direct sur le modèle de stockage et d'indexation de la base. Des travaux existent déjà dans un contexte plus général sur Millipèdes [YAE03]. La combinaison de ces propriétés mémoires avec les autres contraintes matérielles génèrerait certainement des problèmes intéressants.

*Calibrage des ressources* : les ressources matérielles embarquées ont un impact immédiat sur le coût de la puce, surtout lorsque celle-ci s'adresse à un marché de masse. Il est donc particulièrement important de calibrer au plus juste les ressources matérielles à intégrer d'après les besoins de l'application cible. Nous avons mené dans ce contexte une première étude du calibrage de la RAM [ABP03]. En effet, la RAM présente une cellule de très faible densité, occupant ainsi actuellement le tiers de la puce, et représente ainsi le tiers de son prix (le coût d'une puce est directement lié à sa taille). Cette ressource est donc cruciale. Nous résumons dans la suite les technologies d'exécution mises en ouvre pour utiliser/calibrer le plus efficacement possible la RAM nécessaire aux composants base de données embarqués. Les recherches futures menées sur ce point devraient viser à calibrer la puce selon ses trois dimensions principales (processeur, RAM, mémoire persistante) selon le besoin de l'application. La consommation processeur peut être minimisée par l'utilisation intensive d'index et la matérialisation, et la quantité de mémoire persistante peut être modulée par compression des données de bases et des structures accélératrices. Bien sûr, les différentes dimensions du problème ne sont pas indépendantes. Tous cela fait du co-design complet de la puce face au besoin de l'application un défit extrêmement intéressant.

*Délégation de ressources* : dans de nombreux cas d'applications, des ressources externes à la puce peuvent être utilisées. Ce genre d'étude peut être vu sous deux angles différents. D'une part, cela répond à l'objectif ultime du calibrage des ressources de la puce, non seulement face aux besoins de l'application, mais encore en prenant en compte dans l'analyse l'environnement externe à la puce. Le challenge principal est naturellement de conserver le très haut degré de sécurité offert par PicoDBMS. Nous avons mené une première étude dans ce contexte, adressant les mêmes applications que PicoDBMS sur une architecture disposant de capacité de stockages et de traitement extérieures à la puce, en cherchant à assurer le degré de sécurité le plus proche de PicoDBMS. Nous détaillons les problèmes de recherche posés par cette première étude dans la suite. Vue sous un angle différent, cette étude constitue un premier pas vers un problème très large de sécurisation des données et des traitement réalisés sur des plates-formes présentant un niveau de confiance faible (PC/PDA relié à un réseau, serveur base de données hébergé par un DSP, etc.). Cet objectif ambitieux impose de résoudre plusieurs problèmes additionnels majeurs. En effet, le contexte de PicoDBMS suppose que la base de données est accédées via une même carte à puce. Cela signifie que (i) il n'y a pas d'accès concurrents et (ii) partager les données imposent de partager la carte. Supposer que la base est stockée sur un serveur (potentiellement attaquable par un pirate) et retirer les hypothèses précédentes conduit à un spectre d'applications beaucoup plus large, comme un modèle DSP [HIL+02] dans lequel les utilisateurs possèderaient chacun une carte à puce sécurisant l'accès à des données extérieures chiffrées. Ce sujet particulièrement intéressant a généré plusieurs travaux dont l'objectif est d'assurer la confidentialité des données [HIL+02, BoP02, DDJ+03, IMM+04], sans toutefois considérer (i) que le client puisse être le pirate, (ii) que le traitement lui-même puisse être attaqué (le serveur retourne les mauvais blocs de données, une ancienne version, etc.) et (iii) que les données distantes puissent être modifiées de façon mal-intentionnée (le client essaye de s'auto prescrire des médicaments). L'extension de l'étude préliminaire à cette problématique plus générale génèrerait certainement des problèmes intéressants. En effet, lorsque plusieurs utilisateurs accèdent les données chiffrées en même temps, chacun par l'intermédiaire de sa propre puce, les problèmes traditionnels de gestion de transaction apparaissent et son compliqués par les propriétés d'opacité et de résistance aux attaques qui doivent être mises en ouvre sur les données et les traitements.

*Modèle de données XML* : Les études menées sur PicoDBMS l'on étés en considérant le modèle relationnel. Etudier un PicoDBMS XML est naturellement une perspective intéressante. En effet, XML est approprié pour stocker des environnements utilisateurs (VHE, Virtual Home Environment). Considérer le modèle XML soulève différents problèmes. D'abord, de nombreuses façon de stocker et d'indexer des données XML ont été proposées, cependant cela reste à faire dans le cadre des cartes à puce (à notre connaissance). Ensuite, le contrôle d'accès et les droits sur des données XML proposées dans la littérature sont basés sur XPATH et [BDP04] a proposé un évaluateur de droits d'accès sur des flots dans une carte. Ce problème est fortement lié à celui de la délégation de traitement, cependant, la vérification de droits sur des données locales impose des modifications

notables. Enfin, considérer à la fois XML et de nouvelles technologies mémoires est un problème intéressant.

## 5.1 Calibrage de la Quantité de Mémoire Volatile

Le calibrage des ressources est crucial dans le contexte de la carte à puce, et plus généralement dans celui des petits objets intelligents (capteurs météorologiques ou de trafic routier, composant embarquées dans les appareils domestiques, etc.) subissant des contraintes techniques et/ou économiques. En effet, l'exiguïté de l'environnement cible (dispositif embarqué dans le corps), la puissance d'alimentation limitée et la résistance aux attaques (prévient l'observation directe des composants) entraînent la minimisation des ressources matérielles du composant intelligent. De plus, lorsque ces composants sont produits à grande échelle (c'est le cas des cartes à puce), la diminution des coûts de production impose la réduction de la taille du chip (le coût est directement lié à la taille du support de silicium) et réduisant ainsi les ressources embarquées. Dans ce contexte, la RAM est une ressource particulièrement cruciale vue sa faible densité comparée aux autres composants (une cellule de RAM est 16 fois plus volumineuse qu'une cellule de ROM); elle représente dans une puce près du tiers du prix (matériel) de la puce. Elle doit donc être soigneusement calibrée, et minimisée selon les besoins de l'application. L'étude [ABP03] réalise un premier pas dans cette direction, en considérant (i) des données relationnelles stockées à plat et sans index[49], (ii) un dispositif autonome (on s'interdit de se reposer sur des ressources distantes et/ou du swap externe à la puce), (iii) des droits d'accès basés sur des requêtes plates (non imbriquées) comme dans le contexte PicoDBMS, et (iv) de la mémoire persistante électronique (accès rapide en lecture) comme c'est le cas dans les puces.

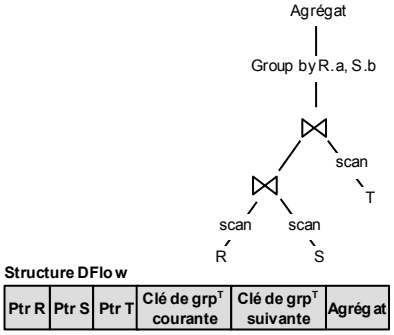
Cette étude est conduite en 3 étapes, que nous résumons dans cette section. La première étape détermine la quantité minimale de RAM nécessaire à l'évaluation de requête, sans se soucier des performances. La deuxième étape consiste à améliorer/optimiser les performances de ce modèle d'exécution à consommation mémoire minimale. La troisième étape s'attache à étudier l'impact sur le modèle précédent de l'ajout incrémental de RAM.

### *5.1.1 Modèle d'Exécution à Consommation Minimale*

La consommation RAM d'un plan d'exécution correspond à la taille cumulée des structures de données internes des opérateurs et des résultats intermédiaires traversant l'arbre d'opérateurs. Le modèle d'exécution doit suivre deux règles pour consommer le minimum de

[49] Les index ne sont pas considérés dans l'étude. La raison principale est de rendre le problème analysable, et de voir jusqu'où on peut aller sans index. D'autres raisons peuvent aussi justifier notre analyse sans index : (i) ils augmentent le coût des mises à jour et compliquent l'atomicité, (ii) ils sont nécessairement absents dans certains contextes évaluant des requêtes sur des flots de données, (iii) ils sont difficiles à maintenir sur certaines technologies de mémoire comme la FLASH gros grain considérée dans [BSS+03], (iv) ils rivalisent avec les données embarquées en terme d'espace de stockage, (v) vu que tous les attributs ne sont pas nécessairement indexés, certaines requêtes doivent être évaluées sans disposer d'index.

mémoire : (R1) la règle d'invariabilité proscrit toute structure de donnée dont la taille est fonction du volume de données interrogées et (R2) la règle de minimalisme interdit le stockage de toute information qui peut être recalculée. Ces deux règles régissent la synchronisation des opérateurs et minimisent le stockage interne, conduisant à une exécution pipeline stricte, représentée Figure 76.




```
GBY.Open ChildOp.Open();Split.Value←+∞          // Scan the whole input in
  repeat                                        // order to find the smallest grouping value (GV) in Split
    ChildOp.Next()
    if Dflow➔GrpLst < Split.Value then Split←DFlow.GrpLst
  until DFlow.Child = EOF                       // Split converges to the smallest GV at EOF
GBY.Next
  if Split.Value ≠ +∞ then                      // there is a next GV to compute
    Current←Split; DFlow.AggLst←0 //  Initialize the computation of the current GV
    Split.Value←+∞; ChildOp.Open() //  Initialize Split for computing the next GV
    repeat                                      // scan the whole input
      ChildOp.Next()
      if Dflow➔GrpLst = Current.Value then      //the current tuple shares the GV
        compute DFlow➔AggLst
      elseif DFlow➔GrpLst∈ ] Current.Value,Split.Value [ then  // Split converges
        Split←DFlow.GrpLst
    until DFlow.Child = EOF
    DFlow.GrpLst←Current                        // Output an aggregate
```

**Figure 76** *Requête exemple et Dflow.* **Figure 77** *Algorithme GroupBy.*

L'algorithme des différents opérateurs du modèle d'exécution à consommation minimale découle de ces règles. Nous les représentons dans [ABP03] selon le modèle itérateur [Gra93] disposant de l'interface *Open, Next, Close,* où *Open* initialise les structures de données internes, *Close* les libère et *Next* produit le tuple suivant de l'opérateur. L'algorithme de groupement (*GroupBy*) est détaillé Figure 77. Lors de l'Open, les valeurs des clés de groupement courantes et suivantes sont fixées respectivement à $-\infty$ et à $+\infty$, et l'opérateur consomme entièrement son entrée à la recherche de la clé de groupement minimale qu'il stocke dans l'attribut de Dflow C*lé de grp*$^T$ *courante*. Sur appel à Next, l'opérateur parcourt le flot d'entrée jusqu'à rencontrer un tuple partageant la clé courante qu'il délivre alors en résultat; chaque tuple consommé par l'opérateur fait éventuellement évoluer la clé de groupement suivante stockée dans C*lé de grp*$^T$ *suivante* lorsque la clé rencontrée en entrée est strictement comprise entre les valeurs courantes et suivantes stockées dans Dflow. Lorsque l'opérateur rencontre la fin du flot d'entrée, il stocke remplace la clé courante par la suivante, ouvre à nouveau son flot d'entrée et répète le processus.

Ce modèle d'exécution, consommant le minimum de RAM, offre de faibles performances dues aux re-itérations successives sur les données engendrées par les opérateurs de groupement (décrit précédemment) et de jointure, qui ne peut être réalisé que par boucle imbriquée. La conséquence dramatique sur le nombre de réitérations (nombre de Open) réalisées est présentée Figure 78. Avec $k$ clés de groupement distinctes à l'entrée du *GroupBy*, le parcours de $T$ est effectué $k \times ||\ R\ joint\ S\ ||$ fois.

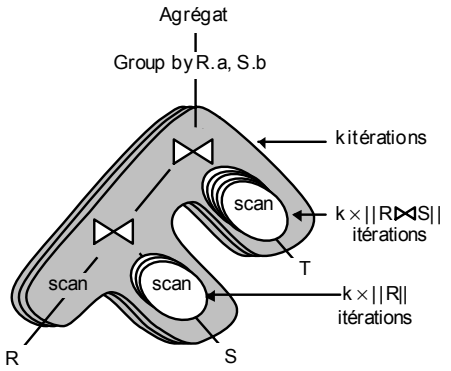



**Figure 78** *Itérations nécessaires à l'évaluation à consommation minimale.*

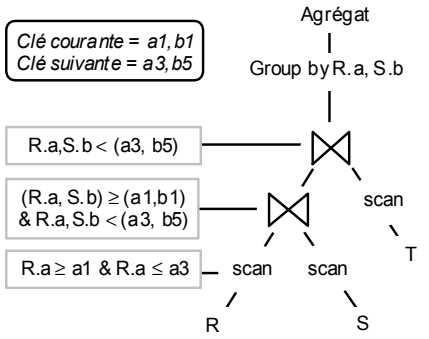



**Figure 79** *Filtres d'itération de l'algorithme de groupement.*

### *5.1.2 Optimisation du Modèle : les Filtres d'Itération*

Des solutions diverses et complémentaires peuvent être envisagées pour réduire le coût de ces itérations sans consommer de RAM supplémentaire. Des techniques traditionnelles d'optimisation globale minimisent le nombre d'itérations total par réorganisation de l'arbre d'exécution, par exemple en opérant les sélections le plus tôt possible et en ordonnant les jointures de façon optimale (envisagées dans [ABP03]).

En outre, nous avons imaginé une technique d'optimisation locale[50] originale réduisant le nombre de tuples considérés au sein de chaque itération. Le flot de tuples entrant de chaque opérateur peut être séparé en trois ensembles distincts: les tuples *Nécessaires* sont ceux qui participent au résultat produit lors de l'itération en cours (pour le groupement, ceux qui partagent la clé courante), les tuples *Indispensables* sont ceux qui modifient l'état interne des opérateurs sans participer au résultat de l'itération courante (pour le groupement, ceux qui partagent une clé de groupement située entre les clés courante et suivante), et les tuples *Inutiles* qui ne sont ni *Nécessaires* ni *Indispensables*. Du filtrage doit être mis en place pour éviter de traiter et d'acheminer ces tuples dans l'arbre d'exécution jusqu'à l'opérateur. La distinction des tuples *Nécessaires*, *Indispensables* et donc *Inutiles* dépend de chaque opérateur. Une fois cette distinction faite, le filtrage des tuples *Inutiles* peut être mis en œuvre (l'étude de minimisation du nombre de tuples *Nécessaires*, conduisant à choisir un algorithme plutôt qu'un autre, est réalisée dans [ABP03]). Chaque opérateur souhaitant éliminer les tuples *Inutiles* de son flot d'entrée exprime un prédicat, appelé filtre d'itération, qui sélectionne uniquement les tuples *Nécessaires* et *Indispensables* pour une itération donnée. Ce prédicat est ensuite vérifié par les opérateurs du sous-arbre, produisant ce flot d'entrée. Conceptuellement, un filtre d'itération est similaire à un prédicat de sélection qui serait appliqué le plus tôt possible dans l'arbre d'exécution. Toutefois, les filtres d'itération peuvent impliquer plusieurs attributs de plusieurs relations de base. Ils sont donc plus

[50] Notons que le terme d'optimisation locale a un sens différent de son utilisation classique, puisqu'il s'applique à une itération plutôt qu'à un opérateur.

complexes que des restrictions classiques, ce qui impose de s'attacher à la manière de les vérifier pour éviter des opérations redondantes. La Figure 79 montre les filtres d'itérations réduisant le flot d'entrée de l'opérateur de groupement. La clé de groupement correspondante à l'agrégat en cours de calcul est (*a1*, *b1*), et la valeur immédiatement supérieure parvenue à l'opérateur de groupement dans l'état actuel de l'exécution est (*a3*, *b5*). Les opérateurs du sous arbre peuvent donc éliminer les instances du produit cartésien partageant une clé strictement inférieure à (*a1*, *b1*) ou supérieure à égale à (*a3*, *b5*).

Le modèle d'exécution proposé en RAM minimale s'adapte naturellement à une quantité de RAM bornée additionnelle disponible lors de l'exécution par ajout d'autres instances du produit cartésien (au lieu d'une seule) dans *Dflow*. Chaque itération produit ainsi plusieurs résultats. Cependant, cette utilisation de la RAM complique la vérification des filtres d'itération, vu que celle-ci s'opère sur des ensembles. On peut cependant facilement balancer l'efficacité des filtres avec le coût de vérification.

## 5.2 Délégation de Ressources

Le stockage persistant sera toujours limité dans les puces. C'est une contrainte à long terme, vu que (i) les technologies mémoire basées sur un matériel ultra dense (Millipèdes, PCM, OUM, etc.) permettant de lever la contrainte de stockage persistant ne sont pas encore au stade de production, et (ii) les puces ont une contrainte de coût qui fait qu'elles sont basées sur de vieilles technologies moins coûteuses rendant peu envisageable l'intégration de technologies ultra pointues dans la puce. De son coté, le débit des puces avec l'environnement extérieur est en train d'exploser. De plus, dans de nombreux contextes, le microcontrôleur embarqué peut bénéficier de ressources distantes capables de stocker les données de la base et d'effectuer des calculs. Cette section jette les bases de l'utilisation de ce type de ressources dans le contexte de PicoDBMS, avec pour objectif d'assurer un niveau de sécurité aussi proche possible de celui offert par PicoDBMS.

L'architecture considérée peut être schématisée comme suit. La puce constitue la ressource offrant opacité et résistance aux attaques autant aux données embarquées qu'aux calculs internes. L'environnement extérieur non sécurisé, destiné à stocker les données de la base, est sujet aux attaques et n'offre aucune garantie aux données stockées ainsi qu'aux calculs que la puce lui délègue. Le terminal est la ressource présentant le résultat final à l'utilisateur; il est donc considéré comme ayant les mêmes droits que celui-ci, et dispose aussi éventuellement de ressources de stockage et de calcul. L'architecture considérée est synthétisée Figure 80, et différentes instances de cette architecture sont détaillées dans le Chapitre 5.

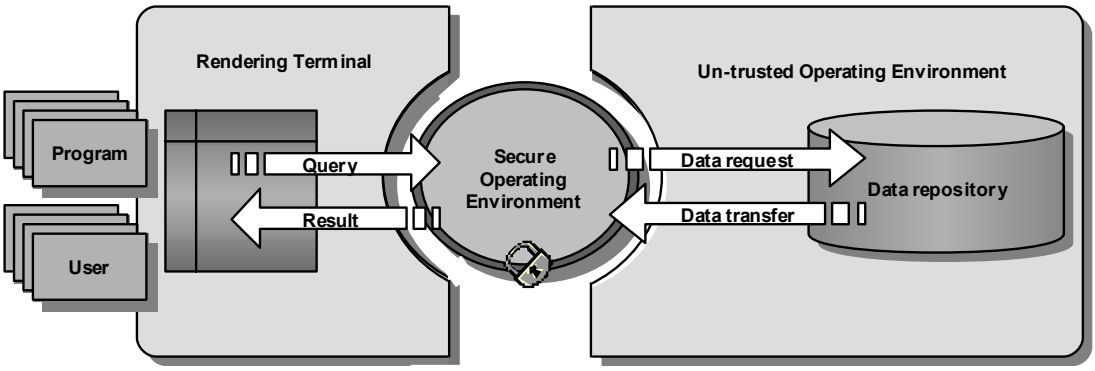



**Figure 80** *Architecture de référence.*

Dans le cadre de PicoDBMS, les données et les traitements étant confinés dans la puce, la protection offerte par la puce constitue la résistance du système. L'architecture considérée ici peut subir des attaques différentes, menées (i) par un intrus infiltrant le système et essayant de déduire des informations des données et/ou des traitements réalisés hors de la puce, (ii) par un utilisateur du système essayant d'outrepasser ses droits d'accès pour lire ou modifier des données, (iii) par l'administrateur de l'environnement extérieur non sécurisé conduisant des attaques sur la ressource dont il a la charge, ou (iv) par un voleur dérobant tout ou partie du système (notamment lorsque celles-ci sont portables). Les attaques menées contre le système (les données et/ou les traitements) peuvent être catégorisées comme suit: *Examen* (le pirate examine les données pour en déduire de l'information), *Altération* (le pirate modifie des données), *Substitution* (le pirate remplace des données valides par d'autres), *Ré-émission* (le pirate remplace des données valides par d'anciennes versions)

L'objectif de la délégation de ressources est d'approcher le niveau de sécurité offert par PicoDBMS, c'est à dire d'offrir opacité et résistance aux attaques aux données stockées hors puce ainsi qu'aux traitements réalisés. L'opacité consiste à assurer que strictement aucune information ne transparaît des données ou des traitements externes. La résistance aux attaques garantie que le pirate ne peut altérer ni les données ni les traitements externes afin d'accéder à des informations outrepassant ses droits. Concernant les données externalisés, la protection peut être assurée par des techniques cryptographiques telles que le chiffrement des blocs de données externes (DES, Triple-DES, AES, etc.) et l'intégration d'information additionnelle (somme de contrôle ou hash, estampille ou version, emplacement du bloc, etc.). Les puces sont capables d'exécuter de telles procédures (contiennent souvent un co-processeur cryptographique), cependant le surcoût engendré par de telles techniques doit être minimisé. Concernant les traitements réalisés à l'extérieur, toute opération effectuée doit être vérifiée par la puce, ce qui peut parfois être aussi coûteux que d'effectuer l'opération elle-même. Pour des opérations de sélection, de telles techniques existent [GMK+04], cependant les opérations plus complexes restent problématiques. Ainsi, la résistance d'un traitement externe parait difficile à assurer. Son opacité semble aussi problématique vu que même effectué sur des données chiffrées, des informations bénéfiques au pirate peuvent

transparaître de l'opération.

Le problème de la délégation de ressources revêt ainsi trois dimensions principales: (i) la dimension architecturale consiste à répartir les traitements et les données (même temporaires) sur les différents composants du système, (ii) la dimension cryptographique consiste à protéger les données et traitement par des techniques cryptographiques contre les 4 formes d'attaques exposées précédemment, et (iii) la dimension évaluation de requête permettant d'évaluer les droits d'accès du contexte PicoDBMS de façon compatible avec les besoins de l'application sur des données externes protégées par des techniques cryptographiques, de façon adaptée à l'architecture. Notre approche du problème est décrite dans les paragraphes suivants.

Comme nous l'avons mentionné, déléguer des traitements à l'extérieur donne des armes au pirate. De plus, la vérification des traitements délégués (l'environnement extérieur peut rendre des résultats erronés dans le but d'outrepasser les droits) est un processus complexe et très coûteux. Pour lever ces entraves, nous envisageons une solution drastique à la dimension architecturale : minimiser les traitements délégués à l'environnement non sécurisé. Cela conduit l'environnement externe à implémenter une interface basique, servant des blocs de données chiffrées désignés par une adresse et une taille, du type *RendDonnée(adresse, taille)→Donnée*. Cette option offre un haut degré d'opacité, seuls les accès aux données ainsi que les informations nécessaires aux communications (nombre et taille des messages échangés) étant exposés au pirate. Assurer plus d'opacité est équivalent au problème de *Récupération Privée d'Information* [CGK+95], qui est clairement hors de propos ici. De plus, cette approche rend soluble le problème de vérification des traitements et de résistance aux attaques, conduisant à simplement s'assurer que les blocs récupérés sont non-altérés (intègres), qu'ils n'ont pas été substitués (provienne de l'adresse demandée), et correspondent à la dernière version. Ainsi, l'évaluation des vues autorisées à l'utilisateur est confinée dans la puce, à l'exception des accès aux données externes, nécessairement délégués. Pour procéder ainsi, la puce doit contenir au moins une zone d'amorce, des clés de chiffrement, et des informations de version utilisées pour se protéger de la ré-émission. De plus, la mémoire persistante de la puce peut être utilisée comme cache des données fréquemment accédées, telles que les méta-données et les droits d'accès. Le terminal, ayant les mêmes droits que l'utilisateur, peut être utilisé pour (i) compléter l'évaluation de la vue sur des données autorisée, (ii) calculer des requêtes basées sur les vues autorisées, et (iii) rendre le résultat final. Ainsi, le terminal peut effectuer la jointure des vues autorisées $V_1$ et $V_2$ sans perte de sécurité.

Dès lors que les données sont stockées à l'extérieur, l'impact des contraintes de la puce sur le moteur d'évaluation est différent. En effet, les données doivent être déchiffrées et vérifiées conformes (non attaquées) dans la puce, ce qui augmente fortement le coût des accès[51]. Ainsi, les techniques dévaluation de requêtes basées sur des itérations successives

[51] Le coût du déchiffrement d'un bloc de 8 bytes avec la puce ayant servit aux expérimentations de

sur les données (cf. Section 5.2) sont proscrites. En outre, des techniques d'indexation intensives envisagées dans PicoDBMS ne peuvent résoudre complètement le problème, sauf à matérialiser toutes les vues. Cependant, l'architecture envisagée dispose de ressources externes, il est donc envisageable d'y allouer une zone de swap. On peut noter que les données swapées doivent bien sûr être opaques et résistantes aux attaques, ce qui (grossièrement) ramène le coût du swap au double de celui d'un accès aux données de base. En conséquence, nous proposons d'utiliser la faible quantité de RAM disponible dans la puce[52] (par exemple, 16 kilobytes) et une zone de swap externe afin d'utiliser des algorithmes dérivés de l'état de l'art (par exemple, la jointure par tri fusion). Pour maximiser l'utilisation de la RAM, nous proposons aussi de suivre le principe consistant à ramener en RAM, à chaque étape de l'exécution, les seules données strictement nécessaires à l'étape en cours. Par exemple, une jointure par tri fusion peut être réalisée en disposant seulement de la clé des tuples et des attributs de jointure.

Ainsi, l'approche retenue dans ce contexte peut être synthétisée comme suit :

1. *Utiliser les index de manière intensive (index de sélection et de jointure)*
2. *Se baser sur des algorithmes de l'état de l'art utilisant la RAM de la puce et une zone de swap externe*
3. *A chaque étape, accéder uniquement les attributs utiles à cette étape*

Cette approche n'a du sens que si l'on peut offrir des accès fin aux données externes. En effet, l'utilisation d'index de sélection et de jointure (premier point) conduit à accéder le sous-ensemble pertinent des tuples (sous-ensemble horizontal) et le troisième point suppose l'accès à un sous-ensemble des attributs (sous-ensemble vertical). Nous ajoutons donc un dernier point à notre approche:

4. *Offrir un accès grain fin aux données externes*

L'accès *grain fin* dépend non seulement du support non sécurisé de stockage externe (plus difficile à offrir en présence d'un disque qu'avec de la mémoire FLASH) et de la façon dont le support est connecté à la puce (USB, réseau, bus, etc.), mais encore de la granularité des techniques cryptographiques envisagées.

# 6 Conclusion

Dans cette étude, nous considérons des applications traitant des données confidentielles, accédées par plusieurs utilisateurs ayant des privilèges différents sur les données. PicoDBMS

PicoDBMS est de l'ordre de 50 µs avec un Triple-DES, ce qui représente plusieurs centaines de fois le coût de lecture dans la mémoire persistante de la puce.

[52] On peut noter que dans ce contexte, la quantité de données admissibles dans la base n'est pas inversement proportionnelle à la quantité de RAM disponible, les données étant stockées hors carte.

[PBV+01] permet d'embarquer des données confidentielles dans une puce électronique et de vérifier les privilèges des utilisateurs en embarquant aussi un moteur d'évaluation de requêtes. Nous sommes allés plus loin que l'étude préliminaire de PicoDBMS: nous avons catégorisé les droits d'accès nécessaires dans le contexte de pico-bases de données, et nous avons évalué les performances de différents modèles de stockage et d'indexation pour atteindre cet objectif. Pour cela, nous avons défini trois catégories principales d'autorisations, couvrant un large panel de situations: les autorisations sur le schéma, sur les occurrences et sur des données calculées. Nous avons ensuite dérivé sept types de droits d'accès relatifs à ces catégories qui devraient être supportés dans un contexte de type PicoDBMS. Nous avons ensuite conduit des expérimentations de performances complètes, basées sur un jeu de requêtes représentatives des différents types de droits d'accès, à la fois sur un prototype de carte réel et sur un simulateur hardware émulant les cartes de future génération. Ces évaluations ont montré la pertinence et l'intervalle de fonctionnement de chaque structure de données et des index selon les droits d'accès nécessaires à l'application et selon la quantité de données embarquées. L'analyse conduite permet aussi de sélectionner les structures de stockage appropriées d'après le volume de données embarquées, les droits d'accès et les temps de réponse à satisfaire. Par exemple, si l'on considère une application pour laquelle le temps de réponse total des requêtes doit être inférieur à la seconde, le stockage en domaine (DS) satisfait l'exigence pour une carte de 64 kilobytes de mémoire, mais au-delà les structures en anneaux (RS et RIS) sont nécessaires. Des conclusions similaires peuvent être tirées pour toute exigence d'application.

Enfin, les perspectives de travail sont nombreuses. La thèse présente des études préliminaires dans le domaine de la co-conception et de la délégation de ressources. En effet, les mesures de performances de PicoDBMS peuvent être utilisées pour déterminer la limite de cette approche du "*tout indexé*" (aucune consommation RAM, exécution pipeline pure), et du "*tout embarqué*" (les données confidentielles et le moteur d'évaluation de requêtes) en terme de temps de réponse et de volume de données. Les chapitres 3 et 5 de la thèse apportent une première contribution à ces problèmes.

# References

[ABB+01] N. Anciaux, C. Bobineau, L. Bouganim, P. Pucheral, P. Valduriez, "PicoDBMS: Validation and Experience", International Conference on Very Large Data Bases (VLDB), 2001.

[ABF+92] P. M. G. Apers, C. A. van den Berg, J. Flokstra, P.W. P. J. Grefen, M. L. Kersten, A. N. Wilschut, "PRISMA/DB: A Parallel Main Memory Relational DBMS", IEEE Transactions on Knowledge and Data Engineering (TKDE), 1992.

[ABM04] K. Atasu, L. Breveglieri, Marco Macchetti, "Efficient AES implementations for ARM based platforms", ACM Symposium on Applied Computing, 2004.

[ABP03a] N. Anciaux, L. Bouganim, P. Pucheral, "Memory Requirements for Query Execution in Highly Constrained Devices", International Conference on Very Large Data Bases (VLDB), 2003.

[ABP03b] N. Anciaux, L. Bouganim, P. Pucheral, "On Finding a Memory Lower Bound for Query Evaluation in Lightweight Devices", Technical Report, PRiSM Laboratory, 2003.

[ABP03c] N. Anciaux, L. Bouganim, P. Pucheral, "Database components on chip", ERCIM News, 2003.

[ABR+01] A. Abrial, J. Bouvier, M. Renaudin, P. Senn, P. Vivet, "A New Contactless Smart card IC using an On-Chip Antenna and an Asynchronous Micro-controller", IEEE Journal of Solid-state Circuit, 2001.

[Acc02] Accenture, "Governing in the e-economy - Accenture selected egovernement credential", 2002. url=www.dpsa.gov.za/documents/networks/eGov/GoverningInTheE-economyAppendixMay'02.pdf

[ADH+01] A. G. Ailamaki, D. J. DeWitt, M. D. Hill, M. Skounakis, "Weaving Relations for Cache Performance", International Conference on Very Large Data Bases (VLDB), 2001.

[ADH02] A. Ailamaki, D. J. DeWitt, M. D. Hill, "Data Page Layouts for Relational Databases on Deep Memory Hierarchies", International Very Large Data Bases (VLDB) Journal, 2002.

[AGP02] M. Abdallah, R. Guerraoui, P. Pucheral, "Dictatorial Transaction Processing: Atomic Commitment Without Veto Right", Distributed and Parallel Databases, 2002.

[AHJ+96] M. Alberda, P. Hartel, E. de Jong, "Using Formal Methods to Cultivate Trust in Smart Card Operating System", Smart Card Research and Advanced Applications Conference (CARDIS), 1996.

[AHK85] A. C. Ammann, M. Hanrahan, R. Krishnamruthy, "Design of a memory resident DBMS", IEEE International Conference Compcon, 1985.

[AHV95] S. Abiteboul, R. Hull, V. Vianu, "Foundations of Databases", Addison-Wesley, 1995.

[All95] C. Allen, "Smart Cards Part of U.S. Effort in Move to Electronic Banking", Smart Card Technology International, Global Journal of Advanced Card Technology, 1995.

[AnB01] J. H. An, M. Bellare, "Does encryption with redundancy provide authenticity?", Advances in Cryptology (EUROCRYPT'01), Lecture Notes in Computer Science, 2001.

[AnK96] R. J. Anderson, M. G. Kuhn, "Tamper Resistance - A Cautionary Note", Computer Laboratory, Cambridge University, Department of Computer Sciences, Purdue University, 1996.

[AnK97] R. J. Anderson, M. G. Kuhn, "Low Cost Attacks on Tamper Resistant Devices", International Workshop on Security Protocols, Lecture Notes in Computer Science, 1997.

[AnP92] A. Analyti, S. Pramanik, "Fast Search in Main Memory Databases", ACM SIGMOD International Conference on Management of Data (SIGMOD), 1992.

[ANS98] ANSI, "Public Key Cryptography For The Financial Services Industry: Triple Data Encryption Algorithm Modes of Operation", X9.52-1998, 1998.

[APE03] APEC Telecommunications and Information Working Group, "Policy and Regulatory Update of Hong Kong China", 2003. url=http://www.apectelwg .org/apecdata/telwg/27tel/plenary/p02.htm

[ARS99] W. Aiello, A. D. Rubin, M. Strauss, "Using smart cards to secure a personalized gambling device", ACM Conference on Computer and Communications Security, 1999.

[Atm02] Atmel Corporation, "Ready-to-use Smart Card Interface ICs for Serial, USB and PCMCIA Links", 2002. url=www.atmel.com/dyn/resources/prod_documents/doc4007b.pdf

[AWH+99] A. Ailamaki, D. DeWitt, M. Hill, D. Wood, "DBMSs on a modern processor: Where does time go", International Conference on Very Large Data Bases (VLDB), 1999.

[Axa] Axalto, Schlumberger subsidiary. url=www.axalto.com

[Axa03] Axalto, "The e-gate advantage for smart card", White Paper, 2003.

[BaT95] D. Batory, J. Thomas, "P2: A Lightweight DBMS Generator", Technical Report, University of Texas at Austin, 1995.

[BBG+98] J. Baulier, P. Bohannon, S. Gogate, S. Joshi, C. Gupta, A. Khivesera, H. F. Korth, P. McIlroy, J. Miller, P. P. S. Narayan, M. Nemeth, R. Rastogi, A. Silberschatz, S. Sudarshan, "DataBlitz: A High Performance Main-Memory Storage Manager", International Conference on Very Large Data Bases (VLDB), 1998.

[BBG+99] J. Baulier, P. Bohannon, S. Gogate, C. Gupta, S. Haldar, S. Joshi, A. Khivesera, H. F. Korth, P. McIlroy, J. Miller, P. P. S. Narayan, M. Nemeth, R. Rastogi, S. Seshadri, A. Silberschatz, S. Sudarshan, M. Wilder, C. Wei, "DataBlitz Storage Manager: Main Memory Database Performance for Critical Applications", ACM SIGMOD International Conference on Management of Data (SIGMOD), 1999.

[BCF01] E. Bertino, S. Castano, E. Ferrari, "Securing XML documents with Author-X", IEEE Internet Computing, 2001.

[BCK96] M. Bellare, R. Canetti, H. Krawczyk, "Keying hash functions for message authentication", Advances in Cryptology (CRYPTO '96), Lecture Notes in Computer Science, 1996.

[BCN+04] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, C. Whelan, "The sorcerer's apprentice guide to fault attacks", Workshop on Fault Detection and Tolerance in Cryptography, 2004.

[BDJ+01] G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, S. Melo de Sousa, "A Formal Executable Semantics of the JavaCard Platform", Lecture Notes in Computer Science, 2001.

[BDL97] D. Boneh, R. A. DeMillo, R. J. Lipton, "On the Importance of Checking Cryptographic Protocols for Faults", Advances in Cryptology (EUROCRYPT'97), Lecture Notes in Computer Science, 1997.

[BDP04] L. Bouganim, F. Dang Ngoc, P. Pucheral, "Client-Based Access Control Management for XML documents", International Conference on Very Large Databases (VLDB), 2004.

[BeN00] M. Bellare, C. Namprempre, "Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm", Advances in Cryptology (ASIACRYPT '00), Lecture Notes in Computer Science, 2000.

[Ber98] S. Berinato, "Smart cards: The intelligent way to security", Network Computing, 1998.

[BFL+96] S. Blythe, , B. Fraboni, S. Lall, H. Ahmed, U. de Riu, "Layout Reconstruction of Complex Silicon Chips", IEEE Journal of Solid-State Circuits, 1993, Kluwer Academic Publishers, 1996.

[BGS00] P. Bonnet, J. Gehrke, P. Seshadri, "Querying the Physical World", IEEE Personal Communications Special Issue on Networking the Physical World, 2000.

[BGS01] P. Bonnet, J. Gehrke, P. Seshadri, "Towards Sensor Database Systems", Mobile Data Management, 2001.

[BHG87] P. A. Bernstein, V. Hadzilacos, N. Goodman, "Concurrency control and recovery in database systems", Addison Wesley, 1987.

[BID+99] A. Bobick, S. Intille, J. Davis, F.Baird, C. Pinhanez, L. Campbell, Y. Ivanov, A. Schtte, A. Wilson, "The KidsRoom: A perceptually-based interactive and immersive story environment", PRESENCE Conference on Teleoperators and Virtual Environments, 1999.

[Big99] P. Biget, "How smarcards can benefit from internet technologies to break memory constraints", Gemplus Developer Conference, 1999.

[BiW98] P. BIGET, J.-J. Vandewalle, "Extended Memory Card", European Multimedia Microprocessor Systems and Electronic Commerce Conference (EMMSEC), 1998.

[BKV98] L. Bouganim, O. Kapitskaia, P. Valduriez, "Memory-Adaptive Scheduling for Large Query Execution", International Conference on Information and Knowledge Management (CIKM), 1998.

[BMK99] P. A. Boncz, S. Manegold, M. L. Kersten, "Database Architecture Optimized for the New Bottleneck: Memory Access", International Conference on Very Large Data Bases (VLDB), 1999.

[BMR01] P. Bohannon, P. Mcllroy, R. Rastogi, "Main-memory index structures with fixed-size partial keys". ACM SIGMOD International Conference on Management of Data (SIGMOD), 2001.

[BoK95] P. A. Boncz, M. L. Kersten, "Monet: An Impressionist Sketch of an Advanced Database System", Basque International Workshop on Information Technology, 1995.

[BoP02] L. Bouganim, P. Pucheral, "Chip-Secured Data Access: Confidential Data on Untrusted Servers", International Conference on Very Large Data Bases (VLDB), 2002.

[BoS00] P. Bonnet, P. Seshadri, "Device Database Systems", International Conference on Data Engineering (ICDE), 2000.

[BRS82] F. Bancilhon, P. Richard, M. Scholl, "On Line Processing of Compacted Relations", International Conference on Very Large Data Bases (VLDB), 1982.

[BSS+03] C. Bolchini, F. Salice, F. Schreiber, L. Tanca, “Logical and Physical Design Issues for Smart Card Databases”, ACM Transactions on Information Systems (TOIS), 2003.

[CaK97] M. J. Carey, D. Kossmann, "On Saying 'Enough Already!' in SQL", ACM SIGMOD International Conference on Management of Data (SIGMOD), 1997.

[Can04] S. Cannady, "Tools to secure your system, privacy, sanity", Trusted Computing Group, White Paper, 2004.

[Car99] L. C. Carrasco, “RDBMS’s for Java Cards ? What a Senseless Idea !”, 1999. url= www.sqlmachine.com

[CC] ISO/IEC FCD 15408-1, "IT Security techniques Evaluation criteria for IT security", Common Criteria.

[CGK+95] B. Chor , O. Goldreich , E. Kushilevitz , M. Sudan, Private information retrieval, Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS'95), p.41, October 23-25, 1995.

[Cha97] S. C. Chan, "An Overview of Smart Card Security", 1997. url=http://www. hkstar.com/~alanchan/papers/smartCardSecurity/

[Cha98] S. Chaudhuri, "An Overview of Query Optimization in Relational Systems", ACM Symposium on Principles of Database Systems (PODS), 1998.

[Che00] Z. Chen, "Java Card Technology for Smart Cards : Architecture and Programmer's guide", Addison-Wesley, 2000.

[ChW00] S. Chaudhuri, G. Weikum, "Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System", International Conference on Very Large Data Bases (VLDB), 2000.

[CKJ+98] B. Calder, C. Krintz, S. John, T. Austin, "Cache-conscious data placement", International Conference on Architectural Support for Programming Languages and Operating Systems, October 1998.

[CoK85] G. P. Copeland, S. Khoshafian, "A Decomposition Storage Model", ACM SIGMOD International Conference on Management of Data (SIGMOD), 1985.

[CRn04] CR80 news, "New report points way for profitable smart card IC manufacturing", Newsletter, 2004.

[DDJ+03] E. Damiani, S. De Capitani Vimercati, S. Jajodia, S. Paraboschi, P. Samarati, "Balancing Confidentiality and Efficiency in Untrusted Relational DBMSs", ACM Conference on Computer and Communications Security (CCS), 2003.

[DDP+02] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, P. Samarati, "A Fine-Grained Access Control System for XML Documents", ACM TISSEC, 2002.

[Den00] D. E. Denning, "Information Warfare and Security", Addison-Wesley, 2000.

[Dep01] Department of veterans affairs, "G-8 Healthcare Data Card Project". Retrieved in 2001. url=http://www1.va.gov/card/

[DGG+03] D. Deville, A. Galland, G. Grimaud, S. Jean, "Assessing the Future of Smart Card Operating Systems", Conference E-SMART, 2003.

[DhF01] J. F. Dhem, N. Feyt, "Hardware and Software Symbiosis Helps Smart Card Evolution", IEEE Micro, 2001.

[DHo04] I. Duthie, B. Hochfield, "Secure Megabytes on your Smart Card", Card Forum International, 2004

[Did04] Digital ID World, "Assuring Networked Data and Application Reliability", Press Release, 2004.

[Dip97] B. Dipert, "FRAM: Ready to ditch niche ?", EDN Access Magazine, Cahners Publishing Company, 1997.

[Dis04] Discretix corporation, "High Capacity SIM Market Overview", 2004. url=http://www.discretix.com/market_hcsim.shtml

[DKO+84] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, D. A. Wood, "Implementation Techniques for Main Memory Database Systems", ACM SIGMOD International Conference on Management of Data (SIGMOD), 1984.

[Eeu] eEurope Smart Cards. url=http://www.eeurope-smartcards.org/

[ElN94] E. Elmasri, S. B. Navathe, "Fundamentals of Database Sytems", Benjamin Cummings, Second Edition, 1994.

[Emp00] Empress Software Inc, "Empress V8.60 Manual Set", 2000. url=www.empress.com

[Emp03] Empress Software Inc, "Empress Product profile", 2003. url=www.empress.com

[ETS99] European Telecommunications Standards Institute (ETSI), "Digital cellular telecommunications system (Phase 2+), Specification of the Subscriber Identity Module – Mobile Equipment (SIM–ME) interface, GSM 11.11 version 8.1.0", 1999.

[FNP+79] R. Fagin, J. Nievergelt, N. Pippenger and H. R. Strong, "Extendible Hashing : A fast access method for dynamic files", ACM Transactions on Database Systems (TODS), 1979.

[FrS04] Frost & Sullivan, "Battle of Platforms", 2004. url=www.smart cards.frost.com

[FTC02] Federal Trade Commission (U.S.), "Freedom of information and privacy act handbook", 2002.

[Fuj02] Fujitsu Microelectronics Europe, "FRAM For Smart Cards", Fact Sheet MB89R076, 2002.

[Fuj03] Fujitsu, "Security Design of Smart Cards and Secure Devices with Embedded FRAM", Vol.21, N°3, 2003.

[GaB01] A. Gabillon and E. Bruno, “Regulating access to XML documents. IFIP Working Conference on Database and Application Security, 2001.

[Gar03] Gartner Dataquest survey, 2003. url=www.cardtechnology.com

[GaS92] H. Garcia-Molina, K. Salem, "Main memory database systems: An overview", IEEE Transactions on Knowledge and Data Engineering (TKDE), 1992.

[GDM98] R. Gupta, S. Dey, P. Marwedel, "Embedded System Design and Validation: Building Systems from IC cores to Chips", International Conference on VLSI Design, 1998.

[Gem] SUMO Product, A 224MB microprocessor Smart Card. url=http://www.gemplus.com/smart/enews/st3/sumo.html

[Gem04b] Gemplus SA, "32-bit processors for smart cards", Retrieved in 2004. url=http://www.gemplus.com/smart/enews/st3/32bit.html

[Gem99] Gemplus, "SIM Cards: From Kilobytes to Megabytes", 1999. url=www.gemplus.fr/about/pressroom/

[Gig01] E. Giguère, "Mobile Data Management: Challenges of Wireless and Offline Data Access", International Conference on Data Engineering (ICDE), 2001.

[GiL99] P. Girard, J.-L. Lanet, "Java Card or How to Cope with the New Security Issues Raised by Open Cards?", Gemplus Developer Conference, Paris, France, June 1999.

[Gir99] P. Girard, "Which security policy for multi-application smart cards?", USENIX workshop on Smart Card Technology (Smart card'99), 1999.

[GKM+04] M. Gertz, A. Kwong, C. Martel, G. Nuckolls, P. Devanbu, S. Stubblebine, "Databases that tell the Truth: Authentic Data Publication", Bulletin of the Technical Committee on Data Engineering, 2004.

[Glo] Global Platform association for smart card. url=http://www.globalplatform.org/

[Gor00] S. Gordon, "Gemplus builds USB into smart card for direct connection", Press Release, article 18304778 EE Times, 2000.

[Gra90] G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System", ACM SIGMOD International Conference on Management of Data (SIGMOD), 1990.

[Gra93] G. Graefe, "Query Evaluation Techniques for Large Databases", ACM Computing Surveys, 1993.

[Gra98] G. Graefe, "The New Database Imperatives", International Conference on Data Engineering (ICDE), 1998.

[GSM] ETSI Standard, "Global System for Mobile communications (GSM) standard for mobile phones", 1990. url=http://www.etsi.org/

[GUW02] H. Garcia-Molina, J. D. Ullman, J.Widom, "Database Systems: The Complete Book", Prentice Hall, 2002.

[GVP+03] P. Ganesan, R. Venugopalan, P. Peddabachagari, A. Dean, F. Mueller, M. Sichitiu, "Encryption Overhead in Embedded Systems and Sensor Network

Nodes: Modeling and Analysis", International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES), 2003.

[Hen01] M. Henry, "Smart Card Security And Applications", Hartech House, Second Edition, 2001.

[Hey94] M. Heytens, S. Listgarten, M. Neimat, K. Wilkinson. "Smallbase: A Main-Memory DBMS for High-Performance Applications", Technical Report, HP Laboratories, 1994.

[HIL+02] H. Hacigumus, B. Iyer, C. Li, S. Mehrotra, "Executing SQL over Encrypted Data in the Database-Service-Provider Model", ACM SIGMOD International Conference on Management of Data (SIGMOD), 2002.

[Hit02] Hitachi Ltd. Semiconductor & Integrated Circuits, "AE45C (HD65145C) Version 01. Smart card Security Target", White Paper, 2002.

[Hol04] Bill Holcombe, U.S. General Service Administration (GSA), "Government Smart Card Handbook", 2004.

[HWH01] N. J. Henderson, N. M. White, P. H. Hartel, "iButton Enrolment and Verification Requirments for the Pressure Sequence Smart card Biometric", E-Smart Conference, 2001.

[IBM] IBM corporation, "Thinkpad TCPA Option". url=http://www.pc.ibm.com/us/security/index.html

[IBM98] IBM corporation, "Smart Cards: A Case Study", IBM Red Book, 1998.

[IBM99] IBM Corporation, "DB2 Everywhere – Administration and Application Programming Guide", IBM Software Documentation SC26-9675-00, 1999.

[Ibut00] "Ibutton: Wearable Java Key Unlocks Doors and PCs", 2000. url=http://www.allnetdevices.com/wireless/news/2000/11/03/wearable_java.html

[IGE00] C. Intanagonwiwat, R. Govindan, D. Estrin, "Directed diffusion: A scalable and robust communication paradigm for sensor networks", International Conference on Mobile Computing and Networking (MobiCOM), 2000.

[IMM+04] B. Iyer, S. Mehrotra, E. Mykletun, G. Tsudik, Y. Wu, "A Framework for Efficient Storage Security in RDBMS", International Conference on Extending Database Technology (EDBT), 2004.

[ImN02] T. Imielinski, B. Nath, "Wireless Graffiti - Data, data everywhere", International Conference on Very Large Data Bases (VLDB), 2002.

[Inf04] Infineon Technologies, "Security & Chip Card ICs SLE 88CFX4002P", Preliminary Short Product Information, 2004.

[IoK90] Y. Ioannidis, Y. Kang, "Randomized Algorithms for Optimizing Large Join Queries", ACM International Conference on Management of Data (SIGMOD), 1990.

[Ion96] Y. Ioannidis, "Query Optimization", Handbook for Computer Science, Ch. 45, pp 1038-1057, CRC Press, Boca Raton, FL, 1996.

[ISO14] International Standardization Organization (ISO), "Identification cards – Contactless Integrated Circuit(s) Cards - proximity Cards", ISO/IEC 14443.

[ISOp1] International Standardization Organization (ISO), "Integrated Circuit(s) Cards with Contacts – Part 1: Physical Characteristics", ISO/IEC 7816-1, 1998.

[ISOp2] International Standardization Organization (ISO), "Integrated Circuit(s) Cards with Contacts - Part 2: Dimensions and Location of the Contacts", ISO/IEC 7816-2, 1999.

[ISOp4] [ISOp4] International Organization for Standardization (ISO), "Integrated circuit(s) cards with contacts – Part 4: Interindustry commands for interchange", ISO/IEC 7816-4, 1995.

[ISOp7] International Standardization Organization (ISO), "Integrated Circuit(s) Cards with Contacts - Part 7: Interindustry Commands for Structured Card Query Language-SCQL", ISO/IEC 7816-7, 1999.

[Iwa03] T. Iwata, "A survey of CBC MAC and variants", Tokyo Workshop, 2003. url=http://crypt.cis.ibaraki.ac.jp/omac/talks/20030603.pdf

[JaK84] M. Jarke, J. Koch, "Query Optimisation in Database Systems", ACM Computing survey, 1984.

[Jea01] S. Jean, "Modèles et Architectures d'Interaction Interne et Externe pour Cartes à Microprocesseur Ouvertes", Université des Sciences et Techniques de Lille, Ph. D. Thesis, 2001.

[JLR+94] H. Jagadish, D. Lieuwen, R. Rastogi, A. Silberschatz, S. Sudarshan, "Dali: A High Performance Main Memory Storage Manager", International Conference on Very Large Data Bases (VLDB), 1994.

[Joi02] Joint Interpretation Library, "Guidance for Smart card evaluation", White Paper, 2002.

[Jun03] W. J. Jun, Giesecke & Devrient, "Smart Card Technology Capabilities", 2003. url=csrc.nist.gov/publications/nistir/IR-7056/Capabilities/Jun-SmartCardTech.pdf

[Jut01] C. Jutla, "Encryption modes with almost free message integrity", Advances in Cryptology (EUROCRYPT'01), Lecture Notes in Computer Science, 2001.

[KaY00] J. Katz, M. Yung, "Unforgeable Encryption and Adaptively Secure Modes of Operation", Fast Software Encryption '00, Lecture Notes in Computer Science, 2000.

[KBC97] H. Krawczyk, M. Bellare, R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC2104, 1997.

[Kek03] M. Kekicheff, "GlobalPlatform: implementations in the financial, government & telecom industries", CARTES Conference, 2003.

[Key01] Keycorp Limited, "MULTOS smart card on 16K, 32K or 64K", Technical Specifications, 2001. url=www.keycorp.net

[Kim04] W. Kim, "Smart Cards: Status, Issues, US Adoption". Journal of Object Technology, 2004.

[KKP] J. M. Kahn, R. H. Katz, K. S. J. Pister, "Next Century Challenges: Mobile Networking for 'Smart Dust'", Department of Electrical Engineering and Computer Sciences, University of California, Berkeley. url= http://robotics. eecs.berkeley.edu/~pister/SmartDust

[KLL+01] J. Karlsson, A. Lal, C. Leung, T. Pham, "IBM DB2 Everyplace: A Small Footprint Relational Database System", International Conference on Data Engineering (ICDE), 2001.

[KMS95] P. Karn, P. Metzger, W. Simpson, "The ESP Triple DES Transform", RFC1851, 1995.

[KOA+99] C. D. Kidd, R. Orr, G. D. Abowd, C. G. Atkeson, I. A. Essa, B. MacIntyre, E. D. Mynatt, T. Starner, W. Newstetter, "The Aware Home: A living laboratory for ubiquitous computing research", International Workshop on Cooperative Buildings (CoBuild), 1999.

[KoK99] O. Kommerling, M. G. Kuhn, "Design Principles for Tamper-Resistant Smart card Processors", USENIX Workshop on Smart card Technology (Smart card '99), 1999.

[KoS91] H. Korth, A. Silberschatz, "Database Systems Concepts", McGraw-Hill, 1991.

[KRS+03] M. Kallahalla, E. Riedely, R. Swaminathan, Q. Wangz, K. Fux, "Plutus: Scalable secure file sharing on untrusted storage", USENIX Conference on File and Storage Technologies (FAST'03), 2003.

[LaR98] J.-L. Lanet, A. Requet, "Formal proof of smart card applets correctness", International Conference CARDIS'98, 1998.

[LeC86a] T. J. Lehman, M. J. Carey, "A Study of Index Structures for Main Memory Database Management Systems", International Conference on Very Large Data Bases (VLDB), 1986.

[LeC86b] T. J. Lehman, M. J. Carey, "Query Processing in Main Memory Database Systems", ACM SIGMOD International Conference on Management of Data (SIGMOD), 1986.

[Lec97] S. Lecomte, P. Trane, “Failure Recovery Using Action Log for Smart cards Transaction Based System”, IEEE On Line Testing Workshop, 1997.

[Ler01] X. Leroy, "On-card bytecode verification for java card", Lecture Notes in Computer Science, 2001.

[LiN96] S. Listgarten, M.-A. Neimat, "Modelling Costs for a MM-DBMS", International Workshop on Real-Time Databases: Issues and Applications (RTDB), 1996.

[LiN97] S. Listgarten, M.-A. Neimat, "Cost Model Development for a Main Memory Database System", Real-Time Database Systems: Issues and Applications (RTDB), 1997.

[Lit80] W. Litwin, "Linear Hashing : A New Tool For File and Table Addressing", International Conference on Very Large Data Bases (VLDB), 1980.

[MaB00] H. Martin, L. du Bousquet, "Automatic test generation for Java-Card applets", Java Card Workshop 2000.

[MaF02] Madden S., Franklin M., “Fjording the Stream: An Architecture for Queries over Streaming Sensor Data”, Int. Conf. On Data Engineering (ICDE), 2002.

[MaG98a] C. Madson, R. Glenn, "The Use of HMAC-MD5-96 within ESP and AH", RFC2403, 1998.

[MaG98b] C. Madson, R. Glenn, "The Use of HMAC-SHA-1-96 within ESP and AH", RFC2404, 1998.

[MaH02] S. Madden, J. M. Hellerstein, "Distributing Queries over Low-Power Wireless Sensor Networks", ACM SIGMOD International Conference on Management of Data (SIGMOD), 2002.

[Mal01] D. L. Maloney, "Card Technology in Healthcare", CardTech/SecurTech, 2001.

[Mao] Maosco consortium LTD, "MultosTM : The multi-application operating system for smart cards". url=http://www.multos.com

[Mao04] Maosco Limited, "MULTOS Product Directory, Part 3 : Card Manufacturers", 2004.

[Mar03] D. Marsh, "Smart Cards Pass the Global Test", EDN magazine, March 2003.

[MaS02] D. Mazieres, D. Shasha, "Building secure file systems out of Byzantine storage", ACM ACM Symposium on Principles Of Distributed Computing (PODC'02), 2002.

[Mas03] Mastercard Inc, "MasterCard Open Data Storage", Retreived in 2003. url=https://hsm2stl101.mastercard.net/public/login/ebusiness/smart_cards/one_smart_card/biz_opportunity/mods/index.jsp?hsmWRredir=0

[MBK00a] S. Manegold, P. A. Boncz, M. L. Kersten, "Optimizing Database Architecture for the New Bottleneck: Memory Access", In the International Very Large Data Bases (VLDB) Journal, 2000.

[MCS88] M. V. Mannino, P. Chu, T. Sager, "Query Evaluation Techniques for Large Databases", ACM Computing Surveys, 1988.

[MEL] Multos Executable Langage (MEL), Multos Developers Guide, url=www.multos.gr.jp/library/pdf/dl004_developers_guide.pdf

[MFH+02] S. Madden, M. J. Franklin, J. Hellerstein, W. Hong, "TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks", International Conference on Operating Systems Design and Implementation (OSDI), 2002.

[MFH+03] S. Madden, M. Franklin, J. Hellerstein, W. Hong, "The Design of an Acquisitional Query Processor for Sensor Networks", ACM International Conference on Management of Data (SIGMOD), 2003.

[MHH04] S. Madden, J. M. Hellerstein, W.Hong, "TinyDB: In-Network Query Processing in TinyOS", Tutorial, International Conference on Data Engineering (ICDE), 2004.

[Mic02] Microsoft Corporation, "Microsoft Palladium: A Business Overview", Retreived in 2002. url=http://www.microsoft.com/presspass/features/2002/jul02/0724palladiumwp.asp

[Mic04] Microsoft Corporation, "Microsoft Next-Generation Secure Computing Base", Retrieved in 2004, url=http://www.microsoft.com/ressources/ngscb

[MIP02] MIPS Technologies Inc, "Smart Cards: the computer in your wallet", White Paper, 2002. url=http://www.mips.com/content/PressRoom/TechLibrary /techlibrary

[Mis82] M. Missikoff, "A Domain Based Internal Schema for Relational Database Machines", ACM SIGMOD International Conference on Management of Data (SIGMOD), 1982.

[MiS83] M. Missikoff, M. Scholl, "Relational Queries in a Domain Based DBMS", ACM SIGMOD International Conference on Management of Data (SIGMOD), 1983.

[MMC+02] S.W. Moore, R.D. Mullins, P.A. Cunnigham, R. Anderson, G. Taylor, "Improving Smart Card Security using Self-Timed Circuits", ASYNC'02, 2002.

[MoK99] M. Montgomery, K. Krishna, "Secure Object Sharing in Java Card", USENIX workshop on Smart Card Technology (Smart card'99), 1999.

[Moo96] S.W. Moore, "Multithreaded Processor Design", Kluwer Academic Publishers, 1996.

[MOP04] MOPASS Consortium. Retrieved in september 2004. url=www.mopass.info

[MOV97] A. Menezes, P. van Oorschot, S. Vanstone, "Handbook of Applied Cryptography", CRC Press, 1997. url=www.cacr.math.uwaterloo.ca/hac.

[MTS+04] S. Moriyama, H. Tanabe, S. Sasaki, S. Toyomura, "Traceability and Identification Solutions for Secure and Comfortable Society", Hitachi Review, 2004.

[MuD02] C. Muller, E. Deschamps, "Smart cards as first-class network citizens", Gemplus Developer Conference, 2002.

[MVS00] U. Maheshwari, R. Vingralek, W. Shapiro, "How to Build a Trusted Database System on Untrusted Storage", International Conference on Operating Systems Design and Implementation (OSDI), 2001.

[Nes01] NESSIE, "Report on the Performance Evaluation of NESSIE Candidates I", Public Report, November 2001.

[Nes03] NESSIE, "Performance of Optimized Implementations of the NESSIE Primitives", Public Report, 2003.

[NET] Netc@rd project. url=http://www.netcards-project.com/new/

[Nik03] Nikkei Electronics. "Using FRAM in Contactless Smart card IC", 2003. http://neasia.nikkeibp.com/nea/200301/alert_223818.html

[NIS01] NIST, "Advanced Encryption Standard (AES)", FIPS Publication 197, 2001.

[NIS02] NIST, "U.S. Government Smart Card Interoperability Specification(GSC-IS, v2.0)", Internal Report 6887, 2002.

[NIS77] NIST, "Data Encryption Standard (DES)", FIPS Publication 46, 1977.

[NIS88] NIST, "Data Encryption Standard (DES)", FIPS Publication 46-2, 1988.

[NIS93] NIST, "Data Encryption Standard (DES)", FIPS Publication 46-3, 1993.

[NIS95] NIST, "Secure hash standard", FIPS Publication 180-1, 1995.

[NoT04] No-TCPA, Retrieved in 2004. url=http://www.notcpa.org/

[NPS+03] M. Neve, E. Peeters, D. Samyde, J.-J. Quisquater, "Memories: a Survey of their Secure Uses in Smart Cards". IEEE International Security in Storage Workshop, 2003.

[NRC01] NRC report, "Embedded Everywhere, A research agenda for networked systems of embedded computers", National Academy Press, 2001.

[OBS99] M. A. Olson, K. Bostic, M. I. Seltzer, "Berkeley DB", USENIX Technical Conference, 1999.

[OCR02] OCR HIPAA Privacy, "General Overview of Standards for Privacy of Individually Identifiable Health Information", 2003

[OHT03] "Omaha Hospitals To Accept Patient Smart Card", Press Release, 2003. url=www.cardtechnology.com

[Ols00] M. A. Olson, "Selecting and Implementing an Embedded Database System," IEEE Computer Magazine, 2000.

[Ora02a] Oracle Corporation, "Oracle 9i lite: Release Notes - Release 5.0.1", Oracle Documentation, 2002.

[Ora02b] Oracle Corporation, "Oracle 9i Lite - Oracle Lite SQL Reference", Oracle Documentation, 2002.

[Ort00] S. Ortiz Jr., "Embedded Databases Come out of Hiding," IEEE Computer Magazine, 2000.

[OSC02] Open Smart Card Infrastructure for Europe, "eESC Common Specifications for interoperable multi-application secure smart cards v2.0. Volume and part structure", Technical Specifications, 2002.

[PBV+01] P. Pucheral, L. Bouganim, P. Valduriez, C. Bobineau, "PicoDBMS: Scaling down Database Techniques for the Smart card". Very Large Data Bases (VLDB) Journal, 2001.

[Per02a] E. Perkins, "META Report: Smart Moves With Smart Cards", Press Release, Information Network, 2002.

[Per02b] Pervasive Software Inc, "Pervasive.SQL v8", 2002. url=www.pervasive.com.

[PfL97] C. P. Pfleeger, S. Lawrence, "Security In Computing", Prentice Hall, Second Edition, 1997.

[Phi04] Philips Semiconductors, "Clockless technology enables new applications", 2004. url=www.semiconductors.philips.com/markets/identification/articles /articles/a57/

[Phi97] Koninklijke Philips Electronics, "Hardware Security of Advanced Smart Card ICs", White paper, 1997. url=http://www.semiconductors.philips.com/markets/ identification/articles/articles/a7/

[PKK+98] J. H. Park, Y. S. Kwon, K. H. Kim, S. Lee, B. D. Park, S. K. Cha, "Xmas: An Extensible Main-Memory Storage System for High-Performance Applications", ACM SIGMOD International Conference on Management of Data (SIGMOD), 1998.

[Pot02] O. Potonniée, "Ubiquitous Personalization: a Smart Card Based Approach", Gemplus Developer Conference, 2002.

[Pot04] O. Potonniée, "A decentralized privacy-enabling TV personalization framework", European Conference on Interactive Television: Enhancing the Experience (euroITV), 2004.

[PRB98] B. Preneel, V. Rijmen, A. Bosselaers, "Principles and Performance of Cryptographic Algorithms", Dr. Dobb's Journal, 1998.

[PRR+03] N. Potlapally, S. Ravi, A. Raghunathan, N. K. Jha, "Analyzing the Energy Consumption of Security Protocols", IEEE International Symposium on Low-Power Electronics and Design (ISLPED), 2003

[PTV90] P. Pucheral, J.-M. Thevenin, P. Valduriez, "Efficient main memory data management using the DBGraph storage model", International Conference on Very Large Databases (VLDB), 1990.

[RaG02] R. Ramakrishnan, J. Gehrke, "Database Management Systems", McGraw-Hill, Third Edition, 2002.

[RaR00] J. Rao, K. A. Ross, "Making B+-trees cache conscious in main memory", ACM SIGMOD International Conference on Management of Data (SIGMOD), 2000.

[RaR99] J. Rao and K. A. Ross, "Cache Conscious Indexing for Decision-Support in Main Memory", International Conference on Very Large Data Bases (VLDB), 1999.

[RBB+01] P. Rogaway, M. Bellare, J. Black, T. Krovetz, "OCB: A block-cipher mode of operation for efficient authenticated encryption". ACM Conference on Computer and Communications Security (CCS), 2001.

[ReB00] A. Requet, G. Bossu, "Embedding Formally Proved Code in a Smart Card : Converting B to C", IEEE International Conference on Formal Engineering Methods (ICFEM), 2000.

[Ren03] Renesas Technology, "Renesas Technology Develops AE-5 Series of 32-Bit Smart Card Microcontrollers and Releases AE57C with Large-Capacity On-Chip Memory as The First Product", Product announcement, 2003.

[Ren04] Renesas Technology, "First in the industry to combine true smart card functions in FLASH memory cards", 2004.

[RES03] Roadmap for European research on Smart card rElated Technologies (RESET), "Final Roadmap", Project Deliverable, 2003.

[Rij] Rijndael web page. url=http://www.esat.kuleuven.ac.be/~rijmen/rijndael/.

[SAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, T. G. Price. Access Path Selection in a Relational Database Management System. ACM SIGMOD International Conference on Management of Data (SIGMOD), 1979.

[Sac87] G. M. Sacco, "Index access with a finite buffer", International Conference on Very Large Data Bases (VLDB), 1987.

[ScD89] D. Schneider, D. DeWitt, "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment", ACM SIGMOD International Conference on Management of Data (SIGMOD), 1989.

[ScD90] D. Schneider, D. DeWitt, "Tradeoffs in processing complex join queries via hashing in multiprocessor database machines", International Conference on Very Large Data Bases (VLDB), 1990.

[Sch] F. Schutz, "Cryptographie et Sécurité", Cours. url=http://cui.unige.ch/tcs/cours/crypto

[Sch96] B. Schneier, "Applied Cryptography", Second Edition, John Wiley & Sons, 1996.

[ScS99] B. Schneier, A. Shostack, "Breaking up is hard to do: Modeling Security Threats for Smart Cards", USENIX Symposium on Smart Cards, 1999.

[SeG01] P. Seshadri, P. Garrett, "SQLServer for Windows CE - A Database Engine for Mobile and Embedded Platforms", International Conference on Data Engineering (ICDE), 2000.

[Sel99] M. I. Seltzer, "Gizmo databases", USENIX Conference, 1999.

[SeO99] M. I. Seltzer and M. Olson, "Challenges in embedded database system administration", Embedded System Workshop, 1999.

[Ses00] P. Seshadri, P. Garrett, "SQLServer For Windows CE – A Database Engine for Mobile and Embedded Platforms", International Conference on Data Engineering (ICDE), 2000.

[Ses99] P. Seshadri, "Honey, I shrunk the database". ACM SIGMOD International Conference on Management of Data (SIGMOD), 1999.

[Sha02a] B. Sharpe, Appliance Studio Ltd, "Trends in Information Technology", 2002.

[Sha02b] Sharp Corporation, "Micro Computer Chip For Combination Type Smart Card, Model No.: RK410XXXX", Technical Specifications, 2002.

[She03] M. M. Shen, "The Promise of Future Technologies", NY Conference and exposition, 2003.

[She99] K. Shelfer, "The Intersection of Knowledge Management and Competitive Intelligence: Smart Cards and Electronic Commerce. Knowledge Management For The Information Professional", Information Today, Inc. Medford, 1999.

[SHS01] C. A. Stein, J. H. Howard, M. I. Seltzer, "Unifying File System Protection", USENIX Annual Technical Conference, 2001.

[ShV02] W. Shapiro, R. Vingralek, "How to Manage Persistent State in DRM Systems", Data Right Management Systems Workshop, 2001.

[SIA02] Semiconductors Industrial Association (SIA), "STATS: SIA Annual Databook", 2002. http://www.semichips.org

[SIN] SINCE project. url=http://www.eurosmart.com/since/index.htm

[SIN02a] Secure and Interoperable Networking for Contactless in Europe (SINCE), "Interoperability", Project Deliverable, 2002.

[SIN02b] Secure and Interoperable Networking for Contactless in Europe (SINCE), "Interoperable European Electronic ID / Public Service Cards", Project Deliverable, 2002.

[SIN03c] Secure and Interoperable Networking for Contactless in Europe (SINCE), "Open Smart Card Infrastructure for Europe volume 6, Part 4, Contactless Technology: Certification", Project Deliverable, 2003.

[SIN04a] Secure and Interoperable Networking for Contactless in Europe (SINCE), "Open Smart Card Infrastructure for Europe volume 6, Part 3, Contactless Technology: Survey", Project Deliverable, 2004.

[SIN04b] Secure and Interoperable Networking for Contactless in Europe (SINCE), "Open Smart Card Infrastructure for Europe volume 6, Part 2, Contactless Technology: Security and Threat Evaluation relating to Contactless Cards", Project Deliverable, 2004

[SKW92] V. Singhal, S. Kakkad, P. Wilson, "Texas: An Efficient, Portable Persistent Store", International Workshop on Persistent Object Systems, 1992.

[Sma] Smart Card Alliance. url= http://www.smartcardalliance.org/

[Sma03a] Smart Card Alliance, "Easy Access and HIPAA Compliance for Health Care Organizations According to New Smart Card Alliance", White Paper, 2003.

[Sma03b] Smart Card Alliance, "Privacy and Secure Identification Systems: The Role of Smart Cards as a Privacy-Enabling Technology", White Paper, 2003.

[Sma03c] Smart Card Alliance, "HIPAA Compliance and Smart Cards: Solutions to Privacy and Security Requirements", White Paper, 2003.

[Spe03] R. Spencer, Reuters Business Insight Finance, "Key Issues in Plastic Cards: Technology comes of age", 2003.

[Sti00] D. Stinson, "Cryptography: Theory and Practice", Second Edition, 2000.

[STm02] STMicroelectronics Corporation, "STMicroelectronics Debuts World's Most Advanced Smart Card Memory Technology, New 'Page Flash' technology will benefit Mobile Access Networks and Mobile Multimedia Services", 2002. url=www.st.com/stonline/press/news/year2002/p1249m.htm

[STM04] STMicroelectronics Corporation, "ST22FJ1M: 1MB Flash Smart card IC", Technical Specifications, 2004.

[Sun00a] Sun Microsystems, "Java Card 2.1.1 API Specification", Technical Specifications, 2000.

[Sun00b] Sun Microsystems, "Java Card 2.1.1 Runtime Environment (JCRE) Specification", Technical Specifications, 2000.

[Sun00c] Sun Microsystems, "Java Card 2.1.1 Virtual Machine (JCVM) Specification", Technical Specifications, 2000.

[Sun01] Sun Microsystems, "Java Card Platform Security", White Paper, 2001.

[Syb00] Sybase Inc., "The Next Generation Database for Embedded Systems", White Paper, 2000.

[Syb99] Sybase Inc., "Sybase Adaptive Server Anywhere Reference", Sybase Documentation, 1999.

[SYo93] E. J. Shekita, H. C. Young, "Multi-Join Optimization for Symmetric Multiprocessor", International Conference on Very Large Data Bases (VLDB), 1993.

[SYT93] E. Shekita, H. Young, and K. L. Tan, “Multi-Join Optimization for Symmetric Multiprocessors”, International Conference on Very Large Data Bases (VLDB), 1993.

[Tal99] L. Talvard, "The API Provided by the Smart Cards For Windows", Gemplus Developer Conference, 1999.

[TCG04] Trusted Computing Group, Retrieved in 2004. url=https://www.trusted computinggroup.org/

[TCP04] Trusted Computing Platform Alliance, Retrieved in 2004. url=http://www. trustedcomputing.org/

[Ten99] Times-Ten Team, "In-Memory Data Management for Consumer Transactions The Times-Ten Approach", ACM SIGMOD International Conference on Management of Data (SIGMOD), 1999.

[TPG04b] Trusted Computing Group, "Specification Architecture Overview, Revision 1.2", 2004.

[TrC] A. Tria, H. Choukri, "Invasive Attacks", Gemplus report. url= www.win.tue.nl/ ~henkvt/invasive-attacks.pdf

[Tua99] J.-P. Tual, “MASSC: A Generic Architecture for Multiapplication Smart Cards”, IEEE Micro Journal, 1999.

[USG03] United States General Accounting Office (US-GAO). "Electronic Government: Progress in Promoting Adoption of Smart Card Technology", 2003. url=www.gao.gov/cgi-bin/getrpt?GAO-03-144

[Vac03] F. Vacherand, "New Emerging Technologies for Secure Chips and Smart Cards", MINATEC Conference, 2003.

[Val87] P. Valduriez, "Join Indices", ACM Transactions on Database Systems (TODS), 1987.

[VGD+02] P. Vettiger, G. Gross, M. Despont, U. Drechsler, U During, B. Gotsmann, W. Haberle, M. A. Lantz, H. E. Rothuizen, R. Stutz, G. K. Binning, "The 'Millipede'- Nanotechnology Entering Data Storage", IEEE Transactions on Nanotechnology, 2002.

[Vin02] R. Vingralek, "Gnatdb: A small-footprint, secure database system", International Conference on Very Large Databases (VLDB), 2002

[VMS02] R. Vingralek, U. Maheshwari, W. Shapiro, "TDB: A Database System for Digital Rights Management", International Conference on Extending Database Technology (EDBT), 2002.

[VRK03] H. Vogt, M. Rohs,R. Kilian-Kehr, "Middleware for Communications. Chapter 16: Middleware for Smart Cards", John Wiley and Sons, 2003.

[Web] Webster online dictionary. url=http://www.m-w.com/

[Wei91] M. Weiser, "The Computer for the Twenty-First century", Scientific American, 1991.

[Wei96] M. Weiser, "Open House", Web Magazine of the Interactive Telecommunications Program of New York University, 1996. url=http://www.ubiq.com/hypertext/weiser/wholehouse.doc

[YAE03] H. Yu, D. Agrawal, A. El Abbadi, "Tabular Placement of Relational Data on MEMS-based Storage Devices", International Conference on Very Large Data Bases (VLDB), 2003.

[YaG02] Y. Yao, J. Gehrke, "The Cougar Approach to In-Network Query Processing in Sensor Networks", SIGMOD Record, September 2002.

[Yos03] J. Yoshida, "Smart cards break out of traditional roles as chips advance", Press Release, EE Times, 2003.