PicoDBMS: Scaling down Database Techniques for the Smartcard

Christophe Bobineau[*], Luc Bouganim[*], Philippe Pucheral[*], Patrick Valduriez[**]

[*]PRiSM Laboratory
78035 – Versailles
France
{Firstname.Lastname}@prism.uvsq.fr

[**]INRIA Rocquencourt
78153 – Le Chesnay
France
Patrick.Valduriez@inria.fr

Abstract

Smartcards are the most secure portable computing device today. They have been used successfully in applications involving money, proprietary and personal data (such as banking, healthcare, insurance, etc.). As smartcards get more powerful (with 32 bit CPU and more than 1 MB of stable memory in the next versions) and become multi-application, the need for database management arises. However, smartcards have severe hardware limitations (very slow write, very little RAM, constrained stable memory, no autonomy, etc.) which make traditional database technology irrelevant. The major problem is scaling down database techniques so they perform well under these limitations. In this paper, we give an in-depth analysis of this problem and propose a PicoDBMS solution based on highly compact data structures and query execution without RAM. We show the effectiveness of our techniques through performance evaluation.

1 Introduction

Smartcards are the most secure portable computing device today. The first smartcard was developed by Bull for the French banking system in the 80s to significantly reduce the losses associated with magnetic stripe credit card fraud. Since then, smartcards have been used successfully around the world in various applications involving money, proprietary data and personal data (such as banking, pay-TV or GSM subscriber identification, loyalty, healthcare, insurance, etc.). While today's smartcards handle a single issuer-dependent application, the trend is toward multi-application smartcards. Standards for multi-application support, like the JavaCard [21] and Microsoft's SmartCard for Windows [16], ensure that the card be universally accepted and be able to interact with several service providers. According to DataQuest [8], 990 million smartcards will be shipped in the year 2K and smartcards could become one of the world's highest-volume markets for semiconductors.

As smartcards become more and more versatile, multi-applications and powerful (32 bit processor, more than 1MB of stable storage), the need for database techniques arises. Let us consider a health card storing a complete medical folder including the holder's doctors, blood type, allergies, prescriptions, etc. The volume of data can be important and the queries fairly complex (select, join, aggregate). Sophisticated access rights management using views and aggregate functions are required to preserve the holder's data privacy. Transaction atomicity and durability are also needed to enforce data consistency. More generally, database management helps to separate data management code from application code, thereby simplifying and making application code smaller. Finally, new applications can be envisioned, like computing statistics on a large number of cards, in an asynchronous and distributed way. Supporting database management on the card itself rather than on an external device is the only way to achieve very high security, high availability (anywhere, anytime, on any terminal) and acceptable performance.

However, smartcards have severe hardware limitations which stem from the obvious constraints of small size (to fit on a flexible plastic card and to increase hardware security) and low cost (to be sold in large volumes). Today's microcontrollers contain a CPU, memory including about 96 KB of ROM, 4 KB of RAM and up to 128 KB of stable storage like EEPROM, and security modules [23]. EEPROM is used to store persistent information; it has very fast read time (60-100 ns)

Proceedings of the 26th International Conference on Very Large Databases, Cairo, Egypt, 2000

comparable to RAM but very slow write time (10 ms/word). Following Moore's law for processor and memory capacities, smartcards will get rapidly more powerful. Existing prototypes, like Gemplus's Pinocchio card [10], bypass the current memory bottleneck by connecting an additional chip of 2MB of Flash memory to the microcontroller. Although a significant improvement over today's cards, this is still very restricted compared to other portable, less secure, devices such as Personal Digital Assistants (PDA). Furthermore, smartcards are not autonomous, *i.e.,* have no independent power supply, thereby precluding asynchronous and disconnected processing.

These limitations (tiny RAM, little stable storage, very costly write and lack of autonomy) make traditional database techniques irrelevant. Typically, traditional DBMS exploit significant amounts of RAM and use caching and asynchronous I/Os to reduce disk access overhead as much as possible. With the extreme constraints of the smartcard, the major problem is scaling down database techniques. While there has been much excellent work on scaling up to deal with very large databases, *e.g.,* using parallelism, scaling down has not received much attention by the database research community. However, scaling down in general is getting very important for commodity computing and is quite difficult [12].

Some DBMS designs have addressed the problem of scaling down. Light versions of popular DBMS like Sybase Adaptive Server Anywhere [22], Oracle 8i Lite [18] or DB2 Everywhere [13] have been primarily designed for portable computers and PDA. They have a small footprint which they obtain by simplifying and componentizing the DBMS code. However, they use relatively much RAM and stable memory and do not address the more severe limitations of smartcards. ISOL's SQLJava Machine DBMS [7] is the first attempt towards a smartcard DBMS and SCQL [15], the standard for smartcard database language, emerges. While both designs are limited to single select, they exemplify the strong interest for dedicated smartcard DBMS.

In this paper, we address the problem of scaling down database techniques and propose the design of what we call a PicoDBMS. This work is done in the context of a new project with Bull Smart Cards and Terminals. The design has been made with smartcard applications in mind but its scope extends as well to any ultra-light computer device based on a secured monolithic chip. This paper makes the following contributions:

- We analyze the requirements for a PicoDBMS based on a typical healthcare application and justify its minimal functionality.
- We give an in-depth analysis of the problem by considering the smartcard hardware trends and derive design principles for a PicoDBMS.
- We propose a new pointer-based storage model that integrates data and indices in a unique compact data structure.
- We propose query execution techniques which handle complex query plans (including joins and aggregates) with no RAM consumption.
- We show the effectiveness of each technique through performance evaluation.

This paper is organized as follows. Section 2 illustrates the use of take-away databases in various classes of smartcard applications and presents in more details the requirements of the health card application. Section 3 analyzes the smartcard hardware constraints and gives the problem definition. Sections 4 - 5 present and assess the PicoDBMS' storage model and query execution model, respectively. Section 6 concludes.

2 Smartcard Applications

In this section, we discuss the major classes of emerging smartcard applications and their database requirements. Then, we illustrate these requirements in further details with the health card application, which we will use as reference example in the rest of the paper.

2.1 Database Management Requirements

Table 1 summarizes the database management requirements of the following typical classes of smartcard applications.

Applications	Volume	Select / Project	Join	Group by / Distinct	Access rights views	Atomicity	Durability	Statistics
Money & identification	*tiny*					✔		
Downloadable DB	*high*	✔	✔	✔				
User environment	*medium*	✔			✔	✔	✔	
Personal folder	*high*	✔	✔	✔	✔	✔	✔	✔

Table 1: *Typical application's profiles*

- *Money and identification:* examples of such applications are credit cards, e-purse, SIM for GSM, phone cards, transportation cards. They are representative of today's applications, with very few data (typically the holder's identifier and some status information). Querying is not a concern and access rights are useless since 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 the major concern here. The volume of data can be important and the queries complex. The data are typically read-only and public.

- *User environment:* the objective is to store in a smartcard an extended profile of the card's holder including, among others, data regarding the computing environment (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, the data are highly private and must be protected by sophisticated access rights (*e.g.,* the card's holder may want to share a subset of 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. They roughly share the same requirements, which we illustrate next with the healthcare example. Note that queries involving data issued from different folders can make sense. For instance, one may be interested in discovering associations between some disease and the scholastic level of the card holder. This raises the interesting issue of maintaining statistics on a population of cards or mining their content asynchronously.

2.2 The Health Card Application

The health card is very representative of personal folder applications and has strong database requirements. Several countries (France, Germany, USA, Russia, Korea…) are developing healthcare applications on smartcards [5]. The initial idea was to give to each citizen a smartcard containing her identification and insurance data. As smartcard storage capacity increases, the information stored in the card can be extended to the holder's doctors, emergency data (blood type, allergies, vaccination…), surgical operations, prescriptions, insurance data and even links to heavier data (*e.g.,* X-ray examination, scanner images…) stored on hospital servers. Different users may query, modify and create data in the holder's folder: the doctors who consult the patient's past records and prescribe drugs, the surgeons who perform exams and operations, the pharmacists who deliver drugs, the insurance agents who refund the patient, public organizations which maintain statistics or study the impact of drugs correlation in population samples and finally the holder herself.

We can easily observe that: (i) the amount of data is significant (more in terms of cardinality than in terms of volume because most data can be encoded), (ii) queries can be rather complex (*e.g.,* a doctor asks for the last antibiotics prescribed to the patient), (iii) sophisticated access rights management using views and aggregate functions are highly required (*e.g.,* a statistical organization may access aggregate values only but not the raw data), (iv) atomicity must be preserved (*e.g.,* when the pharmacist delivers drugs) and (v) durability is mandatory, without compromising data privacy (logged data stored outside the card must be protected).

One may wonder whether the holder's health data must be stored in a smartcard or in a centralized database. The benefit of distributing the healthcare database on smartcards is threefold. First, health data must be made highly available (anywhere, anytime, on any terminal and without requiring a network connection). Second, storing sensitive data on a centralized server may hurt privacy. Third, maintaining a centralized database is fairly complex due to the variety of data sources. Assuming the health data is stored in the smartcard, the next question is why the aforementioned database capabilities need be hosted in the smartcard rather than the terminals. The answer is again availability (the data must be exploited on any terminal) and privacy. Regarding privacy, since the data must be confined in the chip, so must be the query engine and the view manager. The smartcard being the unique trusted part of the system, access rights and transaction management cannot be delegated to an untrusted terminal.

3 Problem Formulation

In this section, we make clear the smartcard constraints in order to derive design rules for the PicoDBMS and state the problem. Our analysis is based on the characteristics of both existing smartcard products and current prototypes [23, 10], and thus, should be valid for a while. We also discuss how the main constraints of the smartcard will evolve in a near future.

3.1 SmartCard constraints

Current smartcards include in a monolithic chip, a 32 bits RISC processor at about 30 MIPS, memory modules (of about 96 KB of ROM, 4 KB of static RAM and 128 KB of EEPROM), security components and take their electrical energy from the terminal [23]. ROM is used to store the operating system, fixed data and standard routines. RAM is used as working memory for calculating results. EEPROM is used to store persistent information. EEPROM has very fast read time (60-100 ns/word) comparable to RAM, but a dramatically slow write time (10 ms/word).

The main constraints of current smartcards are therefore: (i) the very limited storage capacity; (ii) the very slow write time in EEPROM; (iii) the extremely reduced size of the RAM; (iv) the lack of autonomy and (v) a high security level that must be preserved in all situations. These constraints strongly distinguish smartcards from any other computing devices, including lightweight computers like PDA.

Let us now consider how hardware advances can impact these constraints, in particular memory size. Current smartcards rely on a well established and slightly out-of-date hardware technology (0.35µm) in order to minimize the production cost (less than five dollars) and increase security [20]. Furthermore, up to now, there was not a real need for large memories in smartcard applications like holder's identification. According to major smartcard providers, the market pressure generated by emerging large storage demanding applications will lead to a rapid increase of the smartcard storage capacity. This evolution is however constrained by the smartcard tiny die size fixed to 25 mm² in the ISO standard [14], which pushes for more integration. This limited size is due to security considerations (to minimize the risk of physical attack [2]) and practical constraints (*e.g.,* the chip should not break when the smartcard is flexed). Another solution to relax the storage limit is to extend the smartcard storage capacity with external memory modules. This is being experienced by Gemplus which recently announced Pinocchio [10], a smartcard equipped with 2 MB 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 or encrypted.

Memory type	EEPROM	FLASH	FeRAM
Read time (/word)	60 to 150 ns	70 to 200 ns	150 to 200 ns
Write time (/word)	10 ms	5 to 10 µs	150 to 200 ns
Erase time (/bank)	None	500 to 800 ms	None
Lifetime (*)	10^5 write cycles per cell	10^5 erase cycles	10^{10} to 10^{12} write cycles

* A memory cell can be overwritten a finite number of times.

Table 2: *Performance of stable memories for the smartcard*

Another important issue is the performance of stable memory. Possible alternatives to the EEPROM are Flash memory and Ferroelectric RAM (FeRAM) [9] (see Table 2 for performance comparisons). Flash is more compact than EEPROM and then represents a good candidate for high capacity smartcards [10]. However, flash banks need be erased before writing, which is extremely slow. This makes Flash memory appropriated for applications with a high read/write ratio (*e.g.,* address books). FeRAM is undoubtedly an interesting option for smartcard 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 issue. Thus FeRAM could be considered a serious alternative only in the very long term [9].

Given these considerations, we assume in this paper a smartcard with a reasonable stable storage area (few MB of EEPROM[1]) and a small RAM area (some KB). Indeed, there is no clear interest to have a large RAM area, given that the smartcard is not autonomous, thus precluding asynchronous write operations. Moreover, more RAM means less EEPROM as the chip size is limited.

3.2 Impact on the PicoDBMS architecture

We now analyze the impact of the smartcard constraints on the PicoDBMS architecture, thus justifying why traditional database techniques, and even lightweight DBMS techniques, are irrelevant. The smartcard's properties and their impact are:

- *Highly secure:* smartcard's hardware security makes it the ideal storage support for private data. The PicoDBMS 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 PicoDBMS code must not present security holes due to the use of sophisticated algorithms[2].
- *Highly portable:* the smartcard is undoubtedly the most portable personal computer (the wallet computer). The data located on the smartcard are thus highly available. They are also highly vulnerable since the smartcard can be lost, stolen or accidentally destroyed. The main consequence is that durability cannot be enforced locally.
- *Limited storage resources:* despite the foreseen increase in storage capacity, the smartcard 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 the memory consumption during execution. In addition, the functionalities of the PicoDBMS must be carefully selected and their implementation must be as light as possible. The lightest the PicoDBMS, the biggest the onboard database.
- *Stable storage is main memory:* smartcard stable memory provides the read speed and direct access granularity of a main memory. Thus, a PicoDBMS can be considered as a *main memory DBMS* (*MMDBMS*). However the dramatic cost of writes distinguishes a PicoDBMS from a traditional MMDBMS. This impacts the storage and access methods of the PicoDBMS as well as the way transaction atomicity is achieved.
- *Non autonomous:* compared to other computers, the smartcard 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).

3.3 Problem Statement

To summarize, our goal is to design a PicoDBMS including the following components:

- Storage manager: manages the storage of the database and the associated indices.
- Query manager: processes query plans composed of select, project, join and aggregates.
- Transaction manager: enforces the ACID properties and participates in distributed transactions.
- Access right manager: provides access rights on base data and on complex user-defined views.

[1] Considering Flash instead of EEPROM will not change our conclusions. It will just exacerbate them.

[2] Most security holes are the results of software bugs [20].

Thus, the PicoDBMS hosted in the chip provides the minimal subset of functionality that is strictly needed to manage in a secure way the data shared by all onboard applications. Other components (*e.g.,* the GUI, a sort operator…) can be hosted in the terminal or be dynamically downloaded when needed, without threatening security. In the rest of this paper, we concentrate on the storage manager and the query manager which are the most impacted by the smarcard constraints. Smartcard-specific transaction manager description can be found in [4], while traditional techniques can be used for access right manager.

When designing the PicoDBMS's components, we must follow several design rules derived from the smartcard's properties:

- *Compactness rule:* minimize the size of data structures and the PicoDBMS code to cope with the limited stable memory area (a few MB).
- *RAM rule:* minimize the RAM usage given its extremely limited size (some KB).
- *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.

4 PicoDBMS storage model

In this section, following the design rules for a PicoDBMS, we discuss the storage issues and propose a very compact model based on a combination of flat storage, domain storage and ring storage. We also evaluate the storage cost of our storage model.

4.1 Flat Storage

The simplest way to organize data is *Flat Storage (FS)*, where tuples are stored sequentially and attribute values are embedded in the tuples. Although it does not impose it, the SCQL standard [15] considers FS as the reference storage model for smartcards. The main advantage of FS is access locality. However, in our context, FS has two main drawbacks:

- *Space consuming:* while normalization rules preclude attributes conjunction redundancy to occur, they do not avoid attribute value duplicates (*e.g.,* the attribute *Doctor.Specialty* may contain many duplicates).
- *Inefficient:* in the absence of index structures, all operations are computed sequentially. While this is convenient for old fashion cards (some KB of storage and a mono-relation select operator), this is no longer acceptable for future cards where storage capacity is likely to exceed 1MB and queries can be rather complex.

Adding index structures to FS may solve the second problem while worsening the first one. Thus, FS alone is not appropriate for a PicoDBMS.

4.2 Domain Storage

Based on the critique of FS, it follows that a PicoDBMS storage model should guarantee both data and index compactness. Let us first deal with data compactness. Since locality is no longer an issue in our context, pointer-based storage models inspired by MMDBMS [1, 17, 19] can help reducing the data storage cost. 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 1, 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[3].

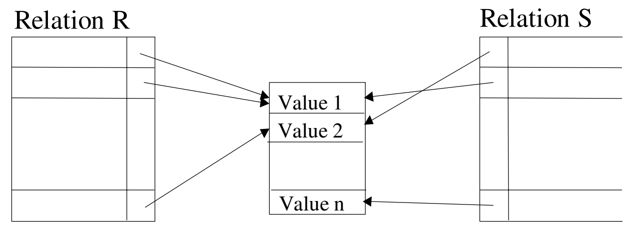


Figure 1 : *Domain Storage*

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 smartcard 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 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).

4.3 Ring Storage

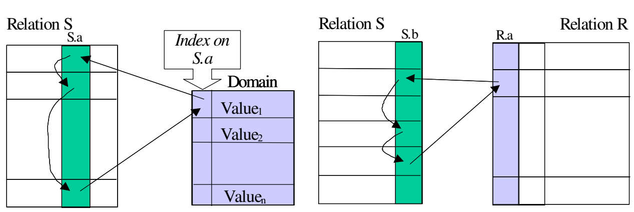
We now address index compactness along with data compactness. Unlike disk-based DBMS that favor indices which preserve access locality, smartcards should make intensive use of secondary (*i.e.,* pointer-based) indices. The issue here is to make these indices as compact as possible. Let us first consider 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

[3] Compression techniques can be advantageously used in conjunction with DS to increase compactness [11].

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. Hence, we call it *Ring index* (see Figure 2(a)). But 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).

Join indices [24] 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. However, most joins are performed on key attributes, *R.a* being a primary key and *S.b* being the foreign key referencing *R.a*. In our model, key attributes are not stored by domain but with FS. Nevertheless, since *R.a* is the primary key of *R*, its extension forms precisely a domain, even if not stored outside of *R*. Since attributes *S.b* take their values in *R.a*'s domain, they reference *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* (*i.e.,* each *S* tuple is linked by a pointer to each *R* tuple matching with it). 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 2(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*. Note that this situation resembles the well-known Codasyl model.



(a) Ring index on a regular attribute *(b) Ring index on a foreign_key attribute*

Figure 2: *Ring Storage*

4.4 Storage cost evaluation

Our storage model combines FS, DS and RS. Thus, the issue is to determine the best storage for each attribute. If the attributes need not be indexed, the choice is obviously between FS and DS. Otherwise, the choice is between RS and FS with a traditional index. Thus, we compare the storage cost for a single attribute, indexed or not, for each alternative. We introduce the following parameters:

CardRel: *cardinality of the relation holding the attribute*
a: *average length of the attribute (expressed in bytes)*
p: *pointer size (3 bytes will be required to address "large" memory of future cards)*
S: *selectivity factor of the attribute.* S=CardDom/CardRel*, where* CardDom *is the cardinality of the attribute domain extension.* S *measures the redundancy of the attribute (*i.e., *the same value appears in* 1/S *tuples).*

Cost(FS) = CardRel*a
Cost(DS) = CardRel*p + S*CardRel*a
Cost(Indexed_FS) = Cost(FS) + S*CardRel*a + CardRel*p
Cost(RS) = Cost(DS) + S*CardRel*p

The cost equality between FS and DS gives: S=(a–p)/a. The cost equality between Indexed_FS and RS gives: S=a/p

Figure 3(a) shows the different values of *S* and *a* for which FS and DS are equivalent. Thus, each curve divides the plan into a gain area for FS (above the curve) and a gain area for DS (under the curve). For values of *a* less than 3 (*i.e.,* the size of a pointer), FS is obviously always more compact than DS. For higher values of *a*, DS becomes rapidly more compact than FS except for high values of *S*. For instance, considering *S=0.5*, that is the same value is shared by only two tuples, DS outperforms FS for all *a* larger than 6 bytes. The higher *a* and the lower *S*, the better DS. The benefit of DS is thus particularly important for enumerated type attributes. Figure 3(b) compares Indexed_FS with RS. The superiority of RS is obvious, except for one and two byte long key attributes. Thus, Figures 3(a) and 3(b) are guidelines for the database designer to decide how to store each attribute, by considering its size and selectivity.

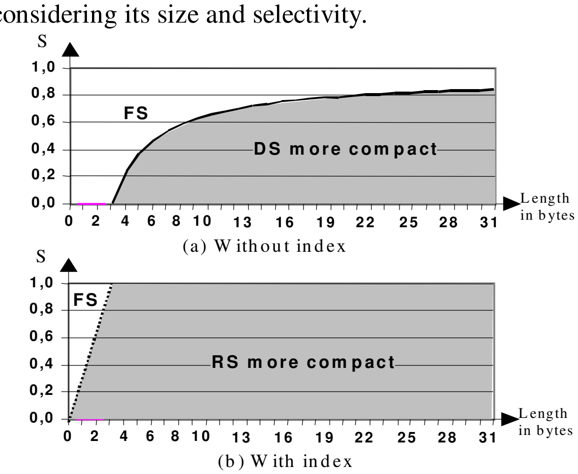


Figure 3: *Storage models tradeoff*

5 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 resort to materialization on disk to avoid memory overflow. These algorithms cannot be used for a PicoDBMS because:

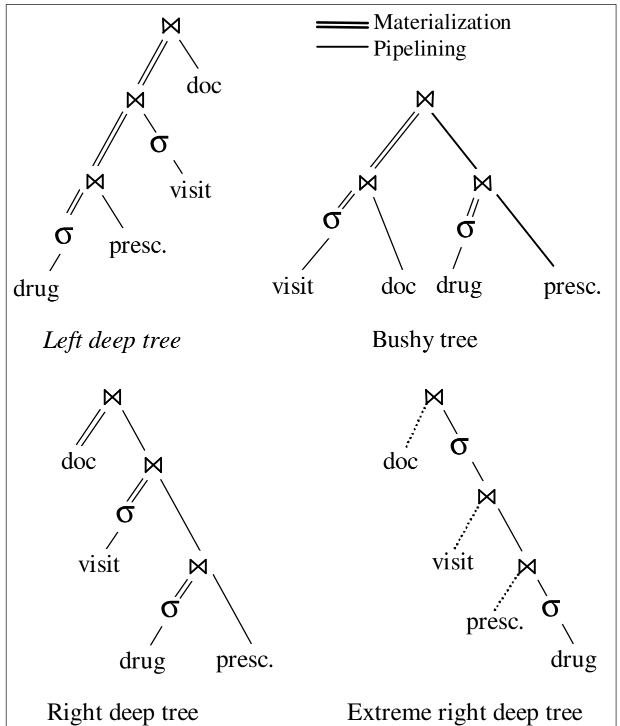
- Given the write rule and the lifetime of stable memory, writes in stable memory are proscribed, even for temporary materialization;
- Dedicating a specific RAM area does not help since we cannot estimate its size a-priori. Choosing it small increases the risk of memory overflow, thereby leading to writes in stable memory. Choosing it large reduces the stable memory area, already limited in a smartcard (RAM rule). Moreover, even a large RAM area cannot guarantee that query execution will not produce memory overflow [3];
- 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. We show the effectiveness of our solution through a performance analysis.

5.1 Basic Query Execution without RAM

We consider 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 [11]. The optimizer can consider different shapes of QEP: *left-deep*, *right-deep* or *bushy trees* (see Figure 4). 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.

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* [11]. 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.



Healthcare Database schema: (sample)	*Doctor (*DocId, name, specialty, …*)* *Prescription (*VisId, DrugId, qty, …*)* *Visit (*VisId, DocId, date, diagnostic, …*)* *Drug (*DrugId, name, type, …*)*

Query Q1 : 'Who prescribes antibiotics in 1999'

Figure 4: *Several execution trees for query Q1*

Let us now detail how select, project and join are performed. These operators can be executed either sequentially or with a ring index. Given the access rule, the use of indices seems always to be the right choice. 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 operators are pushed up to the tree since no materialization occurs. Note that the final project incurs an additional cost in case of ring attributes. Without indices, joining relations is done by a nested-loop algorithm since no other join technique can be applied without ad-hoc structures (*e.g.,* hash tables) and/or working area (*e.g.,* sorting). The cost of indexed 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 cost is proportional to $n*m$

starting with *Doctor* and to *m* starting with *Visit*. Assuming now a bi-directional index, the join 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.

5.2 Complex Query Execution without RAM

We now consider the execution of aggregate, sort and duplicate removal operators. 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 smartcard 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.

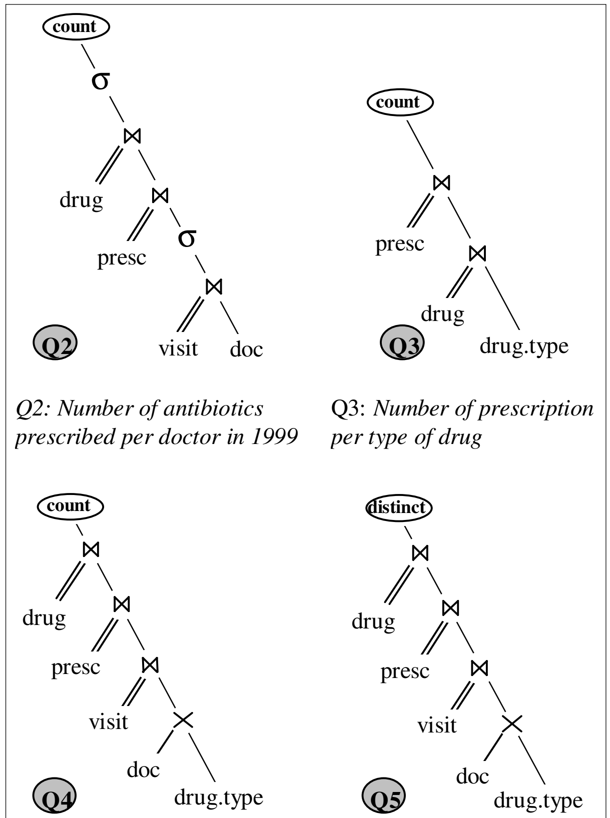
We propose a solution to the above problem by exploiting two properties: (i) aggregate and duplicate removal can be done in pipeline if the incoming tuples are yet grouped by distinct values and (ii) pipeline operators are order-preserving since they consume (and produce) tuples in the arrival order. Thus, enforcing an adequate consumption order at the leaf of the execution tree allows pipelined aggregation and duplicate removal. For instance, the extreme right deep tree of Figure 4 delivers the tuples naturally grouped by *Drug.id*, thus allowing group queries on that attribute.

Let us consider now query Q2 of Figure 5. As pictured, executing Q2 in pipeline requires rearranging the execution tree so that relation *Doctor* is explored first. Since *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 relation *Drug* and domain *drug.type*. Domain values being unique, this join produces the tuples in the adequate order. If domain *Drug.type* does not exist, an operator must be introduced to sort 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*.

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.



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: D*istinct doctors and type of drug*

Figure 5: *Four 'complex' query execution plans*

5.3 Performance Evaluation

Our proposed query engine can handle fairly complex queries, taking advantage of the read and access rules[4] while satisfying the compactness, write, RAM and security rules. We now evaluate whether the PicoDBMS performance matches the smartcard application's requirements, that is any query issued by the application can be performed in reasonable time (*i.e.,* may not exceed the user patience). Since the PicoDBMS code's simplicity is an important consideration to conform to the compactness and security rules, we must also evaluate which acceleration techniques (*i.e.,* ring indices, query optimization) are really mandatory. For instance, an accelerator reducing the response time from 10 ms to 1 ms

[4] With traditional DBMS, such techniques will induce so many disk accesses that the system would thrash!

is useless in the smartcard context[5]. Thus, unlike traditional performance evaluation, our major concern is on absolute rather than relative performance.

Evaluating absolute response time is complex in the smartcard environment because all platform parameters (*e.g.,* processor speed, caching strategy, RAM and EEPROM speed) strongly impact the measurements[6]. Measuring the performance of our PicoDBMS on Bull's smartcard technology is attractive but introduces two problems. First, Bull's smartcards compatible with database applications are still prototypes [23]. Second, we are interested in providing the most general conclusions (*i.e.,* as independent as possible of smartcard architectures). Therefore, we prefer to measure our query engine on two old-fashion computers (a PC 486/25Mhz and a Sun SparcStation 1+) which we felt roughly similar to forthcoming smartcard architectures. For each computer, we vary the system parameters (clock frequency, cache) and perform the experimentation tests. The performance ratios between all configurations were roughly constant, the slowest configuration (Intel 486 with no cache) performing 8 times worse than the fastest (RISC with cache). In the following, we present response times for the slowest architecture to check the viability of our solutions in the worst environment.

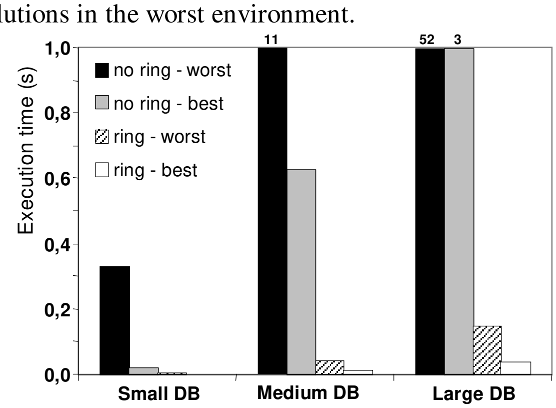


Figure 6: *Performance results for query Q1*

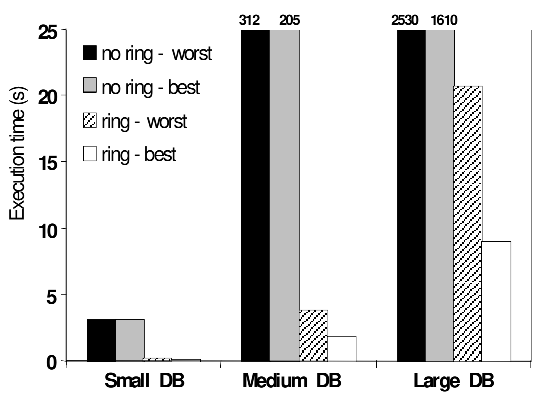


Figure 7: *Performance results for query Q4*

[5] With traditional DBMS, such acceleration can improve the transactional throughput.

[6] With traditional DBMS, very slow disk access allows to ignore finer parameters.

We generated three instances of a simplified healthcare database: the *small, medium and large* databases containing respectively (10, 30, 50) doctors, (100, 500, 1000) visits, (300, 2000, 5000) prescriptions and (40, 120, 200) drugs. Although we tested several queries, we describe below only the two most significant. Query Q1, which contains 3 joins and 2 selects on *Visit* and *Drug* (with selectivities of 20% and 5%) is representative of medium-complexity queries. Query Q4, which performs an aggregate on two attributes and requires the introduction of a Cartesian product, is representative of complex queries. For each query, we measure the performance for all possible query execution plans, varying the storage choices (with and without select and join ring indices). Figures 6 and 7 show the results for both best and worst plans on databases built with or without join indices.

Considering SPJ queries, the PicoDBMS performance clearly matches the application's requirements as soon as join rings are used. Indeed, the performance with join rings is at most 146 ms for the largest database and with the worst execution plan. With small databases, all the acceleration techniques can be discarded, while with larger ones, join rings remain necessary to obtain good response time. In that case, the absolute gain (110 ms) between the best and the worst plan does not justify the use of a query optimizer.

The performance of aggregate queries is clearly worst because they introduce a Cartesian product at the leaf of the execution tree. Join rings are useful for medium and large databases. With large databases, the optimizer turns out to be necessary since the worst execution plan with join rings achieves a rather long response time (20,6 s).

The influence of ring indices for selects (not shown) is insignificant. Depending on the selectivity, it can bring slight improvement or overhead on the results. Although it may achieve an important relative speed-up for the select itself, the absolute gain is not significant considering the small influence of select on the global query execution cost (which is not the case in disk-based DBMS). Select ring indices are however useful for queries with aggregates or duplicate removal, that can result in a join between a relation and the domain attribute. In that case, the select index plays the role of a join index, thereby generating a significant gain on large relations and large domains.

Thus, this performance evaluation shows that our approach is feasible and that join indices are mandatory in all cases while query optimization turns out to be useful only with large databases and complex queries.

6 Conclusion

As smartcards become more and more versatile, multi-applications and powerful, the need for database techniques arises. However, smartcards have severe hardware limitations which make traditional database technology irrelevant. The major problem is scaling down database techniques so they perform well under these limitations. In this paper, we addressed this problem and proposed the design of a PicoDBMS, concentrating on the components which require non traditional techniques.

This paper makes several contributions. First, we analyzed the requirements for a PicoDBMS based on a healthcare application which is representative of personal folder applications and has strong database requirements. We showed that the minimal functionality should include select/project/join/aggregate, access right management and views as well as transaction's atomicity and durability.

Second, we gave an in-depth analysis of the problem by considering the smartcard hardware trends. Based on this analysis, we assumed a smartcard with a reasonable stable memory of a few MB and a small RAM of some KB, and we derived design rules for a PicoDBMS architecture.

Third, we proposed a new highly compact storage model that combines Flat Storage (FS), Domain Storage (DS) and Ring Storage (RS). Ring Storage reduces the indexing cost to its lowest bound. Based on storage cost evaluation, we derived guidelines to decide the best way to store an attribute.

Finally, we proposed query processing techniques which handle complex query plans with no RAM consumption. This is achieved by considering extreme right-deep trees which can pipeline all operators of the plan including aggregates. We measured the performance of our execution model with an implementation of our query engine on two old-fashion computers which we configured to be similar to forthcoming smartcard architectures. We showed that the resulting performance matches the smartcard application's requirements.

This work is done in the context of a new project with Bull Smart Cards and Terminals. The next step is to implement our PicoDBMS on Bull's smartcard new technology, called *OverSoft* [6], and to assess its functionality and performance on real world applications. To this end, we are building an experimentation platform, called *Virtual Campus*, to deal with advanced student folders at the University of Versailles. We also plan to address open issues such as protected logging for durability, query execution on encrypted data and statistics maintenance on a population of cards.

References

[1] A. Ammann, M. Hanrahan, and R. Krishnamurthy. Design of a Memory Resident DBMS. *IEEE COMPCON*, 1985.

[2] R. Anderson, M. Kuhn. Tamper Resistance – a Cautionary Note. *USENIX Workshop on Electronic Commerce*, 1996.

[3] L. Bouganim, O. Kapitskaia, P. Valduriez. Memory-Adaptive Scheduling for Large Query Execution. *Int. Conf. on Information and Knowledge Management (CIKM)*, 1998.

[4] C. Bobineau, L. Bouganim, P. Pucheral, P. Valduriez. PicoDBMS: Scaling down Database Techniques for the Smartcard. *PRiSM Technical Report* n°2000/05, 2000.

[5] F. A. van Bommel, J. Sembritzki, H.-G. Buettner. Overview on Healthcard Projects and Standards. *Health Cards Int. Conf.*, 1999.

[6] Bull S.A. Bull unveils iSimplify! the personal portable portal. www.bull.com/bull_news/

[7] L. C. Carrasco. RDBMS's for Java Cards ? What a Senseless Idea ! www.sqlmachine.com, 1999.

[8] DataQuest. *Chip Card Market and Technology Charge Ahead*. MSAM-WW-DP-9808, 1998.

[9] B. Dipert. FRAM: Ready to ditch niche ? *EDN Access Magazine*, Cahners Publishing Company, 1997.

[10] Gemplus. SIM Cards: From Kilobytes to Megabytes. www.gemplus.fr/about/pressroom/, 1999.

[11] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2), 1993.

[12] G. Graefe. The New Database Imperatives. *Int. Conf. on Data Engineering (ICDE)*, 1998.

[13] IBM Corporation. *DB2 Everywhere – Administration and Application Programming Guide*. IBM Software Documentation, SC26-9675-00, 1999.

[14] International Standardization Organization (ISO). *Integrated Circuit(s) Cards with Contacts – Part 1: Physical Characteristics*. ISO/IEC 7816-1, 1998.

[15] 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.

[16] Microsoft Corporation. Windows for SmartCards Toolkit for Visual Basic 6.0. www.microsoft.com/windowsce/smartcard/, 2000.

[17] M. Missikov, M. Scholl. Relational Queries in a Domain Based DBMS, *ACM SIGMOD Int. Conf. On Management of Data*, 1983.

[18] Oracle Corporation. *Oracle 8i Lite - Oracle Lite SQL Reference*. Oracle Documentation, A73270-01, 1999.

[19] P. Pucheral, J. M. Thévenin, P. Valduriez. Efficient Main Memory Data Management Using the DBGraph Storage Model, *Int. Conf. on Very Large Data Bases (VLDB)*, 1990.

[20] B. Schneier, A. Shostack. Breaking up is hard to do: Modeling Security Threats for Smart Cards. *USENIX Symposium on Smart Cards*, 1999.

[21] Sun Microsystems. *JavaCard 2.1 Application Programming Interface Specification*. JavaSoft documentation, 1999.

[22] Sybase Inc. *Sybase Adaptive Server Anywhere Reference*. CT75KNA, 1999.

[23] J.-P. Tual. MASSC: A Generic Architecture for Multiapplication Smart Cards. *IEEE Micro Journal*, N° 0272-1739/99, 1999.

[24] P. Valduriez. Join Indices, *ACM Trans. on Database Systems*, 12(2), 1987.