1 Introduction

Smartcard is the most secure and cheap portable computing device today. It has been used successfully around the world in various applications involving payment and identification, making it the world’s highest-volume market for semiconductors in the year 2K. As smartcards become multi-application (by hosting a dedicated Java Virtual Machine) and more and more powerful (32 bit processor, more than 1MB of stable storage), the need for database management arises. Embedding database management (query processing, access rights, transaction control) in the card indeed simplifies and makes application code smaller and safer.

A smartcard embedding a DBMS is undoubtedly the best candidate to host data in a highly-available and secure way. This makes it well-suited for ubiquitous computing, where data need be available anywhere, anytime and from any terminal without compromising privacy. Examples of applications in secure ubiquitous computing are: (i) downloadable databases storing confidential data (e.g., diplomatic, military or business information) that can be downloaded on the card, for example before traveling, (2) user environment representing an extensible card’s holder profile collecting computing environment data (PC’s configuration, passwords, cookies, bookmarks, software licenses…), an address book as well as an agenda, and (3) personal folders of different nature: scholastic, healthcare, car maintenance, loyalty.

However, smartcards have severe hardware limitations (tiny RAM, little stable storage, very costly write in stable storage and lack of electrical autonomy) making traditional database techniques irrelevant. Following Moore’s law for processor and memory capacities, smartcards will get rapidly more powerful but will remain very restricted compared to other portable, less secure, devices such as Personal Digital Assistants (PDA). While small footprint versions of popular DBMS (e.g., Oracle 8i Lite, DB2 Everywhere) have been designed for portable computers and PDA, they do not address the more severe limitations of smartcards. ISOL’s SQLJava Machine DBMS [Car99] is the first attempt towards a smartcard DBMS and SCQL [ISO99], 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 a recent paper [BBP+00], we addressed the problem of scaling down database techniques for the smartcard and proposed the design of what we call a PicoDBMS. We gave an in-depth analysis of the problem by considering the smartcard hardware trends, derived design principles for a PicoDBMS and proposed a dedicated storage and query execution model.

Since then, and as promised in that paper, we have implemented a full-fledged PicoDBMS prototype in JavaCard. We are porting it on Bull’s 32 bit smartcard, which we got only recently. Thus, the main objective of the demonstration is to:

- Validate our design by building a complex database application (with respect to the smartcard) on our prototype and showing the benefits of the approach.
- Validate our techniques by showing that they match the smartcard hardware constraints and the user’s response time expectation.

The paper is organized as follows. Section 2 introduces the health card application which we will use for the demonstration. Section 3 presents the PicoDBMS design and implementation choices which are needed to understand the value of the demonstration. Section 4 presents the demonstration platform and discusses the way we validate our techniques.

2 The Health Card Demonstrator

The proposed demonstration is a sample of healthcare application which is very representative of personal folder applications and has strong database requirements. The information stored in the future health cards should include the holder’s identification, insurance data, emergency data (blood type, allergies, vaccination…), the holder’s doctors, prescriptions and even links to heavier data (e.g., X-ray examination, scanner images…) stored on hospital servers. Different users may share data in the holder’s folder: the doctors who consult the patient’s past records and prescribe drugs, 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.

The demonstration will show that our PicoDBMS prototype meets the healthcard application’s requirements, that is: (i) being able to manage a significant amount of data (more in terms of cardinality than in terms of volume because most data can be encoded), (ii) supporting complex queries (e.g., a doctor asks for the sum of antibiotics prescribed to the patient in the last three months), (iii) handling sophisticated access rights management using views and aggregate functions (e.g., a statistical organization may be allowed to access aggregate values only but not the raw data). The benefit of hosting the healthcare database on smartcards rather than on a server is again high availability and high privacy of medical records. The smartcard being the unique trusted part of the system, the data as well as the query engine, the view manager and the access right manager must be confined in the chip.

3 PicoDBMS Design and Implementation

This section recalls from [BBP+00] the hardware constraints of the smartcard, the PicoDBMS design rules which we derived and the main implementation techniques for a PicoDBMS.

3.1 SmartCard constraints

Current smartcards include in a monolithic chip, a 32 bit RISC CPU, memory modules (of about 96 KB of ROM, 4 KB of RAM and 128 KB of EEPROM), security components preventing tampering and take their electrical energy from the terminal [Tua99]. Three important characteristics distinguish a smartcard from any other computing device. First, CPU power is overabundant with respect to memory capacity because smartcards have been designed to support heavy cryptographic algorithms (RSA, DES). Second, smartcards have a tiny die size to minimize the risk of physical attack and to prevent the chip to break when the card is flexed. This strongly impacts the capacity of the memory modules and especially of the RAM which is far more space consuming than ROM or EEPROM. However, according to major smartcard providers, the market pressure should lead to a rapid increase of the smartcard stable storage capacity1. Third, the EEPROM, which serves as stable storage, has a very fast read time (60-100 ns/word) comparable to old fashion RAM, but a dramatically slow write time (more than 1ms/word). Unfortunately, the lack of electrical autonomy makes deferred I/O impossible.

3.2 PicoDBMS design rules

Smartcard’s hardware security makes it the ideal storage support for private data. The impact of ultimate security is that the data as well as the query engine, the view manager and the access right manager must be confined in the chip. Only functions that do not impact the query result can be externalized to the terminal (typically, the GUI, the query parsing and the sort operator). When designing the PicoDBMS’s embedded components, we must follow several design rules derived from the smartcard’s properties [BBP+00]:

- **Compactness rule:** minimize the size of data structures and the PicoDBMS footprint to cope with the limited stable memory area.
- **RAM rule:** minimize the RAM usage given its extremely limited size. In addition, minimizing RAM usage allows to design smartcards with bigger EEPROM.
- **Write rule:** minimize writes in stable storage given their dramatic cost (=1 ms/word).
- **Read rule:** take advantage of the fast read operations (=100 ns/word).
- **CPU rule:** take advantage of the overabundant CPU power (i.e., the time complexity of an algorithm is not a major concern).
- **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.

3.3 PicoDBMS storage model

From the read and access rules, it follows that a PicoDBMS is a typical main-memory DBMS. Thus, it can take advantage of a pointer-based storage and access model to meet compactness and performance altogether. As pictured in Figure 1, the PicoDBMS exploits a combination of Flat Storage (FS), Domain Storage (DS) and Ring Storage (RS). By storing the attribute values embedded in the tuples, FS is adequate when the attribute is small (i.e., smaller than a pointer) or does not present value redundancy (e.g., primary key). In the other cases, DS should be used. DS precludes any duplicate value to occur by grouping values in domains (sets of unique values) and by using references in place of the attribute values. While DS addresses data compactness, RS addresses index compactness. RS links together all tuples sharing the same attribute value through a circular chain of pointer headed by this value. This chain of pointer is stored again in place of the attribute values, providing a similar and compact implementation of both select and join indices. In Figure 1, the RS storage of DrugId can be exploited as a regular index to speed up the join between Prescription and Drug.

---

1 This is being experienced by Gemplus which recently announced Pinocchio, a smartcard equipped with 2 MB of Flash memory [Gem99].
3.4 Query processing

Traditional query processing uses main memory or disk for storing temporary data structures and intermediate results. The RAM and write rules preclude the use of these algorithms in our context. In addition, the security rule precludes any intermediate results to be externalized outside of the chip.

To conform to these three rules, the PicoDBMS execution model must be able to tackle any query, whatever be its complexity and the volume of data it involves, with - almost - no RAM consumption. To this end, we proposed a new execution strategy called extreme right deep tree which executes all operators (including select, join, aggregate and duplicate removal) in a pure pipeline fashion [BBP+00]. Since no intermediate result can be materialized and no temporary data structure can be build, selections (other than the one located at the extreme right deep leaf) must be applied after joins and joins must be computed either by using join ring indices or by nested loops.

Aggregate and duplicate removal can be done in a pipeline fashion as well if and only if: (i) the incoming tuples are yet grouped by distinct values and (ii) the pipeline operators are implemented in an order-preserving way (i.e., they consume and produce tuples in their arrival order). Thus, enforcing an adequate consumption order at the leaf of the execution tree allows pipelined aggregation and duplicate removal. However, this may lead to introduce a Cartesian product in the execution plan if the result must be grouped on several attributes. In the example pictured in Figure 2, the result being grouped on Doctor.id and Drug.type, a Cartesian product is required at the leaf of the tree in order to produce tuples ordered by Doctor.id and Drug.type. This is the price to pay for a pure pipeline execution.

Figure 1: FS, DS and RS on the healthcare database

Figure 2: A complex extreme right deep tree

4 PicoDBMS Demonstration

In this section, we present our demonstration platform and describe how we will demonstrate the healthcare application and three technical points: storage, query execution and performance. To make the demonstration user-friendly and easy to follow, we use graphical tools that help understand the behavior of the PicoDBMS

4.1 Demonstration platform

The demonstration platform (Figure 3) includes a full-fledged PicoDBMS prototype, a JDBC driver, a user interface and a monitor interface. Our PicoDBMS prototype is written in JavaCard 2.1 and runs today on a smartcard simulator and soon on the most recent Bull’s 32 bit smartcard. It includes a storage manager, a query manager, an access right manager and a view manager. The PicoDBMS footprint is around 30KB in its full-equipped configuration.

The user interface comes as a Java applet that can be downloaded on any Java-enabled terminal. It allows formulating SQL queries (see Figure 3), which are sent to the PicoDBMS via a dedicated JDBC driver. The monitor interface delivers both static information (EEPROM image) and runtime information (RAM usage, number of read and write operations, execution time, query execution plans) about the PicoDBMS.

The monitor interface comes as a Java applet that can be downloaded on any Java-enabled terminal. It allows formulating SQL queries (see Figure 3), which are sent to the PicoDBMS via a dedicated JDBC driver. The monitor interface delivers both static information (EEPROM image) and runtime information (RAM usage, number of read and write operations, execution time, query execution plans) about the PicoDBMS.

One difficulty in assessing the performance of our PicoDBMS is that its internal state cannot be easily sniffed when embedded in a chip. Thus, we use the smartcard simulator to experiment with the PicoDBMS on databases larger than the smartcard storage capacity and help understanding the behavior of the PicoDBMS' internals, in combination with the monitor interface.
4.2 Demonstrating the Healthcard application

We will show various scenarios with the healthcard application. First, we will launch the application (which would result from inserting the card in the terminal) and show database schema and access right information. Then, we will exercise the application with the various interfaces (i.e., Patient, Doctor, etc.) and show that the different access rights are enforced inside the card. This will validate the high security of a PicoDBMS.

4.3 Demonstrating the Storage Model

We built a graphical tool to dump the smartcard’s EEPROM memory structure and exhibit major statistics of EEPROM usage. This tool delivers detailed analysis of the PicoDBMS’s behavior during insertions, deletions and updates. Using this tool, we will show to which extent the PicoDBMS storage model satisfies the healthcare application’s requirements in terms of storage capacity. We will also compare our PicoDBMS with other compact storage models (e.g., flat file, light versions of popular DBMS, etc…).

4.4 Demonstrating the query processing Model

We will exercise different types of queries, from simple select/project/join queries to complex queries including aggregation and duplicate removal. This will demonstrate the ability of the PicoDBMS to satisfy the healthcare application’s requirements in terms of query and view capabilities. Furthermore, we will use a graphical tool to monitor the query execution engine. For each query, the execution plan, the RAM consumption and the distribution of the processing time on the different operators will be dynamically displayed.

4.5 Demonstrating the PicoDBMS performance

Our query execution model allows processing fairly complex queries without RAM consumption at the cost of performance degradation. It is therefore necessary to evaluate whether any query can be performed in reasonable time (i.e., without exceeding the user’s patience).

In [BBP+00], we simulated the smartcard environment on old-fashion computers (configured to match forthcoming smartcard capabilities) and we showed that, even for large database and complex queries, the estimated performance was acceptable. Using a real smartcard platform for performance measurements is useful to complement our preliminary conclusions. However, Bull’s 32 bit smartcard storage capacity is about half the one expected. Thus, we will conduct the experiments on a small database on Bull’s smartcard and those on larger databases on a smartcard simulator. We will use these numbers to calibrate our performance model in order to estimate accurately the performance of the PicoDBMS with large databases.

References

[BBP+00] C. Bobineau, L. Bouganim, P. Pucheral, P. Valduriez. PicoDBMS : Scaling Down Database Techniques for the Smartcard (Best Paper Award). Int. Conf. on Very Large Databases (VLDB), 2000.


2 Since our prototype is written in JavaCard, we can easily switch to a more powerful card if available before September 2001.