# Logical and Physical Design Issues for Smart Card Databases

CRISTIANA BOLCHINI, FABIO SALICE, FABIO A. SCHREIBER, and LETIZIA TANCA
Politecnico di Milano

The design of very small databases for smart cards and for portable embedded systems is deeply constrained by the peculiar features of the physical medium. We propose a joint approach to the logical and physical database design phases and evaluate several data structures with respect to the performance, power consumption, and endurance parameters of read/program operations on the Flash-EEPROM storage medium.

Categories and Subject Descriptors: H.2.1 [**Database Management**]: Logical Design; H.2.2 [**Database Management**]: Physical Design—*Access methods*

General Terms: Design, Performance

Additional Key Words and Phrases: Design methodology, access methods, data structures, flash memory, personal information systems, smart card

## 1. INTRODUCTION

The Information Management area is currently seeing the growth, in number as well as in size, of applications that can profit from modern portable devices such as palm computers, cell phones, and smart cards.

While smart cards have been recognized as being among today's most secure portable computing devices [ItGov 2002; Sun Microsystems 1999; DataQuest 1998; Bobineau et al. 2000], almost no attention has been devoted to the most appropriate ways of adapting database techniques to this most powerful tool. Bobineau et al. [2000] made a very thorough examination of the relevant issues that arise in this context, by proposing a storage model complete with query cost evaluation, plus transaction techniques for atomicity and durability in this particular environment. To our knowledge, this is the most complete attempt to analyze in-depth database management system (DBMS) design problems with respect to microdevices.

The attention of Bobineau et al. [2000] was mostly devoted to DBMS design techniques, following the traditional assumption that data structures and

Authors' address: Politecnico di Milano, Dip. Elettronica e Informazione, P.zza L. da Vinci, 32, 20133 Milan, Italy; email: {bolchini,salice,schreibe,tanca}@elet.polimi.it.

access methods should be designed once and for all by the DBMS manufacturer while the database designer should be in charge of conceptual and logical database design, confining the physical design phase to the mere choice of fields to be indexed. By contrast, we claim that, besides the typical interface for table definition and indexing, the data definition language (DDL) of a smart card DBMS should expose also the handles for a number of data structures (with the corresponding algorithms), ready to be chosen by the smart card database designer, appropriately assisted by system support. Indeed, owing to the intrinsic limitations of the smart card medium, the database designer should be aware of the physical constraints from the very beginning of the design process; there are two main design phases where these constraints are to be enforced:

—The classical information system lifecycle contemplates, at conceptual design time, a view design and integration phase where data and procedures belonging to information system areas are first singled out and then appropriately integrated. When the system is distributed, at logical design time tables are partitioned among the system sites according to appropriate fragmentation and replication criteria. In a similar way, in our context, such data and procedures must be accommodated on the smart card itself, and appropriately distributed and replicated on other fixed sites the smart card might be connected to.
—At logical/physical design time, the threefold objective of performance, power consumption and memory endurance optimization must influence the evaluation of on-card data structures and algorithms, and the policy of data manipulation (e.g., delete operations). An accurate study of this phase allows the smart card database designer, appropriately supported by our tool, to specify to the DBMS which of these data structures and algorithms should be used for table access.

Note that power management is also an issue in this scenery, since sometimes the smart card terminal is not directly connected to a power source, for example, in case of subscriber information module (SIM) cards for cellular phones.

Moreover, some applications can involve sensible personal data that require a variable level of privacy. Since privacy aspects have a great importance in a uniform storage medium with tightly packed sensitive information, their impact should be considered as early as possible in the design phase [Bolchini and Schreiber 2002].

The purpose of our research was to conceive a full-fledged database design methodology, which should guide the database designer from the conceptual design step, carried out in the traditional way by using one of the well-known conceptual design models, to the semiautomatic choice of the data structures, access methods, and memory allocation/management policies, based on the analysis of the physical storage devices currently offered by smart card technology. To this aim, we propose a *logical-physical data model*, that is, a number of structures and algorithms the smart card DBMS should implement and offer to the choice of the system-assisted database designer.

Since the real restrictions of today's smart cards concern much more secondary storage capacity than the computational power of their instruction sets, rather than studying possible reductions to SQL (as the standard ISO/IEC 7816-7 does, introducing Structured Card Query Language (SCQL) [Rankl and Ewffing 1999], a subset of SQL for smart cards), we concentrate on the best data organization policies for secondary memory, in order to implement a full-fledged query language.

In this paper we introduce the design methodology, and then concentrate on the logical-physical data model that supports the design of such novel database applications.

The paper is organized as follows: after the introduction of a running example that will be used throughout the paper, Section 2 outlines the proposed methodology and singles out the specific topic addressed by this paper, that is, the guidelines for semiautomatic design of data structures and access methods for smart card data. Section 3 presents the reader with a brief on the most recent technological issues related with our problem, namely, smart card storage devices. Section 4 introduces the data structures we propose for the physical storage of on-card tables, together with the annotations the designer is required to specify in order for the system to provide physical design support. Sections 5 and 6 examine the issue from the DBMS viewpoint, that is, they study the cost of the various access operations as well as the most suitable physical implementation policies, which are strongly related to the technological features of smart card memory support. The discussion is supported by simulation results in Section 7. Future developments and research trends are discussed in Section 8, along with the final conclusions.

## 2. A SMART CARD DATABASE DESIGN METHODOLOGY

In this section, after a short discussion on possible categories of smart card databases, we introduce our running example and briefly outline the general methodology for smart card database design. The next sections are devoted to the details of those methodology steps that are the subject matter of this paper.

In our view, smart card databases can be classified into two main categories:

(a) *Single-application databases (SADs)*, where only one application is modeled: examples of this category are personal financial databases as the stock portfolio, or a personal travel database, recording all the travel information considered interesting by the smart card owner; the PIA (portable internet access database), reported in the sequel as a running example, belongs to this category.

(b) *Personal (micro) information systems (PISs)*, whose most noticeable example is the *citizen's card,* recording such administrative personal data as driver's licence and car information, passport, judicial registry, etc.; another example of this category is the medical record, reporting the owner's clinical history complete with all the past clinical tests and diagnoses [Sutherland and Van Den Heuvel 2002].

A noticeable feature of smart card databases is that the owner and all his/her information have a peculiar role in the entity-relationship (ER) schema; in fact, while items of the card owner's data constitute the virtual center of the database, they often amount to a unique entry, that is, a singleton record. This role could be compared to that of the home page of a Web site, which is just a singleton entity in the site schema, or to the *fact table* of a relational online analytical processing (OLAP) schema for a data warehouse [Atzeni et al. 2000].

This point is much less relevant for SADs than for PISs, where the future applications of smart cards will see the integration of more than one personal application, all conceptually related to one another by means of such a database center. The design of this category of smart card databases presents many similarities to that of distributed applications, for the relevant information must be distributed among the smart card and other fixed devices the smart card will be connected to: imagine the road police department recording all the driver's license and car data, a portion of which is also recorded and appropriately updated on-card, while the court of justice records contains all the possible legal charges against the owner. Notice that this consideration has driven us to proposing, for our methodology, some initial steps that mimic very closely distributed database design.

Now we introduce our running example: a portable database used to store personal information related to Internet navigation.

### 2.1 The Portable Internet Access Database

As an example of a single-application database, let us consider the case of a smart card for Internet access data, storing personal information related to a user's Internet navigation history. Here the key idea is to store information concerning favorite Web sites (bookmarks), {login, password} pairs used to access protected areas of the Web, a log of the most recently accessed unified resource locator (URLs), and, finally, an electronic purse (e-purse) to directly manage payments, with a log of the most recent ones; note that e-purses might be multiple, as in the case of several credit cards (for instance, a corporate one for company expenses and a personal card for one's own use).

All this information is meant to be available on the smart card, possibly protected by encryption in order to guarantee information privacy: the card owner will sit in front of a generic Internet-connected machine, and find his/her Internet environment ready for use.

The entity-relationship diagram in Figure 1 shows the portable internet access (PIA) conceptual schema.

### 2.2 The Methodology

The methodology we propose for smart card database design is shown in Figure 2. It is composed of a common track and two branches. The common track, derived from distributed/federated database design methodologies [Ceri and Pelagatti 1984; Tamer and Valduriez 1991], takes care of the conceptual and logical aspects; the lower part deals with "on chip" features: the left

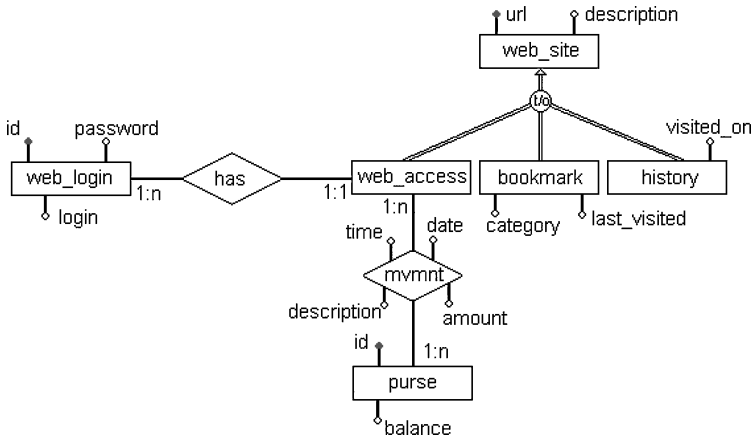


Fig. 1. The entity-relation diagram for the PIA database.

branch concerns privacy/security aspects (discussed in [Bolchini and Schreiber 2002]), while the right branch concerns physical memory data structures and algorithms.

We will discuss the whole tree shortly, but in the following sections we shall mainly focus on the problems related to the lower/right branch. Notice that steps denoted by the "writing hand" icons are tasks for which the designer must provide a manual input to the procedure, while those denoted by PC icons can be automated.

We start by outlining the methodology steps (note that the list numbers are directly related to the numbers in Figure 2):

(1) The relevant information is chosen and modeled; this is done by singling out homogeneous information areas, with the corresponding conceptual views, regardless of the target storage media.
(2) Views are integrated into a global conceptual schema and possible representation and semantic conflicts are resolved; logical entities (for instance relational tables) are designed.
(3) Logical entities are fragmented, possibly both horizontally and vertically, in order to single out "first need" information to be allocated on the smart card. Indeed, as already noted, smart card database design criteria must be very similar to those adopted for distributed database design, since the smart card database will often be part of a larger information system, possibly allocated to several fixed machines besides the smart card(s) itself (themselves).
(4) A strict estimate of the fragment cardinalities is made at this step.
(5) "First need" fragments of the appropriate size are allocated to the smart card, taking into account the card memory size constraint, while the other fragments are allocated to other site(s) of the information system; the interest of this phase of the methodology is better understood reasoning on personal (micro) information systems, for instance a personal medical record,

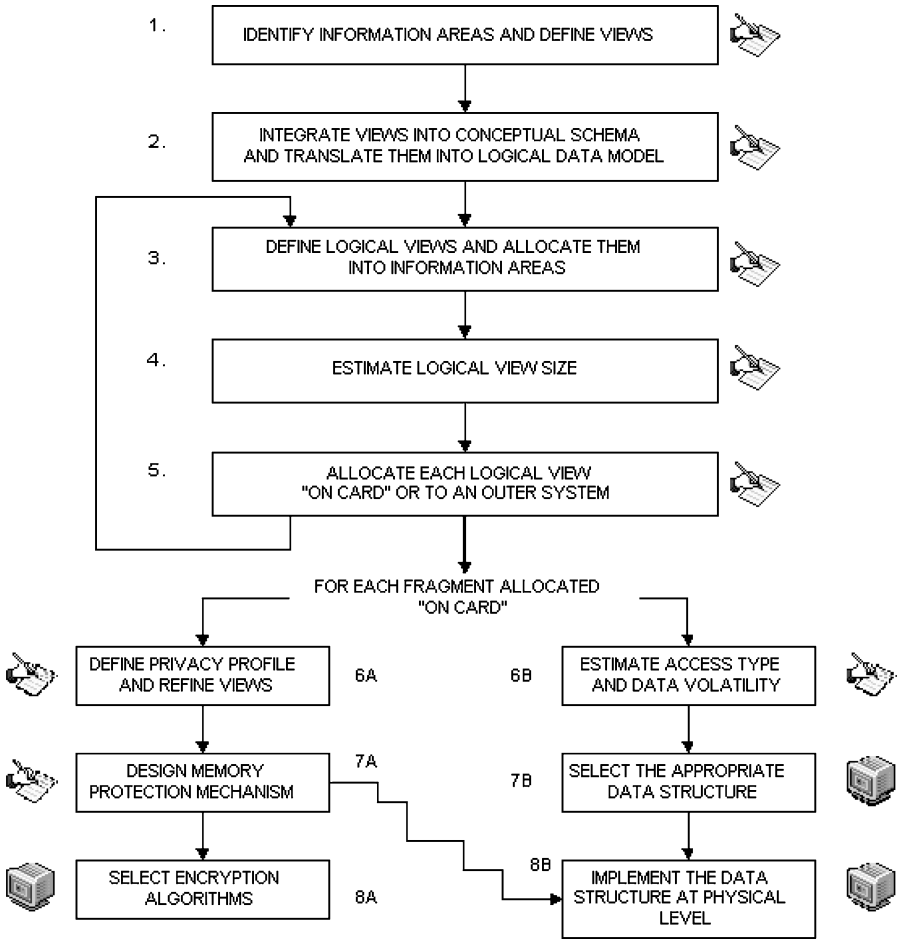


Fig. 2. The entire methodology.

where the most recent clinical tests are kept on card, while the whole personal clinical record is stored in the family physician's computer.

At this point a first comparison is to be made between the fragments and the smart card storage capacity: possibly fragmentation and allocation criteria shall be reconsidered.

(6) (a) Access rights are defined for each fragment and for each user class, and the relevant constraints are included in the view definitions: for instance, in the case of a car accident, the first aid personnel of an ambulance should read from the medical card the patient's blood pressure record, but not his/her possible insurance policies (to be used later by the hospital administration); with respect to the personal legal information card, the traffic authorities should have a read-only access to the relevant information of the judicial registry connected to the individual's license [Bolchini and Schreiber 2002]; (b) the type of access mode (read only, read/write) and the

volatility (for example in terms of update/query ratio) are estimated for each fragment (see Sections 4 to 6 later in this paper).

(7) (a) Memory protection mechanisms are to be designed in order to prevent unauthorised users to access sensitive information; (b) the most convenient data structures are chosen (see Sections 4 to 6).

(8) (a) Possibly an encryption algorithm is chosen for some very sensitive data; (b) access methods are chosen for the data structures defined at step 7(b) under the constraints coming from step 7(a) (see Sections 4 to 6).

At step 6(b) the information is gathered in order to allocate the same type of information, in terms of volume and volatility, on the same sets of blocks. In particular, if asymmetric Flash-EEPROM (EEPROM = electrically erasable programmable read-only memory) is considered (for more details see Section 3.2), where four different block sizes are normally used, very low-volume information can be stored in the smallest blocks while high-volume information can be stored in one or more of the biggest blocks. At steps 7(b) and 8(b), the methodology is supported by a tool for the choice of the appropriate physical data structures and access algorithms among those offered by the smart card DBMS, as shown in Sections 4 to 6.

## 3. SMART CARD TECHNOLOGY ISSUES

Smart cards are essentially devices that allow information storage and processing, but need to interact with an active device providing the necessary power supply. Based on the technology adopted for the memory device and for the on-card processing unit, different types of smart cards can be chosen, according to the required security level.

The simplest architecture is the *processor* card, which contains a microprocessor, a simple cryptographic coprocessor, and blocks of memory including random access memory (RAM), read-only memory (ROM), and a nonvolatile memory (usually EEPROM or Flash-EEPROM). More sophisticated cards, cryptographically enabled, are based on the *processor* card architecture where the simple cryptographic coprocessor is replaced by an advanced one. The improvement in the *crypto* cards allows public key encryption, whereas *processor* cards only provide private key encryption. Given the application environment this paper presents, the target architecture is the microprocessor multifunction card (MMC).

### 3.1 Microprocessor Multifunction Cards

The microcontroller used in Smart card applications contains a central processing unit (CPU) and blocks of memory, including RAM, ROM, and reprogrammable nonvolatile memory (NVM). RAM is used to store executing programs and data temporarily, while ROM is used to store the operating system (Card Operating System, or COS), fixed data, standard routines, and lookup tables. The reprogrammable nonvolatile memory is used to store information that has to be retained when power is removed, but that must also be alterable to accommodate data specific to individual cards or any changes possible

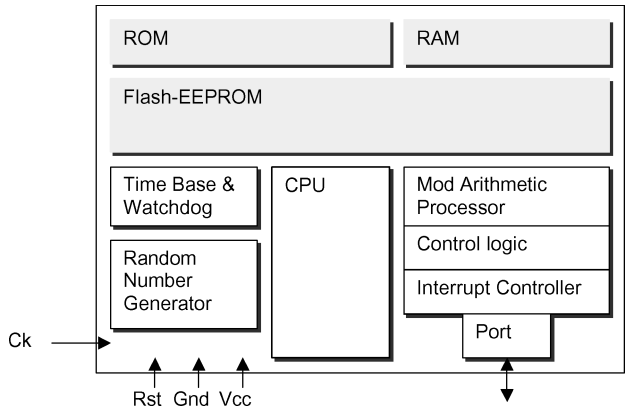


Fig. 3. A schematic representation of a smart card microcontroller.

Table I. Smart Card Microcontroller Characteristics (December 2002)

| Component | Characteristics |
|---|---|
| CPU | 8-16-32 bits |
| RAM | 256 bytes to 1 kbytes |
| ROM | up to 32 kbytes |
| EEPROM | 256 bytes to 64 kbytes |
| write/erase cycles | minimum 100,000 cycles |

over their lifetimes; more specifically, the smartcard NVM constitutes the data storage for the database. Based on this consideration, technology issues have a significant impact on the overall system performance.

Components of this type of architecture include a CPU, RAM, ROM, and EEPROM. The operating system is typically stored in ROM, the CPU uses RAM as its working memory, and most of the data is stored in EEPROM. A rule of thumb for smart card silicon is that RAM requires four times as much space as EEPROM, which in turn requires four times as much space as ROM. Typical conventional smart card architectures (Figure 3) have properties as shown in the following Table I.

A preliminary analysis concerns a comparison of the features and performance of the two main NVM architectures, in order to focus our analysis on a specific memory type. In fact, although classical EEPROM (from now on called simply *EEPROMs*) and Flash-EEPROM (from now on called simply *Flash*) are functionally equivalent (both are electrically erasable reprogrammable nonvolatile memory), they differ in terms of performance, integration density, cost, read/program/erase methods. The comparison we summarize later in this paper seems to be to the advantage of Flash memories as more promising for future MMC. Furthermore, the size reduction of the transistor used to implement memories makes it increasingly difficult to scale EEPROMs since this technology requires relatively high voltage (18 V) to be applied during the write process; thus, EEPROMs will not be suitable for high-density product generation beyond 0.18 $\mu$m [Stockdill 2002].

In EEPROMs, the contents may be erased and programmed at byte level, typically 16, 32, or 64 bytes. The erase operation is transparent to the user; *read time* is approximately 150 ns, *program time* is 10 ms/16, 32, or 64 byte page (157–625 $\mu$s/byte) while *erase time* is not applicable.

In Flash memories *write* data operations are *programming* operations that can only be performed if the target location has never been written before, or has been previously erased. A further complication and space/performance cost arise from the fact the *erase* operation works only on blocks of data at a time. Flash memories are divided into blocks, and each block can be erased separately; *read* and *program* operations work independently of the blocks; any byte (word) can be read or programmed separately.

Thus, in Flash memories the minimum erase element is larger than the minimum read/program element; the erase element is a page, sector, or the entire device, while the read/program element is a byte or a word; *read time* is approximately 80 ns at 5 V (or 120 ns at 3 V), *program time* is 10 $\mu$s/byte at 5 V (17 $\mu$s/byte at 3 V) and the *erase* operation requires approximately 0.45 s/8-kB block at 5 V (0.5 s/8-kB block at 3 V). As for *integration density*, Flash memories are more compact than EEPROMs; in particular, Flash is approximately two times more compact than EEPROM [Stockdill 2002].

Summarizing, although Flash memories are more interesting than EEPROMs in terms of read/write time and integration density, they present a disadvantage concerning the erase operation that is particularly expensive with respect to time and power consumption. However, these drawbacks can be strongly reduced if a correct erase policy is applied. For example, if a block is *erased only when its entire content has to be modified*, an EEPROM memory is approximately four times slower than a Flash one in writing 64k bytes in the same single block for $n$ times. Although this example represents a special case, an appropriate set of policies can make Flash memories much more interesting than EEPROMs.

As a consequence, in what follows we concentrate on the peculiar characteristics of Flash memories, in order to target the previously presented analysis to access times, costs, and architectural characteristics to this specific technology.

In Butler et al. [2001], the authors explored the problems related to EEPROM storage optimization for smart cards and introduced the "transacted memory" technology which embeds transaction capabilities in the memory itself, while managing data information and EEPROM space. The main assumption, which influences and motivates the approach proposed by the authors, is that EEPROM technology allows only block writes. Given this premise, the transacted memory manager provides tags and additional components to optimize storage use and transaction features. In the technological environment we consider, there is not such a restriction to write on a per block basis; thus our approach has different requirements and constraints. Nevertheless, as we will discuss in what follows, we also adopt a tagging technique in order to optimize storage use and to achieve a long EEPROM lifetime.

We must also notice that the peculiar features of Flash memories make smart card databases differ from main memory databases (MMDBs). In fact, Eich [1992] noted: "In a Main Memory Database system . . . the need for I/O

Table II. EEPROM and Flash-EEPROM Technical Characteristics (December 2002)

| | EEPROM | Flash-EEPROM |
|---|---|---|
| Read (nsec) | 150 | 80@5V to 120@3V |
| Program ($\mu$sec/byte) | 157 (64 Byte) to 625 (16 Byte) | 10@5V to 17@3V |
| Erase (sec/block) | N.A. | 0.45@5V to 0.5@3V |
| Cell Components | 2 transistors | 1 transistor |
| Cell Size ($\mu m^2$ @ 0.4 $\mu$m tech) | 4,2 | 2 |
| Cost per bit | Medium | Low |
| Endurance (write/erase cycles) | 10k to 100k write cycle/byte | 10k to 100k erase cycle/block |

operations to perform database applications is eliminated . . .” (p. 507); this assertion is due to the substantial homogeneity of the RAM storage medium for MMDBs [Garcia-Molina and Salem 1992]. Moreover a great part of main memory database studies are devoted to overcoming the RAM volatility properties. This is not the case for a smart card database system where Flash memory acts as a true secondary storage with very different read/write properties with respect to the RAM main memory, making it look more like a traditional DBMS.

### 3.2 Flash Memories: Dimension, Power and Timing Issues

So far, the amount of data storage available in a Flash memory ranges form 32 Mbit to 512 Mbit (both organized by 8 or 16). Research for 512 Mbit and above is currently taking place. Power requirements vary depending on the operation that has to be performed. A *read* operation requires an average of about 10 mA (12 mA max) whereas *program* and *erase* operations require an average of about 20 mA (35 mA max).

Access time depends on both the operations and the mode. To enlarge on what we introduced in Section 3.1, as far as *read* access is concerned, random *read* requires about 150 ns (for example, 100 ns in NOR Flash architecture (Intel, AMD) and 120 ns in DINOR Flash architecture (Mitsubishi)), whereas *read page mode* and *read burst mode* (when supported by the memory) requires about 35 $\mu$s to access the first byte in a page and 50 ns for subsequent reads (for example, 25 $\mu$s first byte and 50 ns subsequent bytes in NAND Flash architecture (Toshiba) and 50 $\mu$s first byte and 50 ns subsequent bytes in AND Flash architecture (Hitachi)). Concerning programming time, each program operation takes approximately an average of 13 $\mu$s per byte or word. Erase time depends on block dimension: typically a 64-kbyte block takes about 0.7 s. Another consideration concerns Flash endurance: in fact, a Flash memory works for 100,000 erase cycles/block. This data is all reported in Table II.

In order to achieve good performance and a long endurance, it is thus necessary to reduce the number of data modifications. Since *update*, *delete*, and *insert* operations are required and inevitable, the remainder of the paper focuses on storage and management techniques that minimize the cost associated with the required operations.

It is worth noting that the time to erase a block is approximately 10 times that required to program a block, which, in turn, is 100 times that required

to read a block; as a consequence, our effort is to reduce the number of erase operations.

## 4. DATA STRUCTURES AND ANNOTATIONS: THE LOGICAL-TO-PHYSICAL DATA MODEL

Given the main applications we envisage for smart card databases, and in general for embedded systems based on a Flash memory permanent storage, we assume that the volumes of data which must be readily available will not need to be not very huge: data fragmentation and allocation will lead to a reduction of the cardinality of the relations actually stored on the smart card. For example, even if we imagine that the PIA will store all the user's visited sites, most probably only the last 30 sites will be kept on card, while the rest will constitute a fragment allocated in the card owner's personal computer. Thus we claim that the data types adopted for physical storage must be very simple, appropriately used in relation to the data volume and usage type of each specific data class.

In order to illustrate this part of the methodology, we rely of the PIA database example, introduced in Section 2: this very simple example will be useful for this paper's purposes, where the relevant issue is more related to logical and physical considerations than to the (equally interesting) subject of fragmentation and allocation criteria.

From now on, to fix ideas and without loss of generality, we shall refer to the *relational model of data* [Atzeni et al. 2000], also because this model affords a high degree of simplicity consistent with the dimensional requirements of the device. However, our considerations can be extended to other logical models without much effort. We consider the logical schema of the PIA database shown in Figure 1 (primary keys are underlined):

```
TBL_BOOKMARKS (URL,DESCRIPTION,CATEGORY, LAST_VISITED)
TBL_HISTORY (URL, DESCRIPTION, VISITED_ON)
TBL_WEBLOGIN (ID, LOGIN, PASSWORD)
TBL_ACCESS (URL, DESCRIPTION, LOGIN_ID)
TBL_MOVEMENTS (PURSE_ID, DATE, TIME, URL, DESCRIPTION, AMOUNT)
TBL_PURSE (ID, BALANCE)
```

In order to model the relations to be stored and managed by the database, four data types have been identified[1]:

—*Heap relation*, characterized by a limited cardinality, generally less than 10 records, used to store a few records, unsorted, typically accessed by scanning all records when looking for a specific one. In the running example, the login/password pairs relation fits these characteristics.

—*Sorted relation*, characterized by medium cardinality (∼100 to ∼1000 records) and by being sorted with respect to a field. This kind of relation is used to store information typically accessed by the order key. Depending on the size of the relation, it will be possible to store the entire relation on

[1]The terms are partially taken from file classification.

card, or eventually only a subset of the records; once the upper bound is reached, the user will have to delete a record before adding a new one. With respect to the running example, URL bookmarks can be managed by means of sorted data.

—*Circular list relation*, still characterized by the same cardinality of the previous data type, but stored and managed as a circular list, typically sorted by *date*/*time*. This kind of relation is typically suitable to manage a fixed number of log data; once the maximum number of records is reached, the next new record will substitute the oldest one. In the PIA example, data logs of the last $m$ URLs most recently visited and the last $n$ payments in the e-purse can be stored by means of circular lists.

—*Generic relation* (multiindex structure), not belonging to the previously defined categories. Such data might be indexed with respect to more than one field, by means of some of the data structures suggested in Bobineau et al. [2000] (for instance the *ring index* approach detailed later in this paper). This is the only data structure we propose which resembles the data structures used in classical DBMSs; however, due to the limited amount of data, we believe that such data structure will seldom be useful.

### 4.1 Table Schema Annotation

In this phase the designer feeds the tool with the information needed to choose, for each relation, one of the four data structures just presented.

As a matter of fact, while logical design involves all the parts of the database, on card or allocated on the fixed devices, schema annotation is applied only to those relations that will reside on the card. Thus, input of this phase is the on-card table and fragment definition as produced by steps 4 and 5 of the methodology.

The designer must input, for each on-card relation, the following information:

—number, type, and size (when not determined by the type, e.g., a variable length character field) of the attributes; such information will provide records' length (in bytes);

—expected relation size, that is, number of records (order of magnitude); and

—existence of an upper bound of the number of records during the stable phase of the database life.

It is then possible to evaluate the amount of permanent memory that will be required to store the data, without taking into account additional costs due to data management. Such overheads will be estimated in the subsequent steps of the design methodology.

Now all the relation schemas must be annotated with those aspects that lead to the selection of the best data type and memory support. The purpose of this annotation is to provide estimates of the elements that mainly impact on access time and card endurance, in order for the tool to select the most suitable physical data schema and implementation. For each table, we must know the

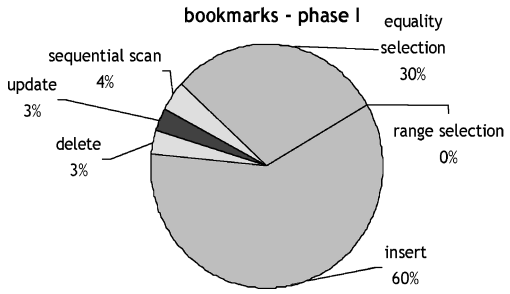


Fig. 4. Expected operation frequency statements for the initial life of the database for the bookmarks relation. The percentages shown on the chart are determined by the input qualitative values.

following:

—Is it a *sorted relation* (no further indices): yes/no?
  —*Log-like* (e.g. sorted on date/time)
  —Other (specify sort key)
—What is the expected *relative* frequency of the following operations:
  —Insert?
  —Delete?
  —Update?
  —Select (sequential scan, equality selection, range selection)?

The designer is expected to give, for each relation, a qualitative composition of the foreseen workload stating how data access will be distributed among the possible query types. Note that it is not required to express a precise value; instead what is needed is a percentage of one operation type with respect to the others. Consider as an example the bookmarks relation in the PIA database. As further discussed later in this paper, we can distinguish two phases in the relation lifetime: an initial time, when the relation is empty and we are building our bookmarks list, and a subsequent phase, when we mainly search the bookmark of the Web site we want to visit.

The designer might thus produce the following two "expected operation frequency statements":

—*Initial/startup phase—phase* I (Figure 4):
  —Insert: high frequency;
  —Delete: low frequency;
  —Update: low frequency;
  —Select:
    Sequential scan: low frequency;
    Equality search: medium frequency;
    Range search: no operation.
—*Stable phase—Phase* II (Figure 5):
  —Insert: low frequency;
  —Delete: low frequency;

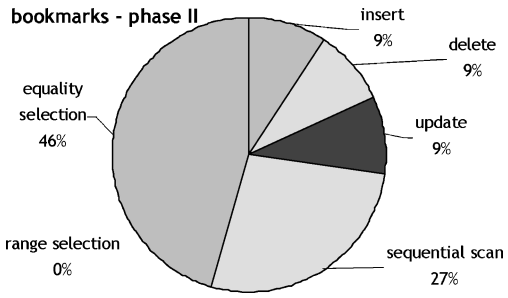


Fig. 5. Expected operation frequency statements during the stable life of the database for the bookmarks relation. The chart shows the percentages determined by the input qualitative values.

—Update: low frequency;
—Select:
Sequential scan: medium frequency;
Equality search: high frequency;
Range search: no operation.

The designer is expected to enter these parameters as qualitative values: for each operation, whether the operation will never occur (no operation), occur seldom (low frequency), occur moderately (medium frequency), or occur often (high frequency) with respect to the possible operations. It has to be understood that the terms *low*, *medium*, and *high* here have only *relative* significance. This information is used to foresee the work balance between the types of operations, also characterizing each table's volatility in order to select the most convenient memory support, especially since the technological aspects have a relevant impact on the smart card performance and life. Such information is required for each table belonging to the "on-card" relations, and it is provided for the long-term, stable, life of the database.

This annotation step has been carried out on the PIA database, whose annotated logical schema is presented in Figure 6.

The next step in the methodology is performed automatically on the basis of the gathered information, as described in the following section.

### 4.2 Physical Design

The tool allocation policy interprets user's annotations to select one of the available data structures for each one of the database relations. When processing each relation and its annotations, one of the following situations is met:

(A) limited cardinality relations;
(B) ordered date/time field (maximum cardinality constraint);
(C) high number of expected records, ordered and with a high volatility (possibly maximum cardinality constraint);
(D) none of the above.

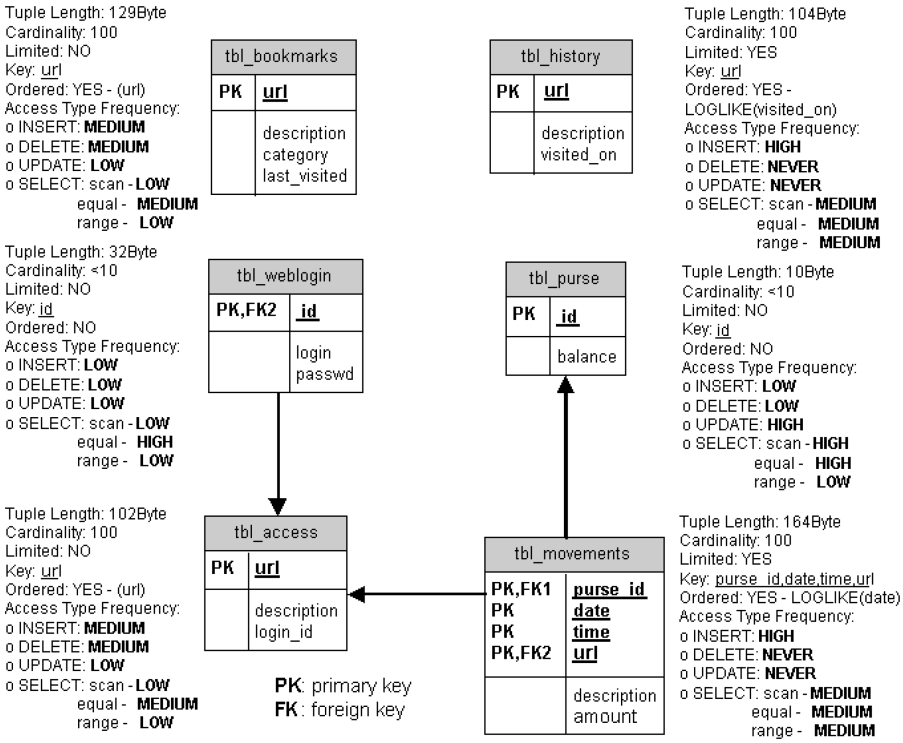


Fig. 6. The annotated logical schema for the PIA database. The table annotations will be used to determine, for each relation, the most suitable physical data model.

In the *physical data organization* step, data structures are selected on the basis of these four categories:

(A) *Limited cardinality relations*. When data cardinality is limited (below 10, 20), a heap organization is viable even for sorted data, since the overhead of any complex structure maintaining the sorting is not costly, owing to the short time required for scanning the whole data.

(B) *Ordered date/time field (maximum cardinality constraint).* The selection of the circular list data model for relations with a date/time ordering and a maximum cardinality constraint is straightforward.

(C) *High number of expected records, ordered and with a high volatility.* Relations belonging to the third class are physically modeled by means of an ordered data structure. It is possible to foresee a particular overhead in maintaining data sorting during the initial phase of the database, when modification operations (i.e., insert, update, delete) prevail with respect to retrieval queries. As a consequence, the implementation of such an ordered data structure will need to provide a mechanism to efficiently support modification operations, as discussed in Section 6.4.

(D) *None of the above*. When the relation cannot be referred to any of the discussed situations, some classical data structure for the implementation

of relational tables is adopted, possibly introducing indices when items of data are mainly downloaded on the card at programming time and thus produce mainly query accesses rather than data modifications, as we will see what follows.

With respect to the PIA example, the applied annotation leads to the following results:

```
TBL_BOOKMARKS: SORTED
TBL_HISTORY: CIRCULAR LIST
TBL_WEBLOGIN: HEAP
TBL_ACCESS: SORTED
TBL_MOVEMENTS: CIRCULAR LIST
TBL_PURSE: HEAP
```

## 5. ACCESS TYPES, OPERATIONS AND COSTS

In this section, we perform a comparison of the four adopted data types with respect to the typical foreseen operations to be performed on the data of each on-card table [Ramakrishnan and Gehrke 2000].

—*Scan*: fetch all records in the table.
—*Search with equality selection*: fetch all records that satisfy an equality selection, for instance, "find the VisitedURL for day = #05/01/2001#."
—*Search with range selection*: fetch all records that satisfy a range selection, such as, for example, "find the movements with amount between $10 and $100."
—*Insert*: insert a given record into a table. Depending on how the records are stored, ordered or not, the operation may require a shift of the records following the position where the new record is inserted, that is, fetch all records, include the new one, and write back all records.
—*Delete*: search the record(s) and remove it (them) by freeing the space. More precisely, it is necessary to identify the record, fetch the block containing it, modify it, and write the block back. Depending on the record organization, it may be necessary to fetch, modify, and write back all the records following the one under consideration.
—*Update*: search a given record(s), read it (them) and rewrite it (them).

In order to estimate the cost (in terms of access/execution time) of different database operations:

—let $B$ be the number of data blocks with $R$ records per block, and $C$ the average time to process a record (e.g., to compare a field value to a selection constant);
—let *FRB*, *FWB*, and *BE* be, respectively, the average time to read, write, and erase a block from/to the Flash memory;
—let *FRR* and *FWR* be the average time to read and write a single record from/to the Flash memory, respectively;

Table III. Access Timing Costs Estimated for the Three Data Models with Respect to the Access Operation

| | Heap data | Sorted data | Circular data |
|---|---|---|---|
| Scan | $T_{S\_H} = B \times R \times (FRR + C)$ | $T_{S\_S} = B \times R \times (FRR + C)$ | $T_{S\_C} = B \times R \times (FRR + C)$ |
| Search eq. | $T_{SEQ\_H} = 1/2\ B \times R \times (FRR + C)$ | $T_{SEQ\_S} = 2 \times FRR \times Log_2\ B + C \times Log_2\ R$ | $T_{SEQ\_C} = 2 \times FRR \times Log_2\ B + R \times Log_2\ C$ |
| Search range | $T_{SR\_H} = B \times R \times (FRR + C) + (FRR + RWR) \times n\_r$ | $T_{SR\_S} = T_{SEQ\_S} + (FRR + RWR) \times n\_r$ | $T_{SR\_C} = T_{SEQ\_C} + (FRR + RWR) \times n\_r$ |
| Insert | $T_{IN\_H} = FRR \times R + C + FWR$ | $T_{IN\_S} = T_{SEQ\_S} + (FRR + FWR + RRR + RWR) \times R + C + BE + 1/2\ B \times (FRR + FWR + RRR + RWR + BE)$ | $T_{IN\_C} = FRR \times R + C + FWR$ |
| Delete | $T_{DE\_H} = T_{SEQ\_H} + (FRR + FWR + RRR + RWR) \times R + C + BE$ | $T_{DE\_S} = T_{IN\_S}$ | $T_{DE\_C} = FRR \times R + C + (FRR + FWR + RRR + RWR) \times R + C + BE$ |
| Update | $T_{UP\_H} = T_{DE\_H}$ | $T_{UP\_S} = T_{SEQ\_S} + (FRR + FWR + RRR + RWR) \times R + C + BE$ | $T_{UP\_C} = T_{DE\_C}$ |

—and finally, for RAM access, let *RRR* and *RWR* represent the time to read and write a record in RAM, respectively.

Table III reports the costs for accessing stored data with respect to the different organizations. The adopted algorithms exploit the existence of a sorting field when available, and take into account the Flash technological requirement of allowing the programming only of unwritten elements. As a result, the algorithms (and consequently the costs) feature—when necessary—a procedure for saving data in RAM, erasing an entire Flash block, and restoring from RAM the part of the block that must be left unchanged. Table III summarizes access costs for the first three discussed data models.

As far as *heap relations* are concerned, there is no specific issue that involves any of the possible data accesses. Block erasure, which is actually the peculiar aspect, occurs anytime a modification needs to be carried out on the stored data, either to delete records or to update existing ones.

*Sorted relations* are the ones mostly affected by the technological constraint, due to the need to rewrite the block containing the information to be updated, and, for the insert and delete operations, eventually the adjacent blocks if the records' shifting involves more than a single block.

As far as *circular list relations* are concerned, let us notice that this kind of relation is typically adopted for logging information, and thus the operations most commonly performed are insertion of a new record after all the valid ones. A pointer to the first free memory location is maintained so that it is not necessary to carry out a search before inserting the record. Delete and update operations are expected to occur seldom; nevertheless their costs have been estimated as well.

*Generic relations*, with *multiple indices*, were presented in Bobineau et al. [2000], where the authors proposed an indexing mechanism for smart cards,

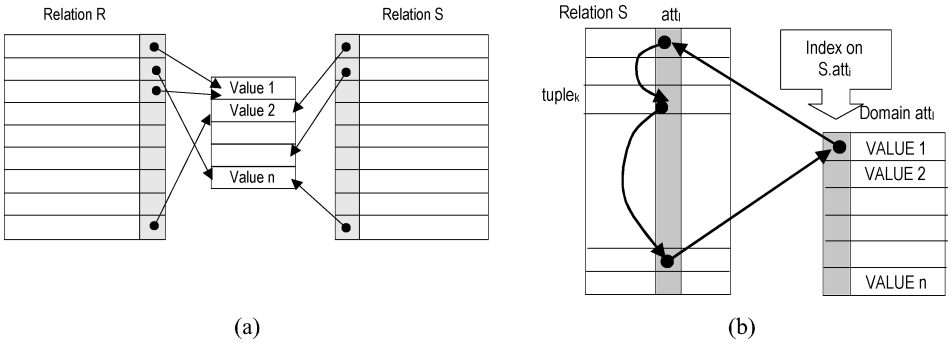


Fig. 7. (a) Domain index and (b) ring index as defined in Bobineau et al. [2000].

to make such indices as compact as possible: the idea applies to indices where attribute domains have very small cardinality. This has not been covered by our cost estimation: the reader may refer to Bobineau et al. [2000], where a thorough cost analysis was provided.

In the *domain index* solution, attribute values are stored as pointers to the domain value, stored only once in an appropriate domain table (Figure 7(a)). In the *ring index* solution, Bobineau et al. [2000] proposed to use *value-to-tuple* pointers within the domain table records in place of traditional *tuple-to-value* pointers. This implements an index structure having a ring form, from the domain values, through all the tuples having such a value (Figure 7(b)). As an alternative, classical structures can be found in the literature [Elmasri and Navate 1994; Wiederhold 1987]. Such indexed structures are, though, particularly complex and hard to manage, especially when considering that the database on smart cards will seldom require complex join operations. Since the analysis of possible real applications on smart cards suggests a limited necessity of multiple indices data structures, which are particularly time and power consuming when implemented on a Flash memory, no such structure is further discussed in this work.

## 6. IMPLEMENTATION OF THE LOGICAL-TO-PHYSICAL DATA MODEL

The goal of the data structures implemented by the smart card DBMS is to optimize performance and minimize power consumption and Flash memory degradation while limiting area and computational overheads. Do note that these aspects are strictly related, and that block erasure significantly affects all these parameters.

According to the technical features of the storage device we have selected, that is, Flash memory, we propose an implementation of the physical data model previously discussed, based on the introduction of two elements:

—Use of *deleted_bit* to carry out a logical rather than physical deletion of the record, to minimize response time, power consumption, and the device degradation implied by the physical block erasure required by a delete/update operation;

—Introduction of a number of *dummy records per block*, allowing the control of the filling of a block and the organization of the records within the block. Such techniques are already widely used in the management of several data structures; noticeable examples can be found in B-trees of order $n$, where each node can host a number of items varying from $n/2$ to $n$, or in static hash tables where the filling of pages is controlled in order to avoid too many collisions [Elmasri and Navate 1994; Wiederhold 1987].

These two aspects are discussed in Sections 6.1 and 6.2, focusing on their benefits and costs, and comparing them with a straightforward implementation of the identified data structures according to traditional approaches.

### 6.1 Use of *deleted_bit*

The introduction of this bit positively affects performance every time a record needs to be deleted and a logical elimination of the record can be performed rather than the physical one. This technique has been adopted for the following three basic situations:

6.1.1 *Record Deletions.* The deletion of a record in a block requires determining the location of the record (by means of a search) within a block and the erasure and reprogramming of the block. In order to save time, a delete can be managed by means of an additional bit per record, *deleted_bit*, to be set (programmed) to 1 when the record is deleted and not valid anymore.

This will cause the presence of invalid records within a block, which leads both to a waste of space and a performance degradation in all operations involving a search. This problem is solved by performing a *garbage collection*-like operation during each erase/program cycle of a block, where only the valid records are programmed, according to the data structure (heap, ordered, . . . ). This solution allows us to reduce the number of erasures to be performed, exploiting the necessary ones to carry out also the pending delete requests.

6.1.2 *Record Updates.* Update operations are often seen as a sequence of delete and insert operations. In our technological scenario, the delete operation consists of programming *deleted_bit*, and introducing a new record. In a sorted relation and in the case of a circular list, when updating a random record (not the last one) the insertion requires the erase/program operation for at least one block; thus there is no gain in interpreting the update operation as an independent delete followed by an insert.

On the other hand, if the insertion requires no ordering, as in the heap and the circular list (when updating the last record) data organization, the update consists of two program operations, a convenient solution in terms of both time and endurance. As a result, depending on the type of data organization, the update consists of the deletion of a record and the introduction of a new independent one, or of the erase/program of the block containing the involved record.

6.1.3 *Circular Lists Implementation.* The circular list data organization requires an append operation and a delete operation for each inserted element.

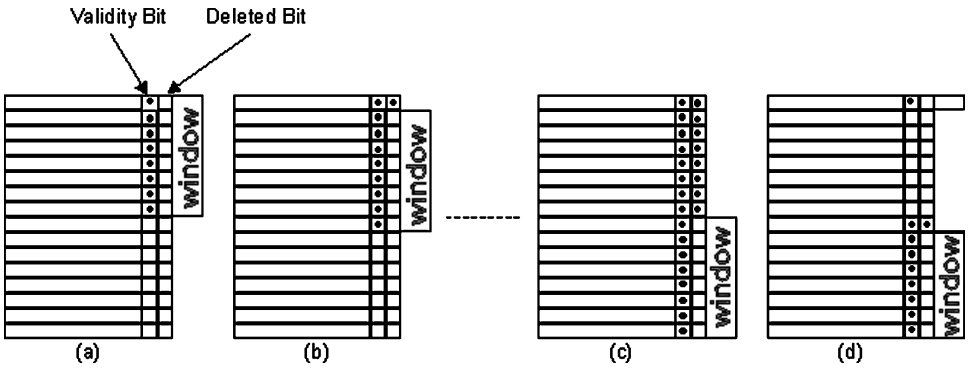


Fig. 8. Circular list implementation by means of a sliding window.

Independently of how the list is organized (single or multiblock), each update requires the erasure of the block impacting energy consumption, endurance and performance.

Let $j$ denote the number of records to be provided by the circular list[2] and $k$ the actual number of physical records used to implement the mechanism. The user view of data is obtained by means of a *sliding window* $j$ records long, while the actual space allocated to the circular list is $k$ records long. In this way, the number of data observable by the user is constant, equal to $j$, and independent of time. This data organization allows us to save energy and maxime performance, but consumes space. Figure 8 shows data organization and its potential evolution in the case of $k = 2 * j$; the gray area represents the sliding window.

Let us assume the initial situation shown in Figure 8(a) with the $j$ elements already inserted; when a new element is appended, the first valid element of the list is deleted and the window slides (the situation in Figure 8(b)). After $k - j$ appends (Figure 8(c)) the available space is used up and a block erase has to be performed before a new append can be completed (Figure 8(d)). It is worth noting that the erase operation has period $k - j$ and is independent of the number of blocks involved.

### 6.2 Use of Dummy Records

By further exploiting the idea of reducing memory modifications by deleting records logically rather than physically (except when necessary), we introduce a storage management policy that "leaves" unused records, so that subsequent insert operations requiring an ordering may be performed without moving the preexisting records. The term *dummy record* has been selected to identify these unused records among the ones containing the relation data.

6.2.1 *Sorted Data Structure Implementation.* Insert operations on a sorted relation are characterized by a high probability to cause a block erasure in order to insert the new record (unless an *append* situation occurs) and eventually cause the adjacent and following blocks to be erased and reprogrammed too.

[2]It is worth noting that it is not relevant whether the physical elements are concentrated into a single block or are distributed on more blocks.

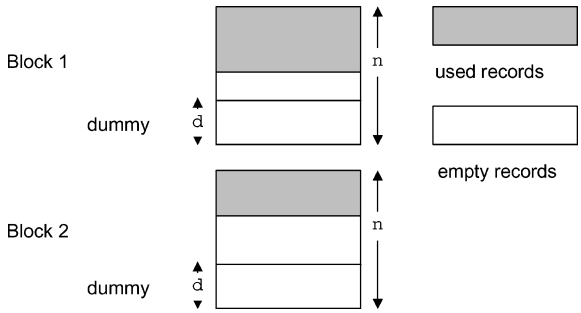


Fig. 9. The buffer of $d$ dummy records out of the $n$ records of the block.

This (these) block(s) erase/program task has (have) a significant overhead in terms of time, power, and endurance. Indeed, the analysis of the actual operations to be carried out relates to the general considerations presented in Section 5, with the only exception related to the fact that the entire block is erased when the operation requires it. The use of *deleted_bit* and its impact on the operation costs is straightforward; thus, our physical implementation policy has envisioned two alternative physical data organizations to limit the overall number of erase/program operations, both based on the use of *dummy records*.

For each $n$-records sized block of memory, $d$ empty records are used as buffers to be left empty and used as the block gets filled; the aim of these records is to reduce the number of blocks involved by the insert operation, possibly to one block, the one where the record is to be inserted. Consider the following situation with respect to the insert operation. In this policy, for the insertion of the first $n$ records, the existence of the dummy buffer does not introduce any difference and Block 1 is totally filled. When the next record needs to be inserted, the total of $n + 1$ records will be distributed between the two blocks (two erase/program operations) so that in Block 1 there is a buffer of $d$ empty records and $d + 1$ records are stored in Block 2. The next insertion will most probably (unless an append occurs and only a program operation is performed) require modifying (erasing and programming) either Block 1 or Block 2, since now there is room in the buffer of Block 1 (e.g. Figure 9). Thus, the main difference in the behavior of the data structure with respect to a structure with no dummy records can be seen after the first $n$ records are inserted, introducing a different number of erase/program operations. Clearly, there is a complexity overhead due to record management and to the need for additional counters for keeping track of the current space in the blocks; moreover, depending on the location of the dummy buffer, there is an impact on *read* operations.

Different scenarios have been taken into consideration, with respect to the *position of dummy records*. Considering the locality of dummy records, the alternatives are as follows:

—dummy records are all adjacent, either at the beginning or at the end of the block, or

—dummy records are distributed in the block.

The policies are described in the following two sections, and the simulations of their behaviors are compared in Section 6.4.3.

6.2.2 *Adjacent Dummy Records Policy.* By keeping all used records adjacent to one another (as well as the dummy ones), search operations are efficient, since the scanned records are only the valid ones; in fact, given $x$ valid records out of the total $n$ records of the block, the cost of a search relates to $x$ rather than to $n$.

*Delete.* *deleted_bit* is programmed. Cost: *search* + *program*.

*Update.* The block is erased and programmed with the modified record. Cost: *search* + *erase* + *program*. It is worth noting that no overflow ever occurs in the following block.

*Insert.* The operation cost, when involving a single block, is that of a *search*, plus the cost of a *program* and, if it is not an append kind of insertion, there is the added cost of the *erase*. Altogether, the costs are as follows:

—Insert (append): *search* + *program*;
—Insert (no append): *search* + *erase* + *program*.

In case there are no dummy records available, this operation may cause an overflow in the block where the new record is to be inserted. Actually, depending on the emptiness level of the following blocks, a waterfall mechanism may start to redistribute records in the blocks. In the worst case, all the blocks of the relation need to be erased and programmed. Do note that, depending on the distribution of the valid records, it might happen that, when inserting a new record, if the following blocks are full (and thus there are free records in the preceding blocks, otherwise the insertion would be forbidden), even a block preceding the one where the record is to be inserted needs to be erased and programmed.

Insertions are managed as follows: the first $n$ records are inserted in the first block. When the $n + 1$th record needs to be inserted, $n - d$ records remain in the first block and, $d + 1$ are programmed in the second block; as a result, the first block is erased and programmed, while the second one is programmed.

The next $d$ insertions require the erasure of a single block (they possibly can fit all in the first block).

When the total number of valid records of all blocks of the relation reaches $(n - d) \times b$, two approaches can be adopted: either the remaining dummy records are redistributed among the blocks or no further movement of records is performed in order to provide a limited number of dummy records per block. Of course, when the number of valid records is equal to $n \times b$, no further records can be inserted.

6.2.3 *Distributed Dummy Records Policy.* By considering an even distribution of the records to be inserted with respect to the sort key, the use of adjacent dummy records limits the number of blocks involved in the operation to one. Nevertheless, in order to perform the insertion of a record in the correct order,

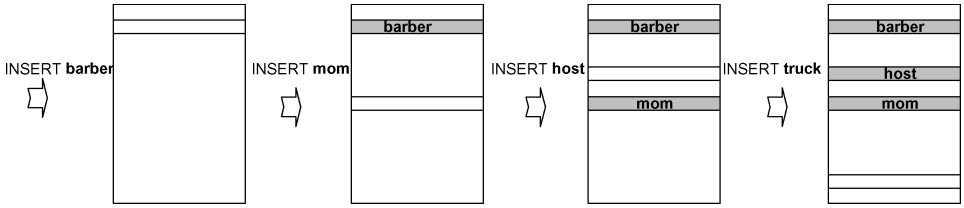


Fig. 10. Distributed dummy records per block to reduce the number of program/erase operations.

if all records are inserted adjacent to one another, the probability of erasing the block that will contain the record is very high.

As a consequence, in order to keep the probability of block erasure low, the dummy records are evenly distributed through the block so that each record is inserted according to the defined order, leaving empty records between the existing records in the block. This is achieved by means of an address-mapping function (similar to a hash function) that, given the sort key value, determines the address in the block of the record to be inserted (Figure 10).

The sequence shown in Figure 10 refers to four insert operations in an ordered fashion: each time an insert is performed, the address-mapping function is evaluated on the value of the field with respect to which ordering is maintained. The address-mapping function is implemented in hardware exploiting the additional spare logic available in the smart card architecture.

To appreciate the effectiveness of the use of dummy records, consider the diagram shown in Figure 11, where each tile represents a block erasure due to an insert operation. As an example, consider the traditional sorted relation management (first diagram). After the first three operations, a block erasure occurs on the sorted relation with no particular management policy; the same occurs with the adoption of the *deleted_bit* policy. Such an event does not occur until the 30th operation, when dummy records are used.

The introduction of *deleted_bit* reduces the number of block erasures (26) with respect to the traditional, straightforward approach (31); the adoption of dummy records significantly improves the performance, requiring only seven erasures, confirming the expected results.

### 6.3 Overheads

The introduction of the proposed logical and physical implementation strategies are characterized by costs in terms of area for storing the additional information and management for maintaining a consistent data and control information.

6.3.1 *Deleted and Validity Bits Implementation.* The additional bits for managing records with both the *deleted_bit* strategy and the *validity_bit* strategy (associated with the dummy records policy) have been represented as part of the data record. Such a representation is a logical one: at the physical level, in order to optimize control bit manipulation, we group them at the beginning (or at the end) of a memory block, as shown in Figure 12.

This separation between control bits and data allows one to optimize the access to valid records and to delimitate the range of action on the memory

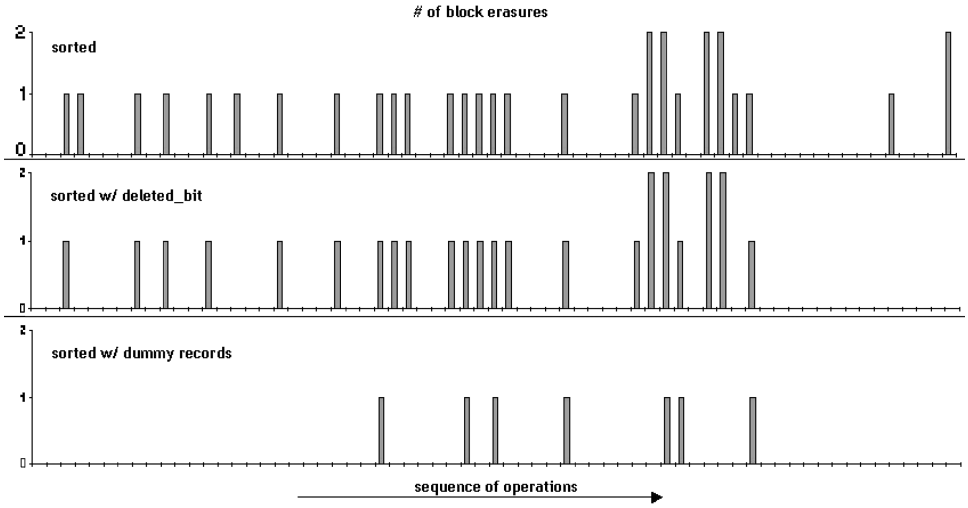


Fig. 11. Block erasure operations caused by a sequence of access operations (insert, select, delete, and update) on a sorted relation; each tile represents one block erasure. Three different data structures are compared: sorted, sorted with *deleted_bit*, sorted with dummy records.

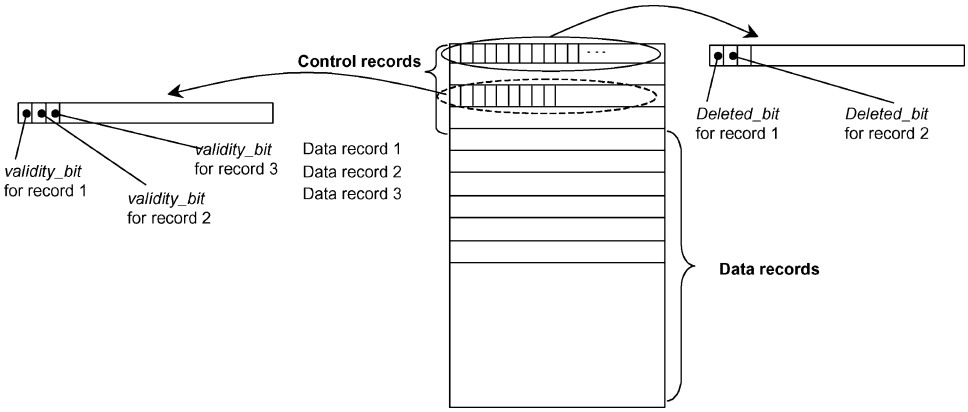


Fig. 12. Flash memory block organization with specific records for control bits (deleted and/or validity) and data.

block for updating the status of a record. It is worth noting that both types of control bits need to be written when a record is first valid and when it is deleted. Thus data record modification implies the programming of control bits, never the erasure, unless the record needs to be physically deleted. In such a case, though, the entire block is erased together with its control bits. As a result, the programming of control bits and their storage in nonvolatile memory does not introduce any overhead in terms of additional block erasure operations; the overhead only consists of the memory block space devoted to such accessory information.

6.3.2 *Counter and Pointer Management.* For the circular list data structure and for the concentrated dummy records strategy, additional information indicating the first available data record is necessary and needs to be stored in nonvolatile memory as well. Such a requirement leads to the reservation of a control memory block for hosting all the pointers. In order to reduce the number of update operations on the NVM it is possible to investigate the possibility of loading such a block of information into RAM at the beginning of the application operations and to keep up-to-date information there. Immediately before terminating the application, the memory block with the pointers is updated, dumping the information stored in RAM.

## 7. EXPERIMENTAL EVIDENCE

A simulator has been developed, modeling the CPU, RAM, and Flash memories, and the system bus, to evaluate the behavior of the proposed data structures and their impact on the parameters determining performance, power consumption, and response time. More in detail, the user can characterize the system model in terms of the following:

—architectural model;
  —Data and address bus;
—memory model:
  —Flash memory block size/number;
  —RAM memory size;
—data model:
  —relation cardinality,
  —record size;
—Model of the operations performed on the data:
  —number of select/delete/update operations,
  —Data set (random generated/user provided).

The evaluation parameters taken into consideration are as follows:

—number of read/written bits (related to response time/power consumption),
—number of bits on the data bus (related to power consumption),
—number of changing bits on bus (related to power consumption),
—number of block erasures (related to response time/power consumption), and
—area.

Once the architecture and data structure have been selected, it is possible to simulate the performance and costs of the adopted data structure either on randomly generated or on user-defined input data. The analysis of the experimental results is reported in the next sections, for a complete evaluation of the proposed approach.

### 7.1 Benefits and Costs Evaluation When Using *deleted_bit*

The introduction of *deleted_bit* improves performance but introduces costs in terms of space overhead and computational complexity (and consequently increases response time). Space overhead is *deleted_bit* itself, one for each record, while computational overhead refers to the time necessary to control *deleted_bit* in order to determine if the record needs to be handled or not. A set of experiments has been carried out to compare the traditional heap and circular list implementations to the proposed implementations adopting *deleted_bit*.

The experimental setup for this comparison adopted for all the analyses we carried out has been built by randomly generating the data set, and by applying a mixed sequence of access operations to simulate a real workload. For each emulated situation 50 different simulations have been repeated, finally reporting the mean value. Figure 13 shows the results for the heap relation case.

The graph in Figure 13(a) shows the average probability of block erasures that occurred during a sequence of 100 operations inserting, deleting, updating and selecting data. The graph in Figure 13(b) shows the number of block erasures occurring during a single simulation in the traditional and in the *deleted_bit* case. The graphs in Figures 13(c) show the number of read/write operations on the Flash memory for the same sequence of operations and the average read/written kbytes from Flash in order to perform such operations, respectively.

As expected, in the traditional implementation, each delete and update operation causes a block erasure, and the situation does not change during time, since the Flash memory block only contains valid records. Things are different in the *deleted_bit* implementation. At the beginning, record deletion and update are performed by invalidating the current record and, in the case of an update, the insertion of the modified record. This behavior does not force any block erasure until the moment when there are no spare records in the block. At that point, a block erasure occurs, a garbage-collection-like operation occurs, and the memory block is left with only valid records. The probability that this event occurs before the first 50 to 60 operations is zero with respect to the simulation results.

Clearly this behavior somewhat depends on the data set used and the sequence of operations, but the use of 50 different simulations should limit their influence. The graph of the average probability does not show that, once the block erasure occurs, the probability goes back to zero until the subsequent block erasure, exposing a quasiperiodical behavior. Therefore, the two traces of block erasures are reported in the middle graph, to better highlight such a behavior. In order to expose such a periodical occurrence in the case of the heap implemented with the *deleted_bit* approach, the test sequence was composed of 400 operations.

As can be noted, the introduction of *deleted_bit* reduces both the amount of accessed data and the number of block erasures, thus leading to a performance improvement and to reduced power consumption. Specifically, for the set of reported experiments, there was an average reduction of 15% for read operation and of 27% in write operations when using *deleted_bit*.

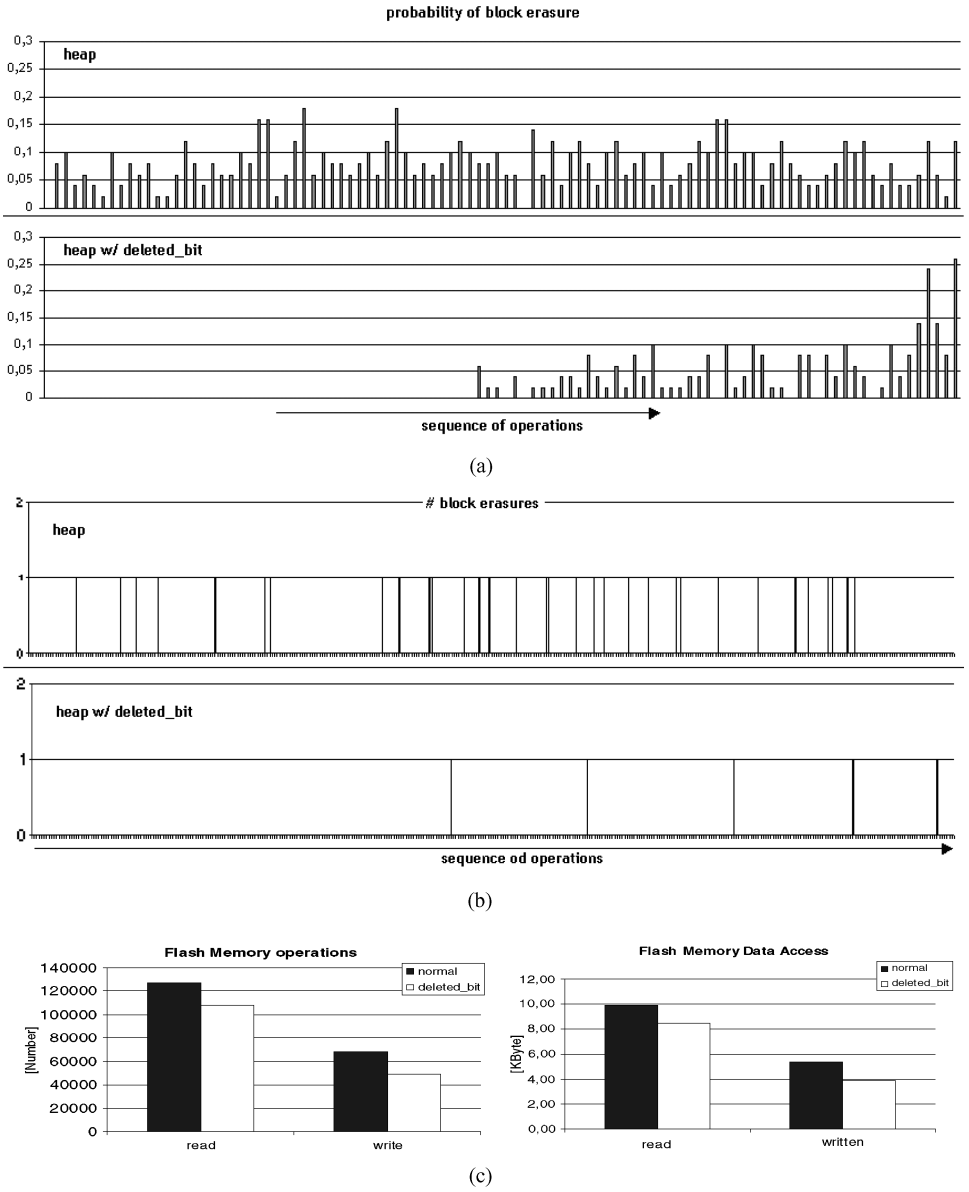


Fig. 13. Performance impact due to the introduction of *deleted_bit* with a heap relation.

Simulation has also been carried out with the distributed dummy physical implementation, without achieving significant results: the number of erasures was comparable to that of the traditional implementation, and sometimes worse. This is the result we expected: maintaining an ordered relation is expensive and only in exceptional situations can such costs be balanced by block erasures caused by a highly volatile heap relation.

Table IV. Access Timing Costs for the Different Sorted Data Policies

| | Adjacent dummy/No dummy | Distributed dummy |
|---|---|---|
| Scan | $B \times TR \times (FRR + C)$ | $B \times R \times (FRR + C)$ |
| Search eq. | $2 \times FRR \times Log_2\ B + C \times Log_2\ TR$ | $\frac{1}{2}B \times R \times (FRR + C)$ |
| Search range | $T_{SEQ\_EQ} + (FRR + RWR) \times n\_r$ | $B \times R \times (FRR\ + C) + (FRR + RWR) \times n\_r$ |
| Insert | $P_{CD} \times (FRR \times TR + C + FWR) + (1–P_{CD}) \times T_{DE}$ | $P_{DD} \times (FRR \times R + C + FWR) + (1 - P_{DD}) \times T_{DE}$ |
| Delete | $T_{SEQ\_EQ} + (FRR + FWR + RRR + RWR) \times R + C + BE$ | $T_{INS}$ |
| Update | $T_{DE}$ | $T_{SEQ\_EQ} + (FRR + FWR + RRR + RWR) \times R + C + BE$ |

### 7.2 Benefits and Costs Evaluation When Using Dummy Records

The adoption of dummy records reduces time and power consumption when introducing new records in a sorted relation, causing possible performance degradation in the search task due to the presence of nonsignificant records (the dummy ones) that nevertheless need to be recognized as such. Table IV shows a comparison of the costs with respect to the three physical data organizations discussed above. A few additional notations are introduced, to distinguish between valid, deleted and dummy records, and precisely:

—*DR*: number of deleted records;

—*VR*: number of valid records;

—*TR*: total number of programmed records (valid or deleted).

It is worth noting that when a relation fits completely on a single block only the distributed and no dummy organizations need to be considered, since the adjacent dummy organization provides the same performance as the no dummy one.

These formulas do not include the pointer to the last valid record (adjacent dummy policy) and the *valid_bit* (distributed dummy policy) management contributions, considering that such data are loaded into the CPU before processing the records, thus requiring a unique additional reading operation from the Flash memory. Nevertheless, such contributions have been explicitly taken into account during simulation, where the count of the number of read and written bits also includes access cost for control information.

The two dummy records management policies have been simulated in order to compare their performance; results for the search and insert operations are shown in Figures 14(a) 14(b).

There is still a final consideration: there are two phases in the life of a smart card database: an initial transient phase, when data is mainly inserted and updated, and a second—more stable—phase, where data is mainly accessed for retrieval. As a consequence, the suitability of a data structure for a given relation may change over time, since the expected frequency of the operations characterizing the two phases would lead to the selection of different data structures and of their related algorithms.

Power consumption maintains a significant role at all times. When considering the first transient phase, data structures and algorithms are targeted to

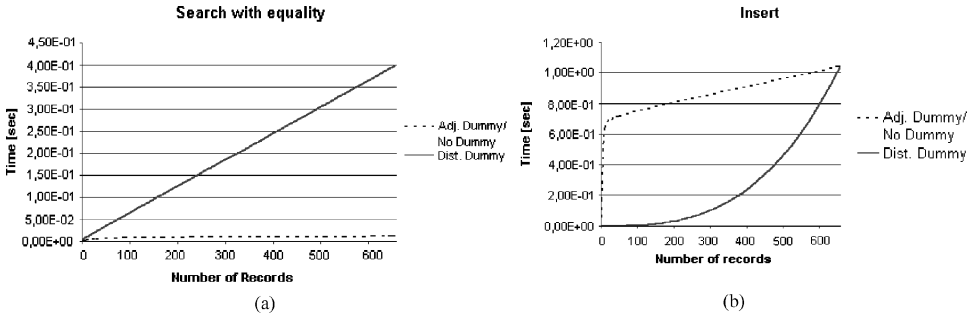


Fig. 14. Simulation of the search with equality and insert times as a function of the number of records in a block.

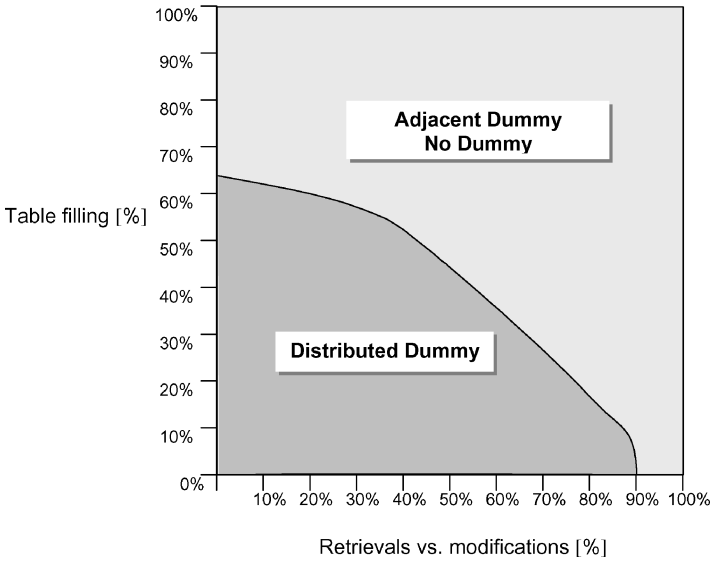


Fig. 15. Suitability areas for the considered policies when analyzed on a single block.

limit the number of memory block erasures, identifying a satisfying tradeoff with respect to space occupation. In the second phase, efficiency in retrieval and data size are predominant aspects. Figure 15 shows a comparison between the different dummy records allocation policies with respect to the mix of operations in the workload.

Since the distributed dummy records approach seems to perform significantly better during the first initial phase, we carried out a set of experiments to highlight this situation, by comparing the normal (traditional), the *deleted_bit* (and no dummy), and the distributed dummy solutions, as we already showed in Section 6.3.2. The results are reported in Figure 16, where a sequence of 250 operations has been applied to the three physical data structures, starting from an empty memory. One hundred simulations for each setup have been performed to achieve the values of the probability of block erasures. A different

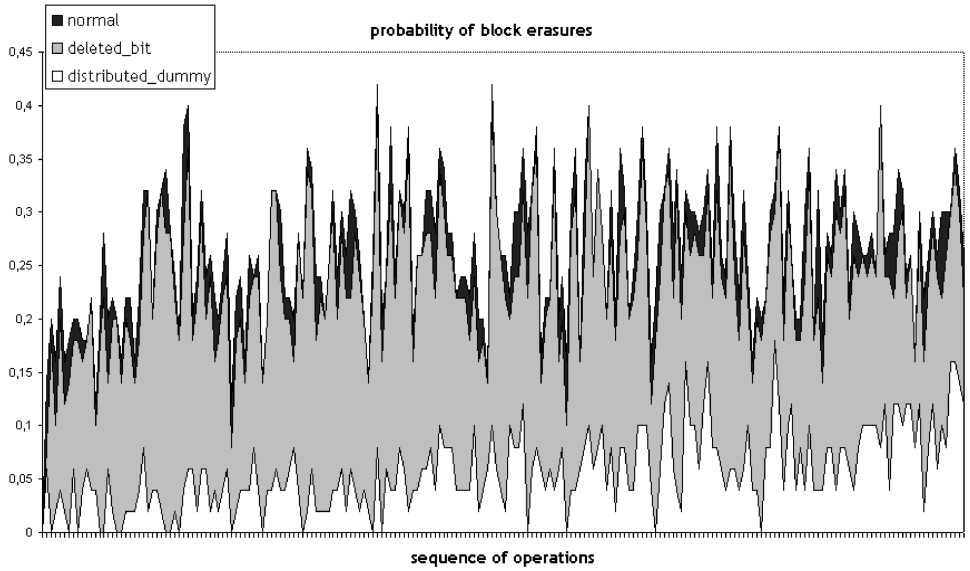


Fig. 16. Performance comparison for a mixed sequence of operations: the *distributed dummy* solution is well below the normal and *deleted_bit* solutions, significantly reducing the number of block erasures required to maintain a sorted relation.

representation of the results with respect to the one used in Figure 11 has been adopted to better point out the differences between performances. For the sake of clarity, the *adjacent dummy* approach is not reported leading to results similar to the *deleted_bit* ones.

As can be seen in Figure 16, the distributed dummy requires a lower number of block erasures (bottom trend line), followed by the *deleted_bit* and normal implementations, during the entire sequence of operations. This will lead to reduced power consumption and response times in performing the desired operations.

It is worth noting that the *deleted_bit* solution, which significantly improves a heap relation, does not provide the same relevant advantages in the case of a sorted relation. As has already been pointed out, in fact, the maintenance of the ordering forces many modifications in the relation records, thus requiring frequent block erasures. Yet, when the volatility of data lowers, that is, in the second phase of the database life, when data is mainly retrieved and seldom modified, the use of *deleted_bit* could substitute for the dummy solution in order to lower costs without requiring too many block erasures.

### 7.3 Overall Evaluation

Each one of the proposed logical and physical data structures has been characterized in terms of costs and benefits, in order to be able to compare the different solutions for each relation while trying to identify the most promising database implementation. Each of the significant elements has been introduced as a contribution to a cost function that computes the cost of each implementation and

identifies for each relation the most convenient solution, exploring the entire solution space.

Such a cost function is also used to evaluate the performance and costs of the possible approaches also considering the expected volatility of relations (the number of changing records within a relation during time) to explore the possibility of adopting different physical data structures based on the "evolutionary phase" of the relation, initial/startup, or stable phase.

## 8. CONCLUSIONS AND FUTURE DEVELOPMENTS

This research proposes a methodology for smart card database design using Flash-EEPROM storage; the process is strongly driven by the technological issues, which impose a very careful design on the logical and physical data structures in order to meet the constraints the smart card architecture introduces, and to provide satisfactory performance.

In this paper we concentrated on the logical-physical smart card database design phase, with attention to the most appropriate data structures and algorithms that must be made available for the database designer to choose. Our analysis pointed out that there are several factors guiding the selection of the most suitable data structures when deriving the physical database design from the annotated logical schema: power consumption, endurance, performance, physical size requirements.

We examined a number of data structures with respect to the insertion, search with equality, search with range selection, and delete and update operations, analyzing their features in terms of time performance and memory size. Furthermore, in order to reduce block erasures that, in Flash-EEPROMs, are lengthy, power-consuming, and life-shortening operations, we focused on the insertion operation in sorted data structures. We introduced a set of dummy records and compared two different policies for their allocation.

We found that the performance of the two allocation policies differ with respect to the mix of operation types. In particular, the distributed dummy records policy is well suited at times when the database contents undergo deep changes, whereas the no dummy or adjacent dummy records policies are suited when most of the operation involves retrieval queries. Simulation results supporting the described behavior are shown.

Our proposal focuses on the logical and physical data structures designed to support a database approach for managing data stored in a smart card. Future work on the methodology will address the following aspects:

—the "upper parts" of the methodology tree, in order to find design, fragmentation and allocation criteria for the different logical units (e.g., tables) of the smart card database;

—ad hoc architectural enhancements, with particular attention to the smart card processor, to the instruction set, to power consumption, to the possibility of implementing different storage technologies on the same card in order to benefit from their peculiarities, and to the analysis of other data management policies for performance optimization.

We will also investigate the following other interesting research issues related to smart card technology:

—transaction management to guarantee data consistency, especially useful for situations where a card is disconnected (e.g. extracted from the reader) before committing the transaction;

—synchronization between the smart card database and the central one, hosted by a server, when smart card data processing is performed off-line.

It is important to notice that the proposed design methodology can be adopted independently of these last two points, since many applications do not require a central server database at all, while others do not need implementation of a full transaction environment, only requiring a minimal amount of data consistency.

## REFERENCES


ATZENI, P., CERI, S., PARABOSCHI, S., AND TORLONE, R. 2000. *Database Systems*. McGraw-Hill, New York, NY.

BOBINEAU, C., BOUGANIM, L., PUCHERAL, P., AND VALDURIEZ, P. 2000. PicoDBMS: Scaling down database techniques for smart card. In *Proceedings of the 26th International Conference on Very Large Databases*. 11–20.

BOLCHINI, C. AND SCHREIBER, F. A. 2002. Smart card embedded information systems: A methodology for privacy oriented architectural design. *Data & Knowl. Eng. 41*, 2-3, 159–182.

BUTLER, M. J., HARTEL, P. H., DE JONG, E., AND LONGLEY, M. 2001. Transacted memory for smart cards. In *Proceedings FME 2001, Formal Methods for Increasing Software Productivity*. 478–499.

CERI, S. AND PELAGATTI, G. 1984. *Distributed Databases: Principles and Systems*. McGraw-Hill, New York, NY.

EICH M. H. 1992. Main memory databases: Current and future research issues. *IEEE Trans. Knowl. Data. Eng. 4*, 6, 507–508.

ELMASRI, R. AND NAVATHE, S. H. 1994. *Fundamental of Database Systems*. 2nd ed. Benjamin Cummings, Redwood City, CA.

GARCIA-MOLINA, H. AND SALEM, K. 1992. Main memory database systems: An overview. *IEEE Trans. Knowl. Data. Eng. 4*, 6, 509–516.

ITGOV. 2002. Smart card adoption for ID application in the Italian Government. Available online at http://www.innovazione.gov.it/ita/comunicati/2002_02_08cie.shtml.

RAMAKRISHNAN, R. AND GEHRKE, J. 2000. *Database Management Systems*, 2nd ed. McGraw-Hill, New York, NY.

RANKL, W. and EWFFING, W. 1999. *Smart Card Handbook*, 2nd ed. Wiley, New York, NY.

STOCKDILL, R. 2002. STMicroelctronics debuts word's most advance smart card memory technology. STMicroelectronics Technical Press Relations, Carrollton, TX. Available online at http://www.st.com/stonline/press/news/year2002/p1249.htm.

SUN MICROSYSTEMS. 1999. JavaCard 2.1 Application Programming Interface Specification, JavaSoft Documentation. Sun Microsystems, Santa Clara, CA.

SUTHERLAND, J. and VAN DEN HEUVEL, W. J. 2002. Enterprise application integration and complex adaptive systems. *Comm. of ACM*, *45*, 10, 59–64.

TAMER, Ö. M. AND VALDURIEZ, P. 1991. *Principles of Distributed Database Systems*. Prentice Hall, Englewood Cliffs, NJ.

WIEDERHOLD, G. 1987. *File Organization for Database Design*. McGraw Hill, New York, NY.


Received August 2002; revised January 2003; accepted May 2003