A Context-Aware Methodology for Very Small Data Base Design

C. Bolchini, F. A. Schreiber and L. Tanca
Politecnico di Milano, Dipartimento di Elettronica e Informazione
Email: bolchini,schreiber,tanca@elet.polimi.it

***Abstract*— The design of a Data Base to be resident on portable devices and embedded processors for professional systems requires considering both the device memory peculiarities and the mobility aspects, which are an essential feature of the embedded applications. Moreover, these devices are often part of a larger Information System, comprising fixed and mobile resources. We propose a complete methodology for designing Very Small Data Bases, from the identification of the device resident portions down to the choice of the physical data structure, optimizing the cost and power consumption of the Flash memory, which – in the greatest generality – constitutes the permanent storage of the device.**

I. Introduction and Motivation

The use of handheld devices, such as Smart Cards, Portable Data Assistants (PDA), Palm PCs and Cell Phones, as intelligent portable terminals of some Information Systems is being widely discussed in recent times [1], [2], [3], [4], [5]. Features required by portable devices in order to manage both local and remote data range from very simple file system functions to a full set of database management capabilities, including some ACID transactions properties. Examples of databases for very small devices – henceforth called *Very Small Data Bases* (VSDB) – are *personal (micro)information systems* such as the so-called *citizen's card* or the *personal medical card*, the *personal financial database*, storing the device owner's stock portfolio, and the *personal travel database*, recording all the travel information considered interesting by the device owner.

A typical architecture for a modern Information System including portable and small devices is a variation of the distributed client-server paradigm. Here, the server power can range from a small workstation to a very large mainframe, and there is no need for the server to be unique; indeed, in the most general case the microdevice will be served, in turn, by one of a number of hosts, possibly having different hardware characteristics and being located at sites very far apart from each other.

The (portable) client can range from a Smart Card to a powerful PC, thus influencing the amount and structure of the data that will be stored locally; traditional applications must be scaled down in order to cope with the limited resources of such devices in terms of processing power and energy autonomy, so that these issues must be considered at an early stage during database design [6]. Mobility (thus time and space) is a very critical issue in this perspective, since the designer has to limit the data immediately available on the device to those strictly related *to the specific context the user is experiencing at a certain specific time*.

Our project proposes a complete design methodology for VSDBs, based on a modification of the classical three levels of the ANSI-SPARC model w.r.t. two main points: a) at the very early steps of conceptual database design, where application features are taken into account, we analyze and introduce context related information: by examining application and context together we determine the VSDB *ambient*, which is *the set of personal and environmental characteristics determining the portion of data that must be stored locally on the device*; b) due to the reduced capacity, limited power and specific operation modes of the storage medium (most often EEPROM Flash memory), physical storage information is taken into account during the logical design process. This phase of logical design, where tables are analyzed w.r.t. the envisaged access types, information volatility, user permissions and protection mechanisms is called the *logistic phase*, since it mainly deals with logistic aspects of data storage.

In the latest decades, all the burden of the design of physical data structures, along with the main issues related to optimal data allocation and access, have been in charge of the DBMS designers; in our project we propose that the logistic phase be taken care of as a joint effort of the database designer together with the DBMS designer. This is accomplished by allowing the database designer to specify to a quite relevant extent his/her desiderata w.r.t. the data structures to be employed for the physical storage of database relations [1].

The rest of the paper is organized as follows. Section II provides an overview of the entire methodology. Section III details the steps of the design methodology leading to the definitions of the data to be stored on the portable device, based on the significant context elements. Section IV proposes the logical and physical data structures allowing a satisfactory performance/cost trade-off in accessing stored data; the last section conlcudes the paper with an evaluation of the proposed data structures w.r.t. performance and power consumption.

II. The complete methodology for VSDB design

Fig. 1 briefly describes the entire framework for designing very small databases for embedded processors and portable devices.

The methodology includes a left-hand track, which takes care of the context-aware conceptual and initial logical design,

This reasearch is partially supported by the FIRB project MAIS

[1] Throughout the paper we will refer to the relational model of data; variants for different data models are immediate.

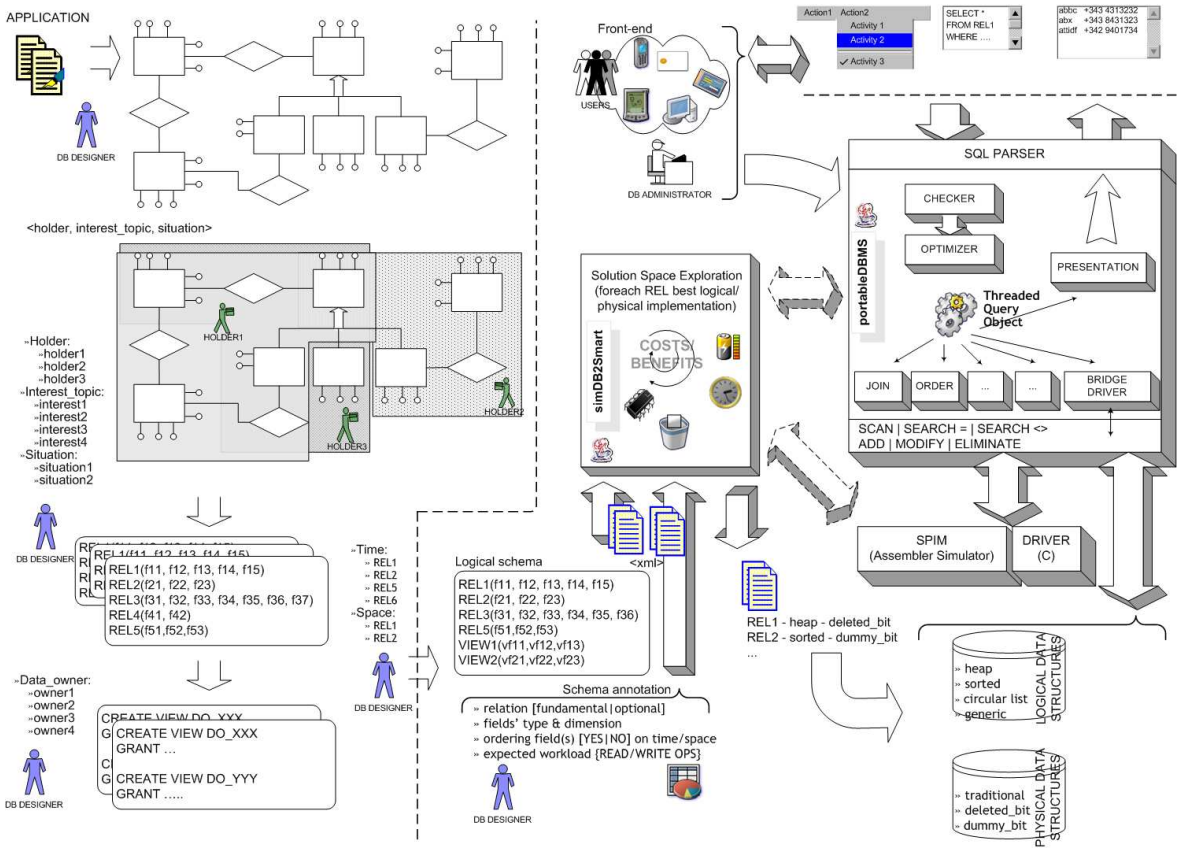

Fig. 1. The complete framework of the Very Small Database design methodology

and of a right-hand part dealing with the logistic/physical phase. The left part is in charge of the database designer, possibly supported by a case tool; here, the relevant information is chosen and modelled by the usual techniques for conceptual database design, but brand-new methodological issues are raised by the need of sorting out the local, "first need" information – stored on the mobile device and defining the VSDB ambient – from the other fragments to be stored in other sites of the Information System and queried only when a connection is available.

The center and right parts concern the DBMS data structures and access methods, along with issues related to security/privacy. Here, physical data structures are chosen by the database designer in collaboration with the DBMS, and with the support of a workload simulator that accepts the designer's estimation on data accesses and provides suggestion on the most appropriate data structures. We are currently developing such a full featured, modular DBMS for portable devices, including the physical data structures defined in this project, along with the case tool that directs the designer's choices in order to achieve the best performance in terms of: a) access time, b) amount of required storage, and c) power consumption. For a detailed description of this part refer to [6] and [7].

III. The Context-Aware Conceptual and Logical Phases

The conceptual phase can be decomposed into the following steps:

1. Application information modelling: this is done by the usual techniques for conceptual database design, taking into account *all the information relevant to the application at hand*, regardless of the target storage media. Indeed, the VSDB design must be merged within the design of the distributed database it belongs to.

2. choice of the *analysis dimensions*: analysis dimensions provide the different perspectives the mobile device is viewed from, and are used to set out the VSDB *ambient*. Here we consider some intuitive dimensions, which can be integrated with extra ones, or dismissed where not appropriate:

- The *holder* dimension refers to the type of users carrying the microdevice, whose views over the whole information system can be quite different. For example, in a medical application, `doctors` will hold information about all their patients, while `patients` will only hold information related to themselves, maybe at a finer level of detail.
- The *interest topic* dimension refers to the particular aspect/subject the user might be interested in at a certain moment. In the medical care case, topics might include `prescriptions` or `chronic diseases`. In a tourist guide application this dimension might refer to the choice of information about the city entertainment, or

about restaurants, etc.

- The *situation* dimension refers to the fact that during the device life the user may wish to access different views of the data, being able to perform different operations. For instance, in the personal medical information system, examples of situations are the `regular`, i.e. ordinary patient's state, as opposed to a temporary `hospitalized` situation.

As an output of this step, the identified dimensions are collected to form the *ambient array model*, which drives the actual choice of the information to be kept on the microdevice; the dimensions used in this paper form a three-positions array model:

<holder, interest_topic, situation>

3. dimensions conceptual design: In this phase, one conceptual schema is built for each dimension value; e.g., w.r.t. the *holder* dimension, in the medical care application we build one view for the `patient`, one for the `doctor` and one for each of the possible other values of this dimension (e.g. hospital administrator). Here a reconciliation work must be done; the conceptual schemata produced by analyzing the application from the viewpoints of the various dimension values must be reconciled with the Global Conceptual Schema, in order for the former to be perceived as views over the latter.

4. conceptual view merge: here the so called *array schemata*, or *chunks*, are derived from the array model, by instantiating the dimensions; examples of chunks in the Medical Care Database (MCDB) case are:

<patient, chronic_diseases, hospital>
This chunk contains all the information needed by a patient at the hospital w.r.t. his/her chronic diseases (if any).

<patient, prescriptions, regular>
This chunk contains all the information needed by a patient in a normal situation, w.r.t. his/her prescriptions (if any).

<doctor, prescriptions, regular>
This chunk contains all the information needed by a doctor regarding all his/her regular patients' prescriptions.

Chunk derivation must be done with an eye to their actual significance: indeed, only some of the possible combinations of dimension values make sense. For example, the chunk
<doctor, accounting, hospital>
makes little sense in view of the applications semantics.

As a conclusion of this phase we assemble chunks in order to define which is the information that must be stored on one single device. For example, normally a patient's smart card will contain all the chunks related to the patient's (regular) situation plus those related to his/her chronic diseases (such as allergies etc.) and prescriptions. When the patient is at the hospital, the "regular" chunks will be removed to leave room for the "hospital" ones. However, if the device is more capacious (for example a PDA) the designer might decide to leave on all the chunks related to different situations at all times.

5. logical view design: here different activities are carried out:

- *logical design of the global database*: examples of tables for the MCDB are:

```
PATIENT(SSN, FName, LName, Sex, BirthD,
DeathD, Address, City, State, Zip, Phone,
BloodType, Notes, MCUID, Booklet, DocID)
MEDICAL_CARE_UNIT(ID, Name, Address, City,
State, Zip, Phone, Type)
SERVICE(ID, Name, Tipology, Difficulty,
Period)
USES(MCUID, SERVICEID)
PRESCRIPTION(SSN, DRUGID, Mode, Dosage,
Administration, StartDate, EndDate,
Comments)
DRUG(ID, Name, Posology, Ingredients,
SideEffects, Manufacturer, Comments)
DRUG_IN_PHARMACY(DRUGID, PHARID)
PHARMACY(ID, Name, Address, City, State,
Zip, Phone, OpeningHrs)
```

- *logical chunk production*: the chunks are defined as logical views over the global logical database produced above; for example, the chunk <patient, prescriptions, hospital> is defined as:

```
CREATE VIEW PAT-PRESC-HOSP AS
SELECT P.SSN, P.FName, P.LName,
DRUG.Name AS DrugName, Posology,
SideEffects, Mode, Dosage, Adminis-
tration, StartDate, EndDate, Comments,
MCU.Name, MCU.Address, MCU.City,
MCU.State, MCU.Zip, MCU.Phone,
MCU.Type
FROM PATIENT P, DRUG, PRESCRIPTION PR,
MEDICAL_CARE_UNIT MCU
WHERE P.SSN = PR.SSN AND PR.DRUGID
= DRUG.ID AND P.MCUID = MCU.ID AND
MCU.Type = ''hospital''
```

- *chunk instantiation*: here the views for the chunk instances are produced. A chunk instance relates to one specific instance of a dimension value. An instance of the `patient` value of the holder dimension is "John Doe" and one of the `chronic_disease` value of the interest topic dimension is "Diabetes". Thus consider the following view instantiation:

```
SELECT * FROM PAT-PRESC-HOSP WHERE
SSN = 930029747 AND MCUID.NAME = ''Mt.
Sinai'''
```

- introduction of the so called *logistic dimensions*, i.e. dimensions which do not influence the actual database design but only the logistic phase:

  - The *data ownership* dimension concerns `read`, `update`, `delete`, and `insert` access rights to the VSDB information, which might be different depending on the user categories. Note that access rights must be analyzed w.r.t. *actors* in general different from the device holders: in the MCDB example a patient's doctor has modify right on the patient's prescriptions; the patient, in turn, may read his/her prescribed drugs, but cannot modify them. The data ownership dimension does not delimit the boundaries of the available information; thus it is used to identify

permission views but not for ambient identification.
- The *time* dimension refers to the information lifespan the VSDB tables must store: for example, one could save the whole medical history of the patients in the fixed machine of their doctor, keeping only the last month's data on the device itself.
- The *space* dimension concerns the physical area of interest. For example, a patient resident in Milan is interested, during a work trip in Genoa, to all medical facilities in that city, disregarding the information about the other such facilities located in other cities.

In a context-aware system, time and space are usually evaluated w.r.t. the actual position of the device ("*now*" and "*here*"), which can be obtained from the system clock and from a positioning system; they determine further tailoring of the chunk aggregations that have been allocated on one device, by means of logical views that limit the information to that pertaining to the current context. For example, w.r.t. *time* and *space*, the following chunck is derived:

```
SELECT * FROM PAT-PRESC-HOSP
WHERE SSN = 930029747 AND StartDate <
Now() AND EndDate > Now() AND City =
Here()
```

IV. Logistic and physical data structures

The right-side branch of our methodology aims at identifying the most convenient logical and physical data structures the DBMS must make available for the VSDB designer. We briefly discuss the technology behind the class of devices we are considering.

A. Technology issues

Currently, the permanent storage medium for portable devices is Flash EEPROM. Using this technology, `write` operations can only be performed if the target location has either never been written before, or has been previously erased. Erasure can only be done at *block* level; `read` and `write` operations work at the single word granularity. Endurance is a critical factor as well; each erasure has an impact on the life of the device whose reliability can be jeopardized. Thus, a DBMS using a Flash memory must reduce the number of data modifications.

B. Data structures for very small databases

Classical, indexed data structures are often inappropriate for VSDB's; indeed, the search needs we have within the small tables stored is often not worth the overload required for managing and maintaining indexes, which are proposed only in the case of tables with large cardinality and special needs for multi key search [8]. Instead, we propose what we call *logistic data structures*, i.e. intermediate data structures that should be chosen to implement each database relation. In the description of such data structures we refer to our MCDB running example, (see Fig. 2), w.r.t. the holder point of view.

A **Heap** relation is used to store a small number of records (generally less than 10), unsorted, typically accessed by scanning all records when looking for a specific one; example of this kind of data is the `DOCTOR` relation in the MCDB database, storing the entries of the different doctors that usually cure the patient.

Sorted relations, characterized by a medium (≅100 - ≅1000 records) cardinality, are used to store information typically accessed by the sort key. The idea here is to impose an upper bound to the number of records that can be inserted based on the complete size of the (fragment of) table. Once the upper bound is reached, the user will have to delete (or store externally) a record before adding a new one. With respect to the running example, the `DRUG` relation can be managed by means of sorted data.

Circular list relations, characterized by a medium cardinality as well, are again suitable to manage a fixed number of log data, sorted by date/time; in this case, once the maximum number of records is reached, the next new record will substitute the oldest one. In the MCDB example, data logs of the patient's last medical exams can be stored by means of circular lists.

Multi-index relations are used to manage generic data, typically when the need is to access efficiently large relations by multiple keys. This is the only data structure we propose which resembles the classical data structures used in DBMS's.

Note that the availability of bigger Flash memories will allow the storage of a larger amount of data in the very same data structures, with no need to move toward more complex (indexed) ones. Indeed, the update of stored values being one of the most expensive tasks w.r.t. Flash memories, the usage of frequently updated pointers (required by indexed structures) is often not advisable, being harmful to both memory endurance and power consumption. On the other hand, a larger memory capacity will require less frequent erasures, increasing system performance and endurance.

Our methodology requires the designer to tag each table of the chunks we want to include in the VSDB under consideration with the following information:

- Tuple length (in bytes) and Expected Relation Cardinality (eventually specifying an upper bound to the number of records to be allowed (e.g. 5 records for the `PREGNANCY` relation from the $holder$ = 'PATIENT' point of view).
- Presence of a sorting field, specifying if the field is a time field leading to a log-like file
- Expected composition of the set of operations on data: *insert*/*delete*/*update*/*select*, the last one further classified in a full select (*scan*), select with equality (*equal*), and select with range (*range*).

The expected mix refers to the relative frequency of operations. For instance, consider the `DRUG` relation; the user can say that the dominant operation will be the *insert*, usually no *delete* and very few *update*. The other common operation is *select*, assuming an equal distribution in the three identified selection schemas. Here, the simulator comes into play, issuing an indication of the data structures the DBMS must employ for the required relations. Data structure implementation is discussed in the following subsection.

	Length	Cardinality	Limited	Key	Ordered	Access type frequency						Data structure ***
						INSERT	DELETE	UPDATE	SELECT			
									scan	equal	range	
P_PeronalInfo	287	1	YES	SSN	NO	NEVER	NEVER	LOW	HIGH	HIGH	HIGH	H
P_DoctorInfo	83	1	YES	N/A	NO	LOW	LOW	LOW	MEDIUM	MEDIUM	MEDIUM	H
P_Pregnancy	20	1	YES	SSN	NO	LOW	LOW	LOW	HIGH	HIGH	HIGH	H
P_Intollerance	30	80	NO	DrugID	YES	LOW	LOW	LOW	HIGH	HIGH	HIGH	S
P_Regular_Use	150	30	NO	DrugID	YES	LOW	LOW	LOW	HIGH	HIGH	HIGH	S
P_Anomalies	50	20	YES	StartD,EndD	YES	HIGH	NEVER	LOW	HIGH	LOW	LOW	CL
P_Patologies	90	20	YES	StartD,EndD	YES	MEDIUM	NEVER	LOW	MEDIUM	LOW	LOW	CL
P_Trauma_Injuries	120	20	YES	StartD,EndD	YES	MEDIUM	NEVER	LOW	MEDIUM	LOW	LOW	CL
P_Allergies	100	20	YES	StartD,EndD	YES	MEDIUM	NEVER	LOW	MEDIUM	LOW	LOW	CL
P_UsefulCenters	167	20	YES	TreatID	YES	HIGH	HIGH	LOW	MEDIUM	MEDIUM	MEDIUM	S

*** H=Heap, S=Sorted List, CL=Circular List

Fig. 2. The Medical Care DataBase: Table Annotation and adopted data structures.

C. Physical implementation: memory management

The technology behind Flash memories and their constraint on data erasure introduces a significant impact on the *delete* and *update* operations, also affecting *insert* operations in sorted relations. In fact, when the stored data need to be modified, at least one (but possibly many) memory block needs to be re-written, implicitly requiring a copy of its content in the RAM, an erasure of the Flash block and a write-back, from RAM to Flash, of the modified content (*dump*/*erase*/*restore* DER sequence). Do note that the DER sequence deeply affects performance (due to the time required for the data "dump"), power consumption and storage endurance.

In order to reduce the number of modifications requiring Flash memory erasure, an additional information is associated with each record:

- **valid bit** to indicate that the record has been programmed;
- **deleted bit** to indicate that the record has been logically (but not physically) deleted.

The use of the *valid bit* is essential when memory is managed in a non-sequential fashion; in particular, the valid bit implements a "distributed" control since each valid record is directly distinguishable from the others, while an end-address (register) implements a "concentrated" control, since it univocally identifies the end of the record list. The concentrated control is a space-aware but energy-time consuming approach since the end-address value needs to be updated every time the list is modified with the *DER* sequence.

The *deleted bit* is used to allow the system to reduce the number of Flash memory erasures by marking the corresponding record and deferring the physical expunging to a later moment. The *deleted bit*, coupled with a not-sequential management of the physical memory leads to reduce the necessity to erase blocks, at the cost of an increase in the amount of required memory and a more complex management policy, as discussed in the following.

When dealing with data sorted with respect to a field, insert and delete operations have a significant overhead due to the necessity to maintain the data ordered; furthermore, if the relation data is distributed over several blocks, the operation might affect multiple blocks. The proposed data structure aims at a) confining block involvement in data manipulation and b) minimizing block erasure. This goal is achieved by introducing a number of dummy records per block (Fig. 3a); such records may be either localized at the end of the block or distributed through it by means of a hashing function, so that future insertions do not always cause a re-organization of previously introduced records (Fig. 3b). The hashing function may be implemented either in software or in hardware; in this case the *valid bit* is mandatory to determine which records are actually programmed and which are not. The use of the concentrated dummy records aims at preventing multiple blocks involvement when records need to be shifted up or down following a delete or insert operation (intra-block erasures). The distributed dummy records solution further limits inter-block erasures.

The *deleted bit* has the same function as above (Fig. 3c).

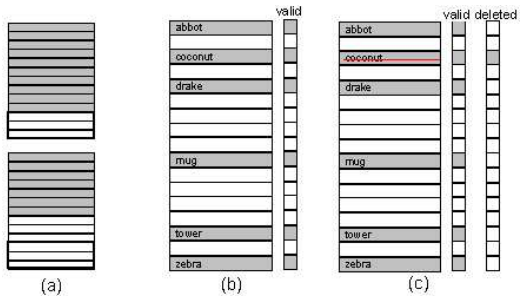

Fig. 3. Use of dummy records a) concentrated at the end of the block - bold frame - or b) distributed through the block. c) Use of distributed dummy records and *deleted bit*

The combined use of dummy records and *deleted bit* is useful in sorted relations, the use of the *deleted bit* technique alone is suitable for circular lists and possibly heap relations, at the cost of additional space requirement w.r.t. the minimum possible amount of memory. The database annotation allows the association of the most convenient data structure, as discussed above. Current research is investigating the impact and costs of such data structures on generic relations not falling in previous cases.

V. Structures evaluation and concluding remarks

The proposed data structures together with the possible physical implementation, have been analyzed and compared

TABLE I

EXPERIMENTAL RESULTS: OCCURRED BLOCK ERASURES AND TRANSMITTED BYTES ON THE SYSTEM BUS W.R.T. "THE NAIVE", NO *deleted bit*, NO DUMMY RECORDS SOLUTION.

Data structure	Strategy	Block Erasures			Transmitted Bits on Bus		
		10%-30%	40%-60%	70%-90%	10%-30%	40%-60%	70%-90%
Heap	Simple	1	1	1	1	1	1
	Deleted_bit	0	0.38	0.98	0.38	0.54	1.00
Sorted	Simple	1	1	1	1	1	1
	Deleted_bit	0.83	0.68	0.79	0.74	0.71	0.77
	Dummy adj.	0.83	0.51	0.44	0.74	0.57	0.45
	Dummy dist.	0.10	0.12	0.24	0.03	0.06	0.22
Circular List	Simple	1	1	1	1	1	1
	Deleted_bit	0	0	0.05	0.07	0.07	0.15

with a "naive" implementation, both w.r.t. performance and power consumption. Costs have been estimated in terms of the total amount of necessary memory due to the additional bits associated with each record, and the dummy records; algorithm complexity and the consequent execution requirements (time and power consumption) have been also estimated. The first results show that the combination of the two strategies - dummy records (requiring the use of the *valid bit*) and *deleted bit* - allows to reduce the average time to perform the operations on data and to significantly limit the number of erasures. It is worth noting that, with the proposed structures, *select* operations have an overhead due to the presence of the empty records that slow the scanning of the data structure. A simulator has been developed to evaluate all parameters and to compare different solutions; currently our effort is devoted to the analysis of generic relations and to the enhancement of the application of these data structures in different moments of the database life, depending on the volatility of data [7].

Tab. I reports a summarization of the experimental results carried out for evaluating the proposed data structures, for each relation. A synthetic workload made of operations evenly distributed between *insert*, *delete* and *update* was used, increasing memory occupation to analyze the impact on block erasures and the amount of information transmitted on the bus, two significant aspects affecting performance and power consumption. The setup for the experiments is: 4Kbyte Flash Memory blocks, a 32 KByte RAM and a 128 byte record size; then number of Flash blocks and the number of records per relation differ in the various situations, depending on the data structure being simulated.

The first part of Tab. I reports the ratio between the number of block erasures occurring when adopting the specified implementation and the number of block erasures when using a "naive" implementation. As an example, consider the *sorted* relation with a half full memory (50%): with the concentrated dummy solution the workload causes only half the number of block erasures w.r.t. the "naive" implementation. Similarly, the second part of the table reports the ratio between the number of bits transmitted on the data bus for each proposed approach w.r.t. the "naive" implementation.

As can be noted, the achieved results show an improvement in Flash memory access both in terms of erasure operations and read/written bits, elements affecting system performance and power consumption.

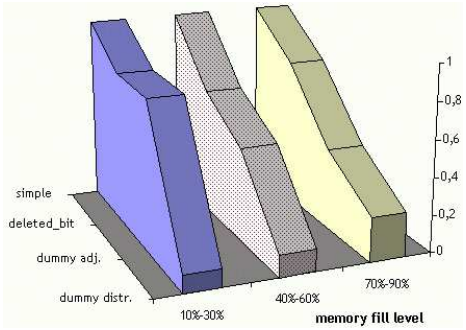

Fig. 4. A relative analysis of data access costs of various implementations of the *sorted* relation ("naive"=1).

Fig. 4 shows that enlarging storage capacity, while keeping the same amount of data, shifts cost and performance towards the values shown for a partially filled memory. Since absolute performance values vary among different vendors' products, in Tab. I and Fig. 4 we provide only relative figures to compare the different proposed logical and physical data structures.

Our plans for the future include the complete formalization of chunk definition and instantiation along with the development of a tool to support the DB designer in his/her tasks.

REFERENCES

[1] S. Banerjee et Al., "Rover: Scalable location-aware computing," *IEEE Computer*, vol. 35, no. 10, pp. 46–53, October 2002.

[2] D. L. Lee, W.-C. Lee, J. Xu and B. Zheng, "Data management in location-dependent information services," *IEEE Pervasive Computing*, vol. 1, no. 3, pp. 65–72, 2002.

[3] H. Mohanty, "Active and nomadic transactions in mobile databases," in *ADBIS-DASFSAA Symposium*, September 2000, pp. 99–107.

[4] B. N. Schilit, J. Trevor, D. M. Hilbert and T. K. Koh, "Web interaction using very small internet devices," *IEEE Computer*, vol. 35, no. 10, pp. 37–45, 2002.

[5] J. Sutherland and W.-J. van den Heuvel, "Enterprise application integration and complex adaptive systems," *Communications of the ACM*, vol. 45, no. 10, pp. 59–64, 2002.

[6] C. Bolchini and F. A. Schreiber, "Smart card embedded information systems: a methodology for privacy oriented architectural design," *Data and Knowledge Engineering*, vol. 41, no. 2-3, pp. 159–182, 2002.

[7] C. Bolchini, F. Salice, F. A. Schreiber and L. Tanca, "Logical and physical design issues for smart card databases," *ACM Trans. on Information Systems*, vol. 21, no. 3, pp. 254–285, 2003.

[8] C. Bobineau, L. Bouganim, P. Pucheral and P. Valduriez, "PicoDBMS: Scaling down database techniques for smart card," in *26th International Conference on Very Large Databases*, 2000, pp. 11–20.