

PoLiDBMS: Design and Prototype Implementation of a DBMS for Portable Devices^{*}

C. Bolchini, C. Curino, M. Giorgetta, A. Giusti, A. Miele,
F. A. Schreiber, L. Tanca

Dipartimento di Elettronica e Informazione - Politecnico di Milano

Abstract. Very Small DataBases (VSDB) is a methodology and a complete framework for database design and management in a complex environment where databases are distributed over different systems, from high-end servers to reduced-power portable devices. Within this framework the architecture of PoLiDBMS, a *Portable Light Database Management System* has been designed to be hosted on such portable devices, in order to efficiently manage the data stored in Flash EEPROM memory. A flexible and modular solution has been adopted with the aim of allowing the development of a system able to be customized in its features, depending on the needed functionality and the available processing power. The first prototype implementation provides all the elementary functionalities of a DBMS, supporting a reduced set of the SQL language that can be of interest in such a limited environment.

1 Introduction

Information systems running entirely or in part on portable devices are nowadays possible, owing to the availability of small physical devices with always improving computational and storage capabilities [1,2,3,4,5].

Unfortunately, software designed to run on portable devices must account for a number of design constraints, related to computational power, energy consumption, and persistent data storage issues; in particular the latter problem has a considerable impact on software implementations, because dependable and efficient persistent data storage is a key requirement for this class of applications. The currently commercially used EEPROM Flash memory technology does not provide support to these requirements, offering memories with bit/byte access granularity and erasure operations (needed to modify data) working on a per-block unit. Such technological constraints have a significant impact on performance, both in terms of response time and power consumption, the latter aspect being extremely important in battery-powered portable devices. Moreover, Flash memory blocks can only be erased a finite number of times (up to 100000) before becoming unusable.

^{*} This research is partially supported by the FIRB project MAIS.

These low-level considerations are the starting point of our work; we aim at improving Flash memory performance by using storage policies that offer simple record based read and write services and tend to minimize the number of required persistent memory block erasures. A number of different approaches have been defined to achieve enhanced memory management w.r.t. EEPROM Flash technology, which have led to the definition and implementation of different storage/access policies, each one representing a unique tradeoff between time and energy costs for write and read operations.

We designed and implemented our Data Base Management System (DBMS) from scratch, planning usage of such data storage routines to circumvent Flash memory limitations, and definitely aiming at portable devices and applications: to represent the aim of the project, the tool has been named Portable Light DataBase Management System (PoLiDBMS).

Note that here efficiency is for some aspects different from the usual DBMS performance concept. For instance, we assume our users to be highly interactive, so we cannot ignore the long delays of block erasures, neither their power consumption, since battery power is a precious resource in portable devices; at the same time we do not strive to execute lots of queries in parallel.

This paper is organized as follows. Section 2 introduces the scenario of the design methodology for Very Small Data Base for portable devices, the framework of the proposed DBMS. An overview of the entire approach will unveil the requirements and constraints that drove the design of our tool. Section 3 presents the DBMS architecture, focusing on its features and peculiarities; the next one proposes a critical evaluation of the achieved results, also discussing on-going work and future developments. Section 5 draws some final remarks and outlines our research trends.

2 The scenario

This section provides a description of the scenario and the motivations for developing PoLiDBMS, highlighting the peculiarities and the differences w.r.t. the environment of traditional DBMSs.

2.1 The VSDB Project

The environment we face is unusual for the common database management systems. For this reason the PoLiDBMS approach is basically different from the one of the existing tools for data management on portable devices. While their aim is to scale down an existing, traditional database management system to fit for the reduced power and memory of portable devices, PoLiDBMS has been developed to exploit a framework for direct data access and management based on a set of low level considerations on the particular kind of memory often used on portable devices: EEPROM Flash memory; indeed, our work starts from a physical view of the problem.

While Flash memory has good read and first-write performance (typically ranging from 80 to 120 nsec for read operations and 10 to 17 μ sec/byte for write operations), in order to modify data it is required to erase an entire block (typically from 0.45 to 0.5 sec/block) and write manipulated data back. This makes standard mass memory access methods disadvantageous, not only since an erasure takes a long time to be performed but also w.r.t. power consumption and memory endurance. The VSDB project proposes particular data storage and management policies with the main aim at reducing the number of memory erasures as much as possible. Two main memory management techniques have been defined [1], named the *deleted bit* and the *dummy record* policies.

2.2 Low Level Data Structures and Operations

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. We propose what we call *logistic data structures*, i.e. intermediate data structures that should be chosen to implement each database relation:

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.

Sorted relations, characterized by a medium ($\cong 100$ - $\cong 1000$ records) cardinality, are used to store information typically accessed by the sort key.

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.

Multi-index relations are used to manage generic data.

The *circular list* logistic data structure is an example of how the DBMS includes additional knowledge related to the application/data being managed. An INSERT SQL command has the following effect: (1) the data is “appended” at the end of the relation, (2) if the relation is full another record needs to be erased, (3) the DBMS chooses always to delete the least recently inserted record. Thus, deciding that a table is implemented by a circular list data structure means that a kind of “knowledge independence” is achieved by delegating this choice to the DBMS.

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 Flash 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.

We minimize block erasures also by introducing a number of dummy records per block [1]; 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.

The DBMS relies on an elementary record-based access method, which allows complete data management [1], while more complex tasks are left to the DBMS at a higher level. The basic functionalities are: *scan*, *equality search*, *range search*, *insert*, *delete*, *update*, used to read an entire relation, to read sets of records from a table selected on an equality or range condition, insert a new record, delete records on an equality condition, update a set of records, respectively. These operations may seem too basic, but combined with a few others used to modify the database schema they are the elementary memory operations necessary to build all the complex manipulation of data required by our DBMS. Within this framework PoLiDBMS has direct access to the memory: the granularity and specificity of the drivers we built offer the possibility of an extreme optimization of physical data management, reaching its top with a study of the flow of bits over the bus to reduce power consumption to its minimum. To test the various solutions and possibly offer feedback to a *workload simulator* for its decisions, an implementation of a low level environment for the described policies was carried out [6], exploiting and modifying the MIPS [7] assembler simulator *SPIM* [8].

2.3 DBMS functional specifications

Functional requirements are related to the environment the portable DBMS is built in, thus the DBMS functionalities are restricted to those considered useful on a portable device. Indeed, a portable device is likely to be used for specific purposes, and the data handled by the DBMS on the portable device often represent only a portion of the global database that has been split (following the methodology proposed in [9]) and stored part on the server side, part on the portable device, to be readily available. Therefore, administration operations such as the initialization of the database schema will be reserved to the DB administrator/designer, at an initial phase of the DB life, allowing the user to access data in read and modify mode, during normal operation, without modifying the database structure.

The current version of the DBMS is focused mainly on query processing, while transaction management is now under an integration phase. At the moment PoLiDBMS supports a subset of SQL DML: select, join, ordering, grouping, nested queries, multiple insert, delete and update operations, field and table alias names (more details can be found in [10]).

First of all, let us consider the DBMS interfaces: the end user must be able to perform queries through a *SQL*, as well as through a *GUI-based* (Graphical User Interface) interface, while the administrator/designer of the database will access extended features through a dedicated channel. As far as memory access is concerned, the DBMS interacts with a driver, that offers an abstracted view of the flash. For testing, performance analysis and profiling purposes, the DBMS is designed to communicate with external simulators and tools. The prototype tool is written in Java, for portability reasons and for easily interacting with other tools developed within the VSDB project.

User input via the *SQL interface*, typically an SQL statement, is parsed and translated into an internal representation of the query, on which the rest of the system will operate. The *GUI-based* access, which is a menu-driven graphical interface, offers the user the chance to perform queries through a simple interface designed to be comfortable over a PDA.

Because the project is aimed at offering the designer the possibility to define low level data storage and management for every relation, the best choice appears to be a dedicated *Administrator Access* interface which provides the DDL standard functionalities along with support for low level decisions about physical data storage and manipulation¹. The administrator access module is ideally placed near the low level memory access module, being tightly connected to it.

PoLiDBMS relies on the *Data Access Layer* for persistent memory access. This highly modular component offers a number of services, some of which are optional and can be included only when needed; the main ones are the aforementioned record-based elementary data manipulation operations. The implementation of this simple interface (described in greater detail in [10]) is provided by *Data Access Drivers*: an example is the native driver (we name it “native” because our implementation relies on JNI to interface with the C code implementing storage policy routines and flash memory access²), which is used by the DBMS core routines to gather records from tables stored on flash memory and modify them with insert, delete and update operations.

PoLiDBMS can interact also with the other tools of the VSDB methodology; some of these are already available while others are under development: the *Workload Simulator* will interact with the DBMS in different ways. It can work “over” the DBMS, to analyze response time for each operation of an interesting subset, or can operate with a particular *Data Access Driver* implementation for low level performance analysis. This interaction is exploited to gather information, necessary for calibrating several simulation parameters to obtain more accurate results. Again, a different Data Access Driver implementation might rely on a serial connection to access a remote entity instead of operating directly a flash memory: as an example, this can be exploited for testing purposes, operating on routines running on the assembler simulator SPIM, to test the framework described in [6], or interacting with other devices such as smart cards.

¹ Thanks to the flexibility of the architecture the DDL may be implemented if needed, making Administrator functionalities accessible through the SQL interface.

² On the Linux platform we have direct Flash memory interfacing capabilities.

Another tool to be integrated is a *Profiler*, that will help the DB Designer in the profiling step of the DBMS: this will be used to calibrate the system deciding query execution policies, as explained in the next section.

3 The DBMS Architecture

This section focuses on the *internal* view of the DBMS architecture, detailing the role of the components that implement our approach in the execution of the supported SQL statements. The design of the architecture (depicted in Figure 1) has been carried out with specific attention to modularization, making future developments (both optimizations and the addition of new features) easier.

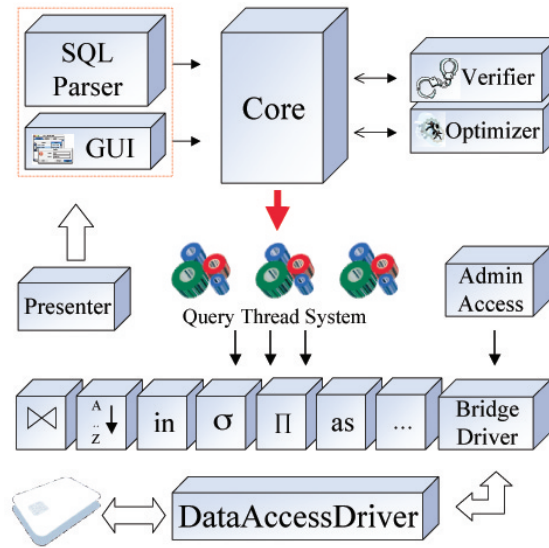


Fig. 1. The DBMS internal architecture.

SQLParser The input of this module is the SQL statement. This component, when no error is detected, builds the *QueryStack* (see below) to be processed by the DBMS core.

QueryStack A component of the *SQLParser* called *StackGenerator* creates the *QueryStack* object, by collapsing in a stack the branches of the tree representation of the query, through a postfix visit. The query construction procedure takes into consideration the basic functionalities offered by the *BridgeDriver*, a flexible component, but specialized w.r.t. the features of the low level data access policies. The *QueryStack* also contains other information, such as the identifier of the user who performed the request (represented by a *User* object), useful both for checking user's permissions to execute the query and to implement the concurrent multi-user feature (see future developments discussed in [10]).

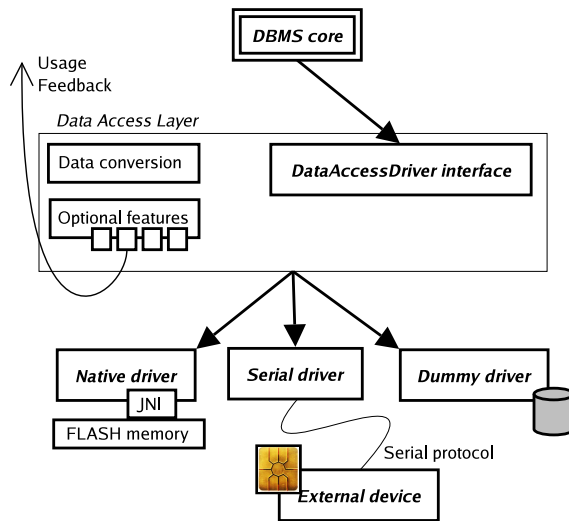


Fig. 2. *Data Access Layer* overview.

Graphical User Interface The GUI, thought as the main user interface to the DBMS, works via predefined menus. Such an interface is an alternative entry point. As a consequence, the module substitutes the *SQLParser* functionalities.

Verifier This specific module has been designed to control user's access rights thus achieving a centralized control of security/privacy issues, both for the SQL parser and the GUI. The *Verifier* module is necessary also for the login phase, hence the independent module solution provides a more flexible architecture.

Optimizer This module plays an important role for the DBMS performance. It analyzes the *QueryStack* object and, taking into consideration the schema of the database and the low level data structures and policies chosen to store the data on flash memory, produces a new, equivalent *QueryStack* object, which is optimized to offer better performance in the execution of the user's request. Another important decision the *Optimizer* can take concerns the execution mode: *single-threaded* or *multi-threaded*. In fact the results of the evaluation and testing analysis for the PoLiDBMS unveiled that running multi-process applications on portable devices is quite expensive due to the reduced computation power, or to the difficulties in multitasking management under a heavy workload. This issue could lead to the unexpected behavior that the execution of certain queries in the single-threaded mode performs better than running them in the multi-threaded way. Therefore the *Optimizer*, exploiting information gathered from performance analysis and profiling, can determine the most convenient mode to execute a certain query (or a certain *class* of queries).

This module is also the one that exploits the knowledge of the physical data structures used to implement the DB relations according to the innovative policies developed for the EEPROM Flash memory support. More specifically, when

selecting the query execution plan, the module takes into account information concerning not only the logistic organization of a relation (*heap, sorted, circular list* or *generic*) but also its physical implementation (*traditional, deleted bit, or dummy bit*). As an example consider the following SQL query:

```
SELECT table1.*, table2.field2, table2.field3
FROM table1 INNER JOIN table2 ON table1.field2 = table2.field1
WHERE table1.field1 = valueX;
```

where **table1** is a small relation (less than 10 records), without any ordering, implemented as a *heap* relation with a *deleted bit* physical data structure, and **table2** is a table storing data sorted w.r.t. **field1**, containing about 100 records (the underlying logistic and physical data structures are *sorted* relation and *distributed dummy bit*, respectively).

The execution of the query will require a *scan* of **table1**, that will be filtered on **field1** once in main memory. Starting from the resulting records, two are the possible plans to perform the JOIN: *search* on relation **table1** using the sorting field (and in particular the hashing approach provided by the driver) or a *scan* of the same relation, performing in main memory the required matches ON **table1.field2 = table2.field1**.

This second option may become more interesting for sorted relations implemented with the *distributed dummy bit* approach when there are several discarded records on the Flash that have been virtually deleted (but not physically) to limit Flash memory erasures. Such information on the memory status are provided by the low level driver and PoLiDBMS is able to exploit them in order to achieve significant performance. The decision to adopt one execution plan rather than another (although, given the device limited resources the number of different plans is quite limited) derives from application profiling carried out by means of the *Workload Simulator* developed to select, for each relation, the most promising data structures in terms of costs and performance.

The innovative logistic/physical data structures selected at design time, the interaction with the *Workload Simulator* and *Profiler*, exploited during the design of a VSDB, allow to move knowledge from the application to the DBMS, thus factorizing common tasks within the strict constraints imposed by the memory structure and the device limited resources.

Admin Access This component of the DBMS is used to perform the typical administration operations. Besides the possibility to alter the database schema (an operation not allowed through the SQL Interface), the administrator can establish how data are physically stored and handled, w.r.t. the low level policies previously described. This requires the administrator to be able to understand the physical implications of these choices, supported in this task by the VSDB methodology.

Data Access Layer In order to access Flash memory, the core of the DBMS interacts with the C Driver through the *Data Access Layer* interface. Alternative

implementations of the *Data Access Layer* may store the data on other possibly virtual units (for example for testing purposes) and from the DMBS-core point of view no function call would change. We also plan to implement a connection to the simulator [11], which would generate detailed statistics, useful for tuning other components. We already have optional pluggable mechanisms allowing the collection of information during normal usage for a given case study, logging calls to the driver to monitor costs and performance.

Other modules Each elementary operation pushed on the *QueryStack* is an instance of a common interface:

- *Joiner*, executes the join operation between two sets of records, offering SQL `INNER JOIN` and `OUTER JOIN` types as well as the cartesian product of the relations;
- *Sorter*, returns the given records ordered on the specified field(s);
- *BridgeDriver*, used to represent a single call to the *Data Access Layer* in the *QueryStack*. In fact, at the launch of the application, the *Data Access Layer* is instanced and that instance is used for the whole application; instead, the *BridgeDriver* is instanced specifically for every memory access operation in the *QueryStack*. According to how it is constructed it may represent a table scan, an equality search, a range search, insert, delete or update [1]. We added some extra features to these basic functions, like the possibility to insert an entire set of records, and to perform a delete or an update of records with a nested statement, such as:

```
DELETE FROM table1
WHERE table1.field4 IN (SELECT field1
                        FROM table2
                        WHERE field2=7);
```

- *Swapper*, used to swap columns when needed, for example when the user indicates, in the target list of a selection query, the columns in an order different from the order they appear in the relation, or to re-order columns in a nested query;
- *FieldRenamer*, used to rename the columns of a relation;
- *Presenter*, returns the results of the query. The functionalities of the *Swapper* and the *FieldRenamer* could have been included in this module, but separate modules have been designed in order to achieve a greater modularity, which allows also to swap columns before the final presentation step
- *Nester*, needed for any operation that involves nested queries, i.e. `IN`, `>` `ANY`, `<` `ALL`, etc;
- *ProjectionManager*, performs the projections of a relation; this module is not only called by the *Presenter*, but can also be used in case the *Optimizer* module pushes a projection to optimize the query execution;
- *SelectionManager*, executes the selection operation of a `ResultSet` given a specific condition.

Core This component is responsible for thread management offering the two previously discussed solutions of single-threaded and multi-threaded execution. When the single-threaded mode is adopted, the *Core* starts a thread only after the previous one has come to an end. When the multi-threaded mode is adopted the *Core* launches all the *QueryThreads* that can run concurrently, yet guaranteeing that updating statements be executed in isolation. The *Core* package includes a module referred to as *ConditionHandler* invoked whenever the evaluation of a condition is necessary. For example the *Joiner* module, or the *SelectionManager* module will use the *ConditionHandler* to determine whether a pair of records must be joined or a record must be kept or discarded. Moreover, the *ConditionHandler* is used to solve elementary operations included in a WHERE clause, e.g.

```
SELECT *
FROM table1
WHERE field1=field2 + field3
```

The *ConditionHandler* behaves differently from other modules, since it is called internally by such modules, but is not instanced and put on the *QueryStack*.

QueryThread This component wraps the *QueryStack* object and executes the elementary operations on the stack; it is used to improve the performance in the execution of more than one query to provide, when possible, the possibility to execute them concurrently. A synchronization mechanism guarantees that the *QueryThreads* (that run independently) return the results in the same order they had been requested.

ACID properties have been taken into account while designing the DBMS architecture, enforcing them at different levels of the architecture and involving several modules (e.g. the *Data Access Driver*). At present no mechanism for integrity constraints enforcement has been introduced yet.

4 DBMS Evaluation

A DBMS prototype has been implemented according to the analyzed requirements and the proposed architecture. As a first step, the primary goal of the development is to verify the feasibility of the project, i.e. the possibility to implement a DBMS targeted for portable device, considering its limited resources, as well as to exploit the advantages that the underlying flash access driver provides. The other main goal of this prototype is the validation of the architectural design of the DBMS. As a result, the proposed implementation is only a preliminary prototype with limited functionalities. The natural evolution of the tool, given the positive evaluation, is the re-engineering of the preliminary modules and the development of advanced components in order to achieve a complete DBMS.

As it is, the prototype can be used to integrate the framework of the Very Small Database Design Methodology, interacting with the other software tools, allowing us to draw preliminary conclusions about the entire methodology.

The adopted programming language is Java [12] based on the necessity to create an integrated environment together with the tool supporting the methodology and the simulator for identifying the most convenient data access/management policies [1]. The target platforms on which the DBMS is meant to run, which range from a Smart Card to a cell phone or a PDA, impose an even tighter requirement for the selection of the programming language. The common factor to these reference platforms is their ability to run Java applications. Our prototype proves the feasibility of a portable light database management system. The quality of the achieved results has been analyzed by testing the application both on a hand held computer (a H3900 iPAQ with linux) and on a full-featured Personal Computer.

From the first tests, performance, both on the full-featured Personal Computer and on the PDA, seems satisfactory, especially for standard, not intensive computations, such as the ones expected to be run. We noticed a sensible performance degradation, on the portable device, for relatively heavy workloads, such as a highly complex query containing an `ORDER BY` clause applied to a thousand records. Thus we can conclude that, considering the peculiar characteristics and the limitations of the portable devices, we have reached our first purpose. The system is actually in a phase of deeper performance testing and benchmarks are foreseen against other DBMSes of the same class.

The project is still under development, taking also into account extensions and functionalities that have been planned, but are currently “work in progress”.

A first goal of these activities is devoted to the optimization of the already developed modules, aimed at exploiting their performance and completing the pending unavailable features.

Another important goal consists in investigating distributed transaction management aspects and the synchronization of the portable database with the one(s) resident on the server side(s). The two issues are closely related as far as a “special” access to the database data is necessary in order to support them, covering also aspects related to ACID properties enforcement and users’ permissions management [13].

5 Final remarks

This paper introduces the architecture of a Database Management System for portable devices, called PoLiDBMS, designed and implemented (in a prototype version) as part of a more comprehensive framework targeting Very Small Databases. The proposed DBMS is built on top of a *Data Access Layer* designed to enhance data access and management performance for small amount of data stored on EEPROM Flash memories.

The DBMS architecture we propose has been specifically designed to cope with the requirements and constraints of small devices characterized by reduced resources. A flexible and modular solution has been adopted with the aim of allowing the development of a system able to be customized in its features, depending on the needed functionality and the available processing power. The

first prototype implementation provides all the elementary functionality of a DBMS, supporting a reduced set of the SQL language that can be of interest in such a limited environment. Such a prototype will be the starting point for a re-engineering process aimed at completing the secondary modules and optimizing the fundamental ones, in pursuit of a full-featured PoLiDBMS. Advanced features have also been investigated and planned, assuming the adoption of the tool also in other application environments.

References

1. C. Bolchini, F. Salice, F. A. Schreiber and L. Tanca, "Logical and physical design issues for smart card databases," *Transactions on Information Systems*, vol. 21, no. 3, pp. 1046–8188, 2003.
2. C. Bobineau, L. Bouganim, P. Pucheral and P. Valduriez, "PicoDBMS: Scaling down database techniques for smart card," in *26th Int. Conf.e on Very Large Databases*, 2000, pp. 11–20.
3. *Smart card adoption for ID application in the Italian Government*, Italian Government, 2002, http://www.innovazione.gov.it/ita/comunicati/2002_02_08cie.shtml.
4. J. Sutherland and W.-J. van den Heuvel, "Enterprise application integration and complex adaptive systems," *Comm. of the ACM*, vol. 45, no. 10, pp. 59–64, 2002.
5. K. Cheverst et alii, "Developing a context aware electronic tourist guide: Some issues and experience," in *Proc. of CHI '2000*, 2000, pp. 17–24.
6. C. Curino, M. Giani, M. Giorgetta and A. Giusti, "MIPS implementation of some *very small data bases* data structures," Polit. di Milano, Tech. Rep., 2003, 2003.45.
7. E. Farquhar and P. Bunce, *The MIPS Programmer's Handbook*. Morgan Kaufmann, San Francisco, CA, 1994.
8. *SPIM20: A MIPS R2000 Simulator*, University of Wisconsin-Madison, 1996, available online at <http://www.cs.purdue.edu/homes/hosking/502/spim/raw.html>.
9. C. Bolchini, F. A. Schreiber, and L. Tanca, "A context-aware methodology for very small data base design," *SIGMOD Rec.*, vol. 33, no. 1, pp. 71–76, 2004.
10. C. Curino, M. Giorgetta, A. Giusti and A. Miele, "Portable Light DBMS: PoLiDBMS White Paper," Polit. di Milano, Tech. Rep., 2003, 2003.46.
11. D. Roncelli, "Definizione e sviluppo di una metodologia per l'allocazione logico/fisica di basi di dati su smart card," 2002/2003, tesi di laurea.
12. Sun, "Java website," 2003, <http://java.sun.com/>.
13. C. Bolchini, A. Lazaric, C. A. C. Pascali, S. Sceffer, F. A. Schreiber, L. Tanca, "Implementation of a distributed commit protocol on the PoLiDBMS," Politecnico di Milano, Tech. Rep., 2004, MAIS Internal Report WP5.2.