

---

## Data Management

C. Bolchini, F.A. Schreiber, and L. Tanca

A multichannel mobile environment offers very interesting challenges for research on effective and efficient data management; indeed, the variety of device storage capabilities, together with the availability of huge amounts of data, only parts of which are interesting to the device user, open up completely new research issues. It is important to be able to select which part of the entire data must be readily available to the user, depending on his/her interests and, more generally the *context*, and to access/manage such data in an efficient way with respect to the device's technical features and limitations (memory, power, performance, and so on).

Within the MAIS project, the Very Small DataBase (*VSDB*) project [73] is aimed at providing both ends of the solution: a design methodology for determining the portion of data to be held on the portable device, and a Data Base Management System (DBMS) for accessing such data in the most convenient way.

The main difference from the traditional design methodologies is the focus on ambient awareness, which allows the specification of the “VSDB ambient”, i.e., the set of personal and environmental characteristics determining the portion of data that must be stored on the portable device.

On the other hand, the DBMS must integrate logical and physical data structures defined to exploit the technological characteristics that are common to portable devices, and provide the classical features of DBMSs that are necessary in the current scenario. As a result, the architecture of these devices has an impact on the data management policies. Fig. 6.1 depicts the project scenario.

Here we assume the existence of a (possibly distributed) database, for which a global schema has been defined and which is located on fixed devices. This means that VSDB's are defined as (collections of) materialized views on this database (Fig. 6.2). Future work is aimed at generalizing our research to the case where a mobile device forms part of a complex information system where no global schema is known.

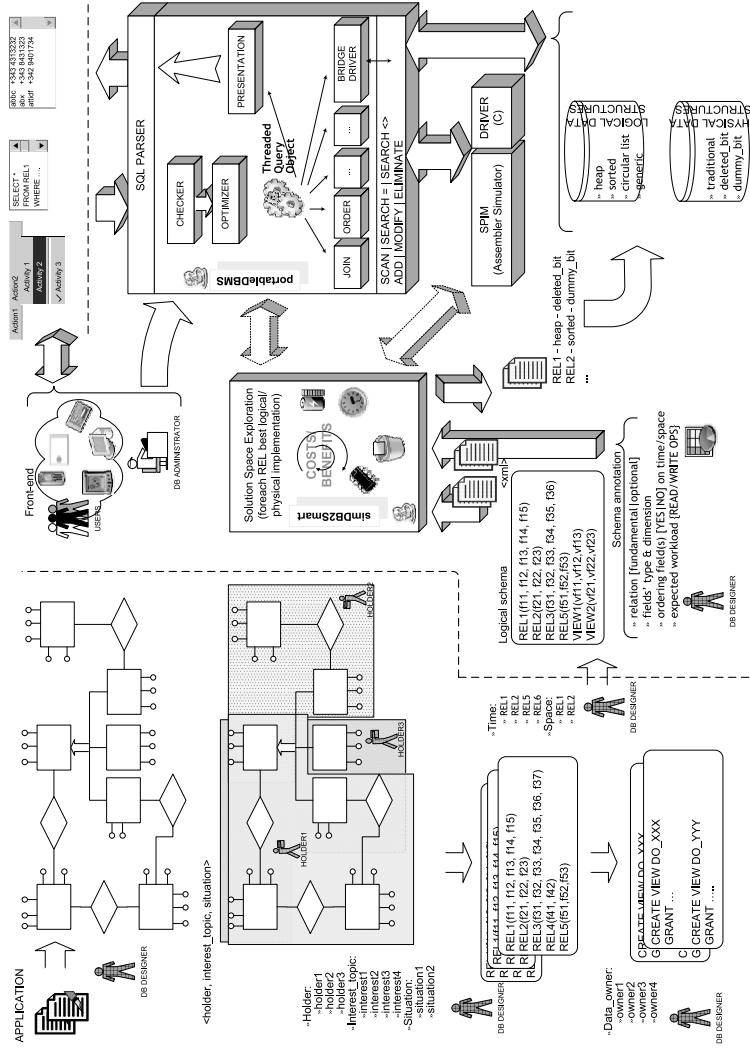
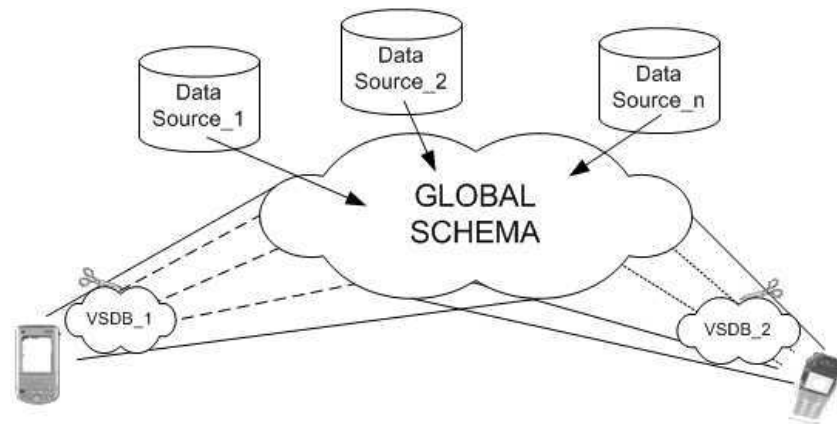


Fig. 6.1. The VSDB project: portable-database design and DBMS

## 6.1 Architectural Features for Data Management

Out of the several families of portable devices, we focus our attention on smart cards, smart phones, and PDAs. The presence of a variable degree of computational capability, provided by the presence of a microcontroller, the ability to interact with other devices through their interfaces, and the presence of a limited amount of NonVolatile Memory (NVM), usually EEPROM flash memory, are common to all these devices. The microcontroller's performance, the types of interfaces provided by the environment/devices, and the amount



**Fig. 6.2.** The overall architecture for data management

of available memory vary from device to device; nevertheless, it is possible to say that, altogether, the resources of portable devices are limited, compared with typical full-size computers. Furthermore, the nature of the nonvolatile memory impacts significantly on the overall performance achievable with these microdevices, as discussed in the next subsection.

### 6.1.1 Memory Types

In general, two kinds of flash memory implementation can be employed for storage in portable devices: NOR and NAND. Owing to density, cost, and speed reasons, embedded and mobile systems are increasingly using NAND flash EEPROM for storage. The most common consumer usage of NAND flash is in the form of SmartMedia cards, which are simply NAND chips bonded to a carrier card. Consumer use of SmartMedia is driving down NAND prices, while driving up densities. As always, though, life is made of compromises, and those advantages come with some limitations that need to be addressed to provide robust data access. Table 6.1 reports the typical characteristics of NOR and NAND EEPROM flash memories.

Program and erasure operations require particular effort compared with classical magnetic-disk or RAM support, since a memory location, independently of the granularity allowed by the specific type of memory, needs to be erased before it can be programmed. More precisely, write operations can only modify 1s to 0s. Changing 0s to 1s requires an erasure; furthermore, in NAND memories, a page may be programmed only a certain maximum number of times.

The amount of data storage available in a flash memory ranges from 64 Mbit to 8 Gbit. Power requirements vary depending on the operation that has to be performed. A read operation requires an average of about 10 mA (12

**Table 6.1.** Characteristics of NAND and NOR EEPROM flash memory [354]

	NOR	NAND
Density	Up to 32 MB chips	From 128 Mbit to 8 GBit.
Cost per MB	\$2	\$0.5
Access	Linear random access	Sector read/write: Page-oriented with spare area in page Sequential access within a page
Organization	Erasable blocks of 8 kB to 128 kB typical	Erasable blocks of 32 x 512-byte pages 16 bytes of extra management data
Target	ROM replacement	Mass storage
Programmability	Byte-by-byte allowing single-bit modification.	Page or partial-page programming.
Endurance	100 k to 1 M erasures	100 k to 1 M erasures
Read speed	50-100 ns	10 $\mu$ s page seek + 50 ns per byte
Program time	5 $\mu$ s per byte	200 $\mu$ s per page
Erase time	1 s per erasable block	2 m per erasable block

mA maximum), whereas program and erase operations require an average of about 20 mA (35 mA maximum). Access time depends on both the type and the mode of operation.

### 6.1.2 Endurance, Power Consumption and Performance

Using this technology, write operations can be performed only if the target location either has never been written before or has been previously erased. Erasure can only be done at *block* level, whereas read and write operations work at 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.

More precisely, every modification on the data affects the endurance of the device, impacts on power consumption, and, depending on which microoperations need to be performed, determines the performance level that can be achieved: the main difference compared with classical storage support consists of the necessity to delete a location before being able to rewrite it, and to erase an entire block even if a single piece of information is modified. In fact, when a block needs to be erased for a single modification, all the information in it must be saved, the block must be erased, and then all data except the part that has been modified must be copied back to the block.

As a consequence, a DBMS using a flash memory must take into account all of these aspects, by trying to reduce the number of data modifications required by data access. Ad hoc physical data structures and data access, storage, and management procedures have been investigated, evaluating endurance, power consumption, and performance levels, in order to select the most promising policies to be adopted in the lowest layer of a DBMS for portable devices.

## 6.2 DBMSs for Small Devices

The resources of portable devices, although limited, allow the user to carry around a useful portion of data, to be read as well as modified. Such data may be part of a larger system (such as a person's medical records) or may be the unique copy of a user's information (such as a person's Internet access data): in both cases a portable DBMS is desirable as a backend for accessing and managing data.

### 6.2.1 Commercial Tools

In this subsection, we present a short survey of the available commercial tools that implement relational-database management systems for portable devices, focusing our attention on their synchronization policies. We consider the following systems:

- Oracle Database Lite 10g [293]
- IAnywhere UltraLite database [192]
- IBM DB2 Everyplace [193]
- Microsoft SQL Server Mobile Edition [267]

#### Oracle Database Lite 10g

Oracle Database Lite is an addition to the Oracle DBMS and is used for mobile and small-footprint devices. Oracle Database Lite uses data synchronization to exchange data between an Oracle database and a remote environment. More precisely, the DBMS includes a bidirectional *synchronization server* with a publication and subscription-based model that allows data to be synchronized between mobile users and the Oracle database. When concurrent data modifications occur on the remote database and on the server, conflicts are resolved by means of configurable standard resolution rules.

The following synchronization and network protocols are supported: TCP/IP, HTTP, 802.11b, PPP12, GPRS, HotSync, and ActiveSync.

#### IAccessAnywhere UltraLite Database

The *UltraLite* database provides mobile users with access to local and remote data when a connection is available, and queues up transactions when offline. A synchronization server provides database-to-database synchronization, offering bidirectional exchange of information between remote databases and an enterprise data source, via a priority approach. Remote devices connect via standard internet protocols, such as TCP/IP, HTTP or HTTPS.

Developers can create complex rules to subset data, by partitioning both horizontally and vertically, in order to select the portion of data that the end users have access to.

### IBM DB2 Everyplace

*DB2 Everyplace* can be used as an independent database, local to the mobile device, or to query information on remote servers when a connection is available. Data can be synchronized between DB2 Everyplace client devices and enterprise data sources using a synchronization server. Synchronization can be bidirectional or unidirectional; conflict resolution and data partitioning are supported.

### Microsoft SQL Server Mobile Edition

The *Microsoft SQL Server Mobile Edition* engine exposes an essential set of Relational-database features. Remote data access and merge replication ensure that data from SQL Server databases can be manipulated off line, and be synchronized later on to the server. No further details are available.

The common factor in all these light DBMSs is the underlying client-server architecture, where the portable device hosting the light DBMS is a client, and a full-featured server is the center of the architecture. The aim of these light DBMSs is to scale down an existing tool, to make it fit the reduced computational power, battery life, and memory of portable devices, but to continue to provide a traditional database management system.

### 6.2.2 PoLiDBMS: System Features

When such reduced resources are considered, not all of the classical features of a DBMS are necessary, especially when one takes into account the limited amount of data held on the device and the fact that the SQL engine will serve the purpose of data access/manipulation rather than database creation or administration. Furthermore, the particular technological characteristics of the storage medium suggest that we need careful manipulation of the stored information to limit endurance degradation and power consumption, and to achieve good performance.

As a consequence, a new DBMS has been developed [71]. We have named it *PoLiDBMS* from *Portable Light DBMS* (and also *Politecnico di Milano DBMS*). In it, a bottom-up approach is adopted, in order to exploit ad-hoc physical data structures, designed to meet the challenges of the storage medium and to fulfill the requirements of efficiently managing small amounts of data. PoLiDBMS is part of the VSDB project and provides an SQL engine for managing the portion of data stored on the portable mobile device.

Although devices are advancing rapidly, system resources, such as available memory, are often scarce, so it is critical that a relational database system is as compact as possible while still providing the essential functionality. The DBMS architecture that we propose has been specifically designed to cope

with the requirements and constraints of small devices characterized by reduced resources [73]. A flexible, modular solution has been adopted with the aim of allowing the development of a feature-customizable system, depending on the functionality needed and the processing power available. The first prototype implementation provides all the elementary functionality of a DBMS, supporting a reduced set of the SQL language that we consider to be of interest in such a limited environment. The following paragraphs describe the physical-design and query-processing policies implemented in our prototype. Transaction handling and synchronization strategies, which have been investigated but only partially implemented, are described in Sect. 6.3.2.

### Data Storage Policies

Classical, indexed data structures are often inappropriate for VSDBs; indeed, our search needs and the fact that searches are conducted within small tables is often not worth the overhead required for managing and maintaining indexes, which have been proposed only in the case of tables with large cardinality and special needs for multikey searches [69]. Instead, we propose what we call *logistic data structures*, i.e., intermediate data structures that are 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 one is looking for a specific record; in the case of a personal-assistant device database, with telephone/Internet access data, an example could be a relation that stores data on the telephone/mail accounts that the owner has.

**Sorted** relations, characterized by a medium cardinality ( $\cong 100$  to  $\cong 1000$  records), are used to store information typically accessed by the sort key. The idea is to impose an upper bound on the number of records that can be inserted based on the complete size of the (fragment of the) table. Once the upper bound is reached, the user will have to delete (or store externally) a record before adding a new one. The address book of the owner's contacts is a relation well suited to this kind of data structure.

**Circular-list** relations, characterized by a medium cardinality as well, are again suitable for managing a fixed amount of log data, for example sorted by date/time; in this case, once the maximum number of records is reached, the next new record will replace the oldest one. The list of the last  $n$  calls can be stored by means of circular lists.

**Multi-index** relations are used to manage generic data, typically when the need is to efficiently access large relations by multiple keys. This is the only data structure that we propose which resembles the classical data structures used in DBMSs, and we shall not elaborate further on this type of structure.

Our methodology requires the designer to tag each table to be included in the VSDB with the following information:

- the tuple length (in bytes) and the expected relation cardinality; it is also possible to specify an upper bound on the number of records to be allowed;

- the presence of a sorting field, specifying whether the field is a time field leading to a log-like file;
- the expected composition of the set of operations on the data: *insert/delete/update/select*, the last one classified further into full select (*scan*), select with equality (*equal*), and select with range (*range*).

The expected composition refers to the relative frequency of operations. For instance, consider a relation storing a list of bookmarks in the above PDA scenario; the user can say that the dominant operation will be *insert* there will usually be no *deletes* and very few *updates*. The other common operation is *select*, assuming an equal distribution among the three selection schemas identified. A simulator has been built [71], to give an indication of the data structures that the DBMS must employ for the required relations. The implementation of the data structures is discussed below.

### Physical Design

The goal of the data structures implemented by the proposed DBMS is to optimize performance and to minimize power consumption and degradation of the flash memory, while limiting memory and computational overheads. Note that these aspects are strongly related, and that block erasure significantly affects all of these parameters.

To our knowledge, other DBMSs for small devices propose physical data structures which are small sized copies of the ones used for classical, magnetic storage devices, and do not take into account the main physical features of flash memories.

In accordance with the technical features of the storage used in mobile devices, i.e., flash memory, we propose an implementation of the physical data model previously discussed, based on the introduction of two elements:

- Use of a *deleted bit* to carry out a logical rather than physical deletion of a record, in order 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 *dummyrecordsperblock*, 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 other data structures; notable examples can be found in B-trees of order  $n$ , where each node can host a number of items varying from  $n/2$  to  $n$ , and in static hash tables, where the filling of pages is controlled in order to avoid too many collisions [151, 384]. The technique uses a *valid bit* to indicate if the record has been programmed or not.

These two additional bits associated with each record allow one to reduce the number of modifications requiring erasure of flash memory; in fact, when the stored data need to be modified, at least one memory block (and possibly



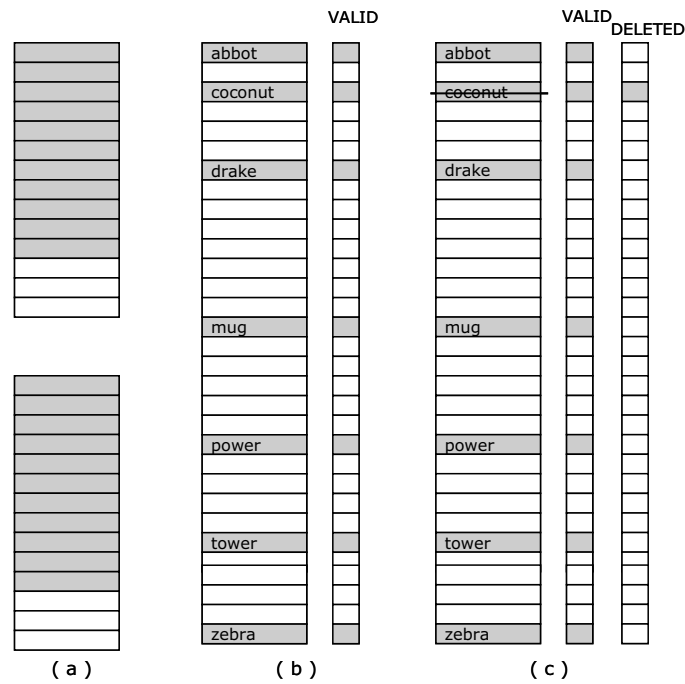
many) needs to be rewritten, implicitly requiring a copy of its contents in the RAM, an erasure of the flash block, and a write-back, from RAM to flash, of the modified contents (a *dump/erase/restore*, or DER, sequence). Note that the DER sequence greatly affects performance (owing to the time required for the data “dump”), power consumption, and storage endurance.

More precisely, the use of the *valid bit* is essential when memory is managed in a nonsequential fashion; in particular, the valid bit implements a “distributed” control, since each valid record is directly distinguishable from the others, whereas an end address (register) implements a “concentrated” control, since it unambiguously identifies the end of the record list. This concentrated control is a space-aware but energy- and time-consuming approach, since the end-address value needs to be updated every time the list is modified by a *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 physical expunging to a later time. The *deleted bit*, coupled with a nonsequential management of the physical memory, reduces the necessity to erase blocks, at the cost of an increase in the amount of memory required and a more complex management policy, as discussed in the following.

When one is dealing with data sorted with respect to a field, insert and delete operations have a significant overhead owing to the necessity to maintain the data in an ordered state; furthermore, if the relation data is distributed over several blocks, the operation might affect multiple blocks. The proposed data structure [73] is aimed at (a) confining the involvement of the blocks in data manipulation and (b) minimizing block erasure. These goals are achieved by introducing a number of dummy records in each block (Fig. 6.3a); such records may be either localized at the end of the block or distributed throughout it by means of a hashing function, so that future insertions do not always cause a reorganization of previously introduced records (Fig. 6.3b). The hashing function may be implemented either in software or in hardware; in this case a *valid bit* is mandatory for determining which records are programmed and which are not. The use of concentrated dummy records is aimed at preventing the involvement of multiple blocks when records need to be shifted up or down following a delete or insert operation (intra-block erasures). The solution of distributed dummy records also limits interblock erasures. The *deleted bit* has the same functionality as described above here (Fig. 6.3c).

The combined use of dummy records and the *deleted-bit* technique is useful in the case of sorted relations, whereas the use of the *deleted bit* alone is suitable for circular lists and possibly heap relations, at the cost of an additional space requirement compared with the minimum possible amount of memory. Tab. 6.2 reports experimental results for the proposed physical data management techniques.



**Fig. 6.3.** Use of dummy records (a) concentrated at the end of the block (white elements) or (b) distributed throughout the block. (c) Use of distributed dummy records and the *deleted bit*

**Table 6.2.** Simulation results: block erasures performed and bytes transmitted on the system bus compared with the “the simple” solution with no *deleted bit* and no dummy records

Data structure	Strategy	Block erasures			Bits transmitted 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 adjacent	0.83	0.51	0.44	0.74	0.57	0.45
	Dummy distributed	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

## Querying

PoLiDBMS provides a basic query-processing feature, similar to those of classical DBMSs. SQL statements are parsed by *SQLParser* and an internal representation of the query is created. The output is a stack of elementary operations executable by a single module (the *Core*), which can be optimized by reorganizing or modifying the elementary operations in order to improve query-processing performance. Such optimizations take information about the logical data structure into account to exploit the peculiarities of the data being manipulated. The last module invoked in the operation sequence is the *Presenter*, which returns the result of the execution of the statement to the caller.

## Interface

A standard API for accessing the DBMS has been developed, to provide a unified, almost classical method of access to the DBMS. Thanks to its highly modular architecture, PoLiJDBC, a JDBC<sup>TM</sup> driver, fits the environment of PoLiDBMS perfectly; it is small, it supports local transactions, it is extensible, and has been written from scratch following a “*scaling-down approach*” [69]. This new-generation driver not only provides the standard JDBC APIs but also enforces the particular features of PoLiDBMS. The standard API has been extended to accommodate PoLiDBMS so that its particular features, such as transaction boundaries and data types, are fully supported and existing applications can be compliant with both JDBC and PoLiDBMS, without expensive code modifications.

## 6.3 Design of Very Small Databases for Mobility

Database design methodologies for small, mobile devices concentrate on defining the notion of an *ambient*, which drives the tailoring of the portion of data to be stored locally. As a matter of fact, this must also regulate the way device data are acquired at synchronization time, i.e. the synchronization issues concerning data semantics, discussed in Sect. 6.3.2.

The notion of an *ambient* that we use in this chapter is only loosely related to the general notion of a *context* in MAIS (see Chap. 2). The ambient of a device is a strongly data-centric concept, which analyzes the device users’ needs in terms of their information needs; in contrast, the notion of a context in MAIS has the twofold objective of configuring the software on board the device (a) on the basis of the needs of the user, in terms of presentation, and (b) on the basis of the characteristics of the device, in terms of the available channels. An example of such a difference can be seen in the concept of time: in the model of the MAIS context it means capturing the moment in time that the user is currently experiencing, while in the case of the ambient array, time

is coupled to the further specification of an interval of interest, and used to filter the information pertaining to that interval (e.g., a patient's prescriptions for the last month, or a doctor's visits this week).

### 6.3.1 VSDB Design Methodology

The VSDB design process consists of three main phases [74, 75]: conceptual design, logical design, and logistic design, which are discussed below.

The *conceptual design phase* can, in turn, be decomposed into the following four steps.

**1. Application information modeling.** This is done using 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. In fact, the design of the VSDB must be merged with the design of the distributed database that it belongs to.

**2. Choice of the *analysis dimensions*.** Analysis dimensions provide the various perspectives that the mobile device is viewed from, and are used to set out the *ambient* of the VSDB. Here we consider some intuitive dimensions, which can be integrated with additional ones or omitted 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, whereas **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 that the user might be interested in, at a certain moment. In the case of medical care, topics might include **prescriptions** and **chronic diseases**. In a tourist guide application this dimension might refer to the choice of information about entertainment in a city, or about restaurants, etc.
- The *situation* dimension refers to the fact that during the life of the device the user may wish to access different views of the data for performing different operations. For instance, in a personal medical-information system, an example of a situation is the **regular** situation, i.e., a patient's ordinary state, as opposed to a temporary **hospitalized** situation.
- The *interface* dimension refers to the kind of access to the contents of the database: access may be required by a human actor or by a machine system. This dimension suggests that different types of interacting entities may need different data presentation profiles; i.e., for a human profile, internal IDs are not necessary, and may be confusing, but for a machine profile, internal IDs are necessary, whereas more expressive textual or visual descriptions are useless.

- The *time* dimension refers to the life span of the information that the VSDB tables must store: for example, one could save the whole medical history of patients in a fixed machine belonging to 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 may be interested, during a work trip to Genoa, in all medical facilities in that city, and information about other such facilities located in other cities is to be disregarded.

Note that the time and space dimensions determine further tailoring of the data aggregations that have been allocated to a device, by means of logical views that limit the information to that pertaining to the current ambient.

As the output of this step, the dimensions identified are collected to form the *ambient array model*, which drives the actual choice of the information to be kept on the microdevice. As an example, we can form the following four-position array model below:

`<holder, interest_topic, situation, time>.`

For simplicity, we have not used the space and interface dimensions here.

**3. Conceptual *chunk* derivation.** Here, the *array schemata*, or *chunks*, are derived from the array model by instantiating the dimensions; some examples of chunks in the case of the medical-care database (MCDB) considered above are the following:

`<patient, chronic_diseases, hospital, past year>.`

This chunk contains all the information needed by a patient in a hospital with respect to his/her chronic diseases (if any) during the whole past year.

`<patient, prescriptions, regular, this month>.`

This chunk contains all the information needed by a patient in a normal situation with respect to his/her prescriptions (if any) during the whole current month.

`<doctor, prescriptions, regular, today>.`

This chunk contains all the information needed by a doctor with respect to all his/her regular patients' prescriptions today.

The derivation of chunks must be done taking into account their significance: only some of the possible combinations of dimension values make sense. For example, the chunk

`<doctor, accounting, hospital, past year>`

makes little sense in view of the application semantics.

As the conclusion of this step we assemble chunks in order to define information that must be stored on one individual device. However, final decisions may be made only at *logistic-design time*, i.e., in a phase when the amount of memory required for the tables can be evaluated. For example, normally a patient's smart card will contain all of the chunks related to the patient's (regular) situation plus those related to his/her chronic diseases (such as allergies) and prescriptions. When the patient is in hospital, the "regular" chunks will be removed to make room for the "hospital" ones. However, if the device has more resources (for example in the case of a PDA), the designer might decide to leave all the chunks related to different situations at all times.

**4. Choice of the *driving dimension*.** The designer needs to decide which dimension is central to the whole analysis process; this depends on the application. The driving dimension's views will be built at conceptual-design time, whereas all the other dimensions' views will be derived at logical-design time. In the application considered here, as is usually the case, we have chosen *holder* as the driving dimension; one conceptual schema must be built for each value of *holder*, i.e., we build one conceptual view for the **patient**, one for the **doctor**, and one for each of the possible other values of this dimension (e.g., the hospital administrator). Here, some 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.

In the *logical design* phase, various activities are carried out:

- *Logical design of the global database*: some 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, this month>.

is defined as:

```
CREATE VIEW PAT-PRESC-HOSP-THISMONTH AS
SELECT P.SSN, P.FName, P.LName, DRUG.Name AS DrugName,
Posology, SideEffects, Mode, Dosage, Administration,
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' AND
PR.ENDDATE >= now() - 30,
```

where `now()` is a system function returning today's date.

- *Chunk instantiation*: here, the views for the chunk instances are produced. A chunk instance relates to one specific instance of a dimension value. This is an example of a view instantiation:

```
SELECT * FROM PAT-PRESC-HOSP-THISMONTH
WHERE SSN = $ID AND COMMENTS like '$prescription' .
```

Such a view contains the parameters `$ID` and `$prescription`, which will be actualized at run time with the specific user's SSN and one of the values of `prescriptions` in the interest topic dimension, e.g., "Antibiotics".

- Introduction of the *logistic dimensions*, i.e., dimensions which do not influence the actual design of the database, but only the logistic phase. We introduce here only the *data ownership* dimension, concerning `read`, `update`, `delete`, and `insert` access rights to the VSDB information, which might be different depending on the category of user. Note that access rights must be analyzed with respect to *actors*, that, in general, are different from the device holders: in the MCDB example, a patient's doctor has the right to modify 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 identification of the ambient.

In the *logistic design phase*, in accordance with what has been said in Sect. 6.2.2, the designer has to tag each table in the chunks to be included in the VSDB with information about the tuple length, the expected cardinality (e.g., five records for the PREGNANCY relation for the holder "PATIENT"), the presence of a sorting field, and the expected relative frequency of each type of operation on data, i.e., *insert/delete/update/select*. For instance, consider the DRUG relation; the user can say that the dominant operation will be *insert*, and here will usually be no *deletes* and very few *updates*. The other common operation is *select*, assuming an equal distribution among the three selection schemas identified (Fig. 6.4).

	LENGTH	CARDINALITY	LIMITED	KEY	ORDERED	ACCESS TYPE FREQUENCY							DATA STRUCTURE ***
						INSERT	DELETE	UPDATE	SELECT				
									scan	equal	search		
P_PersonalInfo	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_Intolerance	30	80	NO	DrugID	YES	LOW	LOW	LOW	HIGH	HIGH	HIGH	S	
P_RegularUse	150	30	NO	DrugID	YES	LOW	LOW	LOW	HIGH	HIGH	HIGH	S	
P_Anomalies	50	20	YES	TreatID, EndID	YES	HIGH	NEVER	LOW	HIGH	LOW	LOW	CL	
P_Pathologies	90	20	YES	TreatID, EndID	YES	MEDIUM	NEVER	LOW	MEDIUM	LOW	LOW	CL	
P_TraumaInjuries	120	20	YES	TreatID, EndID	YES	MEDIUM	NEVER	LOW	MEDIUM	LOW	LOW	CL	
P_Allergies	100	20	YES	TreatID, EndID	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, CL = Circular List

Fig. 6.4. The result of the logistic phase

### 6.3.2 Data Synchronization and Transactions

Data synchronization can be discussed at two levels of abstraction, one concerning the transactional problems related to distributed databases, and the other concerning the aspects related to data semantics.

#### Distributed Commit Protocols

A transaction is a set of operations starting with a **BEGIN TRANSACTION** statement, and concluded with either a **COMMIT** or a **ROLLBACK** statement. The whole sequence of data operations included between these statements must be considered as one *atomic* entity, i.e., either the transaction does its work and thus brings the database from a correct state to a new correct state, leaving a permanent result in secondary storage (**COMMIT** case), or it leaves the database unchanged, possibly undoing all the operations performed in the meanwhile (**ABORT** case).

To preserve transaction atomicity, in the distributed case, several protocols have been designed and implemented; the most popular is the family of *Two Phase Commit (2PC)* protocols [32].

The basic 2PC protocol, often called the *Presumed Nothing* (PrN) protocol [2], requires the participants to explicitly exchange information, and log whether the transaction is to be committed or aborted. At the end of the transaction, the coordinator invites all the participants to commit, and each of them votes to commit or abort its local part, on the basis of local conditions. For the transaction to be committed, the coordinator must collect a unanimous consensus; otherwise, it orders the abort and rollback of all of the local actions and, therefore, of the entire transaction. Several optimizations of the 2PC protocol that make presumptions about missing information have been proposed, in particular, the *One Phase Commit (1PC)* protocols, which rely on the idea of eliminating the voting phase of the 2PC protocol by enforcing some properties of the behavior of participants during the execution of the transaction.



The basic assumption underlying the 1PC protocol, several variations of which have been studied [14, 351], is that a participant does not need to vote. To implement transaction atomicity in PoLiDBMS we chose a 1PC protocol, the *Unilateral Commit Protocol (UCP)* [70, 72], which has been explicitly designed for mobile distributed, disconnected computing applications (e.g., those stored on devices such as smart cards). In this protocol, the coordinator acts as a dictator that imposes its decision on all of the partners. If a crash precludes a participant from conforming to this decision, the coordinator simply forward-recovers the corresponding transaction branch. The gain in terms of performance (blocking I/O, latency, and messages) is obvious and can be exploited greatly in a wireless communication network.

The UCP exhibits the following properties, which are useful in a mobile environment:

- A transaction executed off line can commit as soon as its log has been transferred to the fixed network, without waiting for acknowledgment from the fixed servers.
- The protocol does not require the presence of all servers at commitment time.
- The protocol is composed of a single message round, thereby saving costly wireless communications.
- The protocol does not require a prepare state nor a corresponding interface on the server side.

The UCP distinguishes among five types of components which interact during the execution and termination of a transaction:

1. The Application asks for the execution of a sequence of operations.
2. The LogAgent logs each operation before execution.
3. The Participants execute these operations.
4. The Coordinator pilots the termination protocol.
5. The PAgents (one per Participant) represent the participants in the termination protocol and play an active role during recovery. These also mask the heterogeneity of the participants from the Coordinator, enable the participation of any kind of server (2PC-compliant or not) in the UCP, and acknowledge the Application.

These components may be co-located or not, depending on the hardware and software configuration. Note that the Coordinator is still located on the fixed network, while the other components can potentially be hosted by a mobile partner [70]. Typically the Application, the LogAgent and the Coordinator are located on one site of the fixed network, while the Participants are mobile and their PAgents are located on mobile support stations. The commit scenario produced by the UCP is depicted in Fig. 6.5, where  $T_{ik}$  denotes the local branch of transaction  $T_i$  executed at participant  $P_k$ . Assuming one coordinator and  $n$  participants, the transaction execution protocol based on the UCP is as follows [14]:

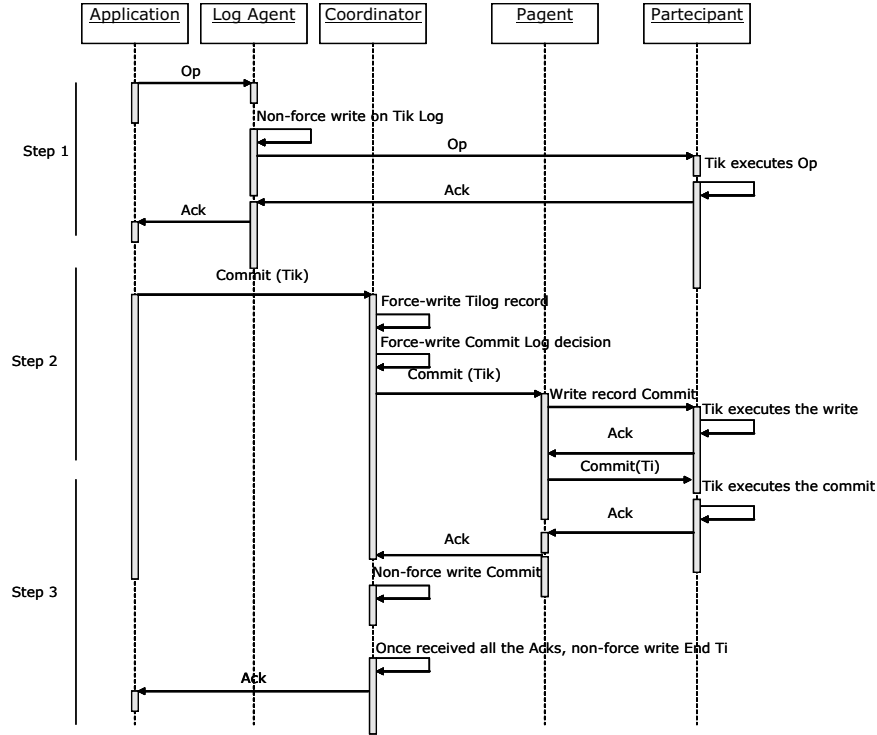


Fig. 6.5. The Unilateral Commit Protocol

1. The Application forwards the transaction branch to be performed to the LogAgent.
2. The LogAgent registers each operation that is to be executed by a non-forced write in its log.
3. Operations are then sent to  $P_k$ , where they are locally executed ( $n$  messages).
4. Participant  $P_k$  acknowledges up to the Application through the LogAgent ( $n$  messages).
5. The Application issues a commit request.
6. The Coordinator takes the commit decision and forces  $T_i$ 's log records and a log commit record, by means of a single blocking I/O (one force). It then broadcasts the commit decision to all participants and waits for their acknowledgments ( $n$  messages).
7. The PAgent asks Participant  $P_k$  to write the commit record.
8. Participant  $P_k$  executes the write and acknowledges up to the PAgent ( $n$  force).
9. The PAgent asks Participant  $P_k$  to actually commit the transaction  $T_i$ .

10. Participant  $P_k$  executes the commit and acknowledges up to its corresponding  $PAgent_k$ .
11.  $PAgent_k$  acknowledges up to the Coordinator ( $n$  messages).
12. The Coordinator performs a nonforced write of  $P_k$ 's acknowledgment related to the commit of transaction  $T_i$ . Then, once all the acknowledgments have been received, a nonforced write is performed and the Coordinator discards all of  $T_i$ 's log records.

In the absence of failures, the entire execution requires only four messages between the Coordinator, the LogAgents, and each of Participant  $P_k$  and  $PAgent_k$  (that is, a total of  $4n$  messages), and  $n + 1$  log forces. Note that if transaction  $T_i$  is to be aborted, the Coordinator discards all of  $T_i$ 's log records and broadcasts an Abort decision message to all Participants. A presumed-abort protocol is assumed. This way, abort messages are not acknowledged and the Abort decision is not recorded in the Coordinator log.

Thus, the UCP exploits a logical logging mechanism (at the Coordinator site), which ensures correct recovery. It also preserves site autonomy and can be applied to heterogeneous transactional systems using different local recovery schemes. The UCP does not require a prepare state nor a corresponding interface on the server side, and it does not require the presence of all servers at commitment time, because of the dictatorial approach used to commit or roll back the transaction. Moreover the UCP is composed of a single message round thereby saving costly wireless communications and it does not increase the communication cost during normal processing (since redo log records are not piggybacked in the messages). It also supports disconnection, since a transaction executed off line can commit as soon as its log has been transferred on the fixed network, and without waiting for acknowledgment from the fixed servers. Details of the choices made in the implementation of the UCP in PoLiDBMS can be found in [72], which can be downloaded from the MAIS Web site [255].

As far as concurrency control is concerned, the standard methods adopted for distributed databases apply here also.

### Semantic Synchronization

Two levels of synchronization need be taken into account in our scenario:

- schema-level synchronization, needed because the database schema available on the portable device must change with changes in the ambient, and
- instance-level synchronization, needed because of data modifications occurring either on the portable device or on the central server.

Let us consider schema-level synchronization first: this situation may arise when another chunk is requested, or when the ambient (`here()`, `now()`, ...) changes and the portable database has to change accordingly.

In the most general case, if there are no storage constraints, a clean copy of the desired chunk instance may be copied to the portable device, after the data has been synchronized between the local and the global database – recall that the global database is formed by the set of all local (fixed or mobile) databases present in the system, whose global schema is assumed to exist and be known. In this case the operation is similar to an initialization of the database on the portable device, and is subject to the same permissions control to verify whether the user is entitled to hold the desired new chunk instance. Of course, optimizations of various kinds can be devised for this situation.

A more complicated and very likely scenario arises when storage constraints are present, as for instance when the microdevice is a smart card. In this case, the system should always take into account the possibility that some data – for example emergency information – might have higher priority over other data to be kept on the microdevice. The problem is solved by introducing the concept of *permanence priority*, meaning that, at design time, we establish that *a certain owner* is given *a certain priority level* with respect to *a certain chunk* to enforce persistency of that chunk in the device's memory. Accordingly, the database schema is associated with a table

PERMANENCE\_PRIORITY(OWNER,CHUNK,LEVEL)

which is used whenever a new chunk is required and the available space is not sufficient.

Thus, the protocol for schema-level synchronization consists of the following operations:

- verify the storage space available on the device;
- if that space is not enough, discard chunks whose priority (with respect to the information owner) is lower than that of the chunks currently on the device;
- upload the required chunk(s).

However, this situation also lends itself to different policies, with respect to the decision about *which of the information at the same priority level should be kept, and which should be discarded*. Here, the notion of *semantic distance* [113, 135] may be used to select information which is *semantically close* to the information held on the device.

Intuitively, the semantic distance is the length of the shortest path connecting two concepts in a conceptual diagram, such as an ER or UML class diagram. For instance, if in an ER diagram patients are related to prescriptions through the concept of a disease, the semantic distance between a patient and a disease is smaller than that between a patient and a prescription; thus, in a situation of storage space shortage, prescriptions might be discarded.

A different scenario occurs when instance-level synchronization is concerned. While at the transaction-related abstraction level we adopted the Unilateral Commit Protocol, at the semantics-related abstraction level we

have to consider priority and rights problems that arise when a user updates the data on a portable device and wants to propagate such an update to other (fixed or mobile) devices(s). This problem has already been examined in the framework of distributed databases, where updated replicas or materialized views can conflict with each other [113, 135], but it becomes more critical when small, mobile devices are involved, since the semantic relationships and dependencies among pieces of information in chunks may again present difficulties.

For example, consider the case where a doctor keeps the set of all patients' prescriptions on his/her device, and where the patients' devices or smart cards also contain their prescription information. Suppose a patient's situation changes from **regular** to **hospital**, and the patient's prescriptions are changed by the hospital staff. The doctor's information remains the same until the two devices are connected again, but in this case, at synchronization time, which is the dominant prescription? One might think that the doctor's word should be taken as the most reliable (and thus the doctor should have the highest priority on the relevant chunk), and this is indeed the general case, but it is not so in the hospitalized situation. Thus, here, semantic dependencies among values of dimensions may affect priority levels between data owners, even in the "simpler" case of instance-level synchronization. Such issues can be resolved again by semantic synchronization protocols strongly based on ownership, or by designing more sophisticated mechanisms, where each update is recorded together with the identity of the actor that performed it, or with the transaction time [360].



<http://www.springer.com/978-3-540-31006-8>

Mobile Information Systems

Infrastructure and Design for Adaptivity and Flexibility

Pernici, B. (Ed.)

2006, XVI, 354 p., Hardcover

ISBN: 978-3-540-31006-8