AN EXTENSIBLE STORAGE MANAGER FOR MOBILE DBMS

Erik Buchmann
Otto-von-Guericke-University of Magdeburg
P.O. Box 4120, D-39016 Magdeburg
Germany
buchmann@iti.cs.uni-magdeburg.de

Hagen Höpfner*
Otto-von-Guericke-University of Magdeburg
P.O. Box 4120, D-39016 Magdeburg
Germany
hoepfner@iti.cs.uni-magdeburg.de

Kai-Uwe Sattler
Otto-von-Guericke-University of Magdeburg
P.O. Box 4120, D-39016 Magdeburg
Germany
kus@iti.cs.uni-magdeburg.de

Abstract The increasing usage of mobile devices like PDAs, laptops, or embedded devices results in a new type of application which must especially consider the strict limitations of the used mobile hardware. One aspect of the application development is the storage and retrieval of data. For non-mobile application this is often efficiently realized with database management systems, which offer standardized interfaces and can be easily integrated into the applications. For mobile devices DBMS are also already available. But existing solutions are not extensible, and therefore, limited to the builtin functionality. That means also, that they include functions which are not always necessary. The optimal DBMS for mobile database systems must allow for the special requirements of its applications in order to reduce the hardware requirements. Thus, it must offer core funtionality which can be extended by additional required features. In this paper, we present a core component of such a customizable DBMS – the storage manager – and describe the architecture as well as the main modules. Furthermore, we show how this modules can be combined in order to address different requirements.

Keywords: Mobile Computing, Storage Management, Customizable DBMS

1. Introduction and Motivation

Due to the increasing usage of mobile computer equipment a new type of application – mobile applications – emerges in importance in comparison to the classical desktop and client server applications. Software for mobile devices must especially consider the limitations (e.g. less memory, small display size, limited power supply) of the mobile devices. On the other hand, users and

*Research supported by the DFG under grant SA 782/3-1.

application developers of such systems want to get nearly the same functionality and comfort like working with non-mobile systems.

An important issue in many applications is efficient data management. Though modern operating systems for mobile devices provide some kind of database support by allowing applications to store records in permanent Flash ROM, often more advanced techniques are required. This includes standard interfaces like JDBC or ODBC, a declarative query and manipulation language, index support for fast access and replication with an enterprise database. Therefore, most DBMS vendors offer smaller versions of their systems for mobile devices, e.g. Oracle 9i lite and IBM DB2 Everyplace. These products mostly provide reduced but fixed functionality, i.e., they are not extensible and customizable. So, some of the offered functions are not necessary for a specific application, and others – non-implemented functions – would be essential.

Specific requirements of mobile database applications exist at several levels of a DBMS. For instance, some applications require query processing. In other applications encryption of stored data is needed, which could be performed transparently by the DBMS. Further examples are transaction support, synchronization with a central database and support for special data types (e.g. geographical objects). If the device offers different types of storage, e.g. Flash ROM or a Microdrive, specific adaptable access modules could increase the performance and reduce the power consumption at the same time.

Obviously, a general "all in one" DBMS cannot fulfill all these requirements, in particular for small devices with limited resources. A viable approach could be a customizable DBMS allowing to choose and combine required modules from the set of available components. Although this idea is not really new – database generators and toolkits have been studied in database research since several years – it plays an important role for mobile DBMS. However, this requires rethinking the functionality of the individual components of a DBMS considering the requirements of the mobile devices.

In this paper, we present first results of our work towards an extensible and customizable DBMS for small devices. We describe the storage manager component of ELORDESS – our *extensible lightweight object-relational DBMS for embedded and small systems*. This storage manager consists of a set of modules which can be combined depending on the requirements of the application. In addition, new modules can be easily plugged in without affecting the application. In this way, storage management functionality can be customized in a wide range and even for individual relations.

The remainder of this paper is structured as follows: After briefly presenting related work, we discuss in section 3 the specific requirements and in section 4 the different approaches for customizable DBMS. Section 5 presents the architecture of the storage manager and section 6 describes the combination of the individual modules for a given example configuration. Finally, we summarize our approach and outline future work.

2. Related Work

There are different approaches to solve the challenges of limited resources, extensibility, modularity and flexibility. Many problems are already addressed in other contexts. Researches in respect of a storage manager providing the features described above overlap with main memory DBMS, extensible database systems, and mobile applications in general.

A hierarchical memory architecture with fast RAM on top and slower harddiscs as secondary storage media have been established on stationary workstations or servers. In contrast embedded or mobile devices often use battery-backed RAM or Flash-ROM in various architectures [Douglis et al., 1994]. Therefore, technologies used in modularized main memory storage managers [Bohannon et al., 1997, Cha et al., 1997] are playing a prominent role. Some hardware components differ in speed, quantity and in their handling. Especially Flash devices need dedicated writing policies to achieve maximum lifetime [Chiang and Chang, 1999]. Variable power requirements of components used in mobile or embedded devices lead to new problems for query processing. For instance, queries can be executed on external servers [Rudenko et al., 1998] or can be optimized considering the energy consumption [Alonso and Ganguly, 1993].

Most applications do not use the whole, but a reduced, varying function set. This kind of applications can be supported by a general purpose database management system which is heavyweight, feature-laden and costly in installation and maintenance, or otherwise by a lightweight, customized system. On lightweight appliances the resource requirements forbid the use of general purpose DBMS.

The classical way extends a DBMS by inserting external supplementary layers on top of the external layer of the three tier architecture presented in [Database Architecture Framework Task Group (DAFTG) of the ANSI/X3/SPARC Database System Study Group, 1986]. But neither requirements of application-specific access methods nor hardware-specific optimizations are satisfied by extending the external layer. Therefore, other approaches introduce a widespread range of extensible architectures allowing the customization of both, internal and external database management system layers by introducing new data types [Stonebraker, 1986] or storage methods.

KIDS [Geppert et al., 1997] uses a broker/services model. A service represents a task or a part of it, which has to be provided by the DBMS. A broker reacts on events and satisfies requests for services. Add-ons for services or brokers provide the extensibility. Another concept introduces a RISC-style DBMS library [Chaudhuri and Weikum, 2000]. A DBMS consisting of an application specific set of RISC-style lightweight components will be faster and needs a smaller amount of memory (like RISC-processors) than other system concepts. Some approaches use small, lightweight database cores like ”Single Schema DBMS” [Batory et al., 1992] and offer interfaces to enhance the core features. DBMS suppliers like Oracle or IBM, which are established with well known products in the database server market, prefer the not extensible and not customizable concept of a small ”general purpose” DBMS for lightweight appliances [Karlsson et al., 2001].

The integration [Geppert and Dittrich, 1994] of these approaches into database systems is a relevant challenge. One solutionr is to use transformations or generator systems [Batory and Thomas, 1995, Sybase, 2000]. Another way is the use of toolkits of reusable components [Chaudhuri and Weikum, 2000] with the ability to be assembled with custom software modules to an application specific system.

This requires to identify different functionality domains which are needed or can be omitted. [Thomas and Batory, 1995] describes – without the claim of completeness – the distinct functionality domains concurrency control, checkpoints and recovery, the ability to use raw devices instead OS files, persistence, databases larger than primary storage, the support for the client/server model or distributed databases, and the computing of dynamic queries, set-oriented queries or joins.

Recapitulating, there are several applicable approaches dealing with extensibility, limited resources and modularity. But none tries to transfer the suitable concepts to an open, lightweight, mobile database management system. We try to close this gap with ELORDESS.

3. Requirements on a Storage Manager for Mobile DBMS

Beyond the general needs for a DBMS, both technical and application specific demands come up for a storage manager applicable on mobile or embedded devices. As shown in section 2, highly different applications need many varying functions in all layers of a DBMS. Database-driven mobile applications can be distinguished between two distinct fields:

Personal information management This field of activity means the “classical” applications for PDA or organizers. Mostly, there is no cooperative work with other users on the same piece of data. The amount of data on the mobile device is usually small, and is at most edited on the mobile device itself. The main challenge for mobile databases for personal information management is *flexibility*.

Replication of large databases The replication of large databases or parts of them – large in the context of mobile, lightweight appliances, e.g. not more than 1 GB – is characterized by the cooperative use of the same data by numerous users. Data are mainly managed and manipulated by DBMS on stationary servers, only a few or no changes are performed on the mobile device. Business applications, geographic information systems or multimedia are typical applications. The most important challenges for mobile databases in this field are *specializability performance*.

Technical requirements result from quality and quantity of the relevant ressources *CPU, memory, network* and *power supply*. The differences between stationary and mobile devices can be described by the following issues:

Quantity In order to meet restictions in weight, size and price, mobile devices are offering significantly less ressource capacities. Therefore, programs for small, lightweight devices must not depend on consuming large amounts of CPU-, memory-, battery- or network resources. Otherwise, if large quantities of some resources are available, they can be used to enhance the quality of service.

Customizability Stationary devices are customizable for the needs of their applications with a broad range of exchangeable hardware components. In contrast mobile devices are at most upgradable with an expansion slot.

Heterogenity Mobile devices are equipped with very heterogeneous resources. For instance, a personal computer uses a x86-CPU, and harddiscs as secondary storage devices. In contrast, mobile devices use completely different central processing units like MIPS, StrongARM or DragonBall, and as secondary storage battery-backed RAM, Flash-ROM, hard discs, network interfaces and so on. The use of heterogeneous hardware requires a flexible, portable architecture that can be customized to deal in an optimal manner with most different devices.

In order to achive a compact system architecture, an ideal storage manager must offer abstract, generalized interfaces which are specializable to any kind of application and hardware component, providing interoperability of existing modules.

4. Approaches for Customizable DBMS

In our approach we decided to create a *toolkit system* to obtain a sufficient compromise between flexibility, extensibility, adaptability and maintainability. A DBMS implementor will be able to assemble a few existing modules very quick to a running system, but also to enhance all modules with extended functionality.

Other concepts lack some features needed by lightweight appliances. The *add-on layer approach* uses a nearly complete DBMS and maps all functionality added to a layer on top of the system with the underlying DBMS. This leads to poor runtime performance and large memory consumption.

Customizable approaches to create database management systems utilize a parameterizable DBMS which can be adjusted by parameters or modified on code level to change its behavior. For this reason very detailed knowledge concerning the specific DBMS and DBMS technology in general is mandatory.

Kernel systems offer a public interface supporting common functionality and hide all other system architectures. Extending the system has to be done by implementing new layers on top of the kernel. New functionality cannot be included in the kernel. Therefore, kernel systems leads to suboptimal database systems.

Generator systems should always produce code which uses resources and satisfies application needs at its best. But unfortunately, generator systems are unable to support a broad range of applications, and extending generator systems itself is a hard task.

We introduce a concept consisting modules. Every module implements one or more distinct services which provide a part of database management system functionality. Because communication across module boundaries leads to some inevitable resource consumption, the suggested modularized architecture offers the option to implement more than one service in a single module.

A module may depend on distinct methods implemented in other modules. For instance an access path module that provides tries needs attributes which implements access on designated parts of the key value. Therefore, a module is characterized by its methods – realized as a Java class – and some interfaces.

The set of modules must be small but applicable to a broad range of DBMS appliances. Components have to be reusable in most usecases. Hence, components are not allowed to be too complex and specific, but rather generally adaptable. Using a large amount of very small modules leads straightforward to reduced CPU performance because extensive parameter passing and method calling. In contrast, defining only a few modules implies poor reusability. For this reason, the only applicable way is obtaining an acceptable compromise.

5. The Storage Manager of ELORDESS: Architecture

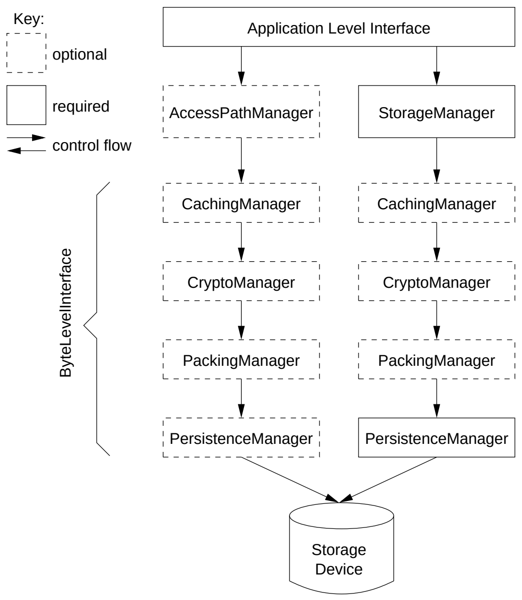

Figure 1. System Architecture

Some tasks of general purpose DBMS should be taken into distinct modules, others have to be implemented in all modules. *Meta-data management* is often described as special task performed by the database catalog. But it is a cross section task that is used by most other tasks. Therefore, we decided to put this functionality in all affected modules.

In a similar way, *transaction management* is utilized in a couple of different tasks and cannot be realized as a separate module. Mobile or distributed transaction management requires flags or time stamps attached on every internal record to solve the concurrent transaction problem. For local transaction processing, every storage module has to be thread-safe and needs methods to delay or release writing of internal records concerned with *commit* and *abort* commands. Because of this, we currently do not implement transaction support.

The architecture is based on three major module interfaces, which are adequate to describe any kind of service supported by ELORDESS. These interfaces are shown as bold-boxed classes in Figure 2 and described as follows:

StorageManager The `StorageManager`-Module has two distinct tasks: first the module transforms data objects of `StorageElement` containers passed through application level to untyped data bytes, which are written to `ByteLevelInterface`, and vice versa. Methods for storing, retrieving and removing data objects perform this task.

Every object held by ELORDESS is typed as `StorageElement`. These objects provide functionality for reading and writing their content from a byte stream. This arrangement is sufficient to obtain the needs of object-oriented database models. For providing object-relational or relational database models, the `StorageElement` class is extended by an array of `Attribute`-objects.

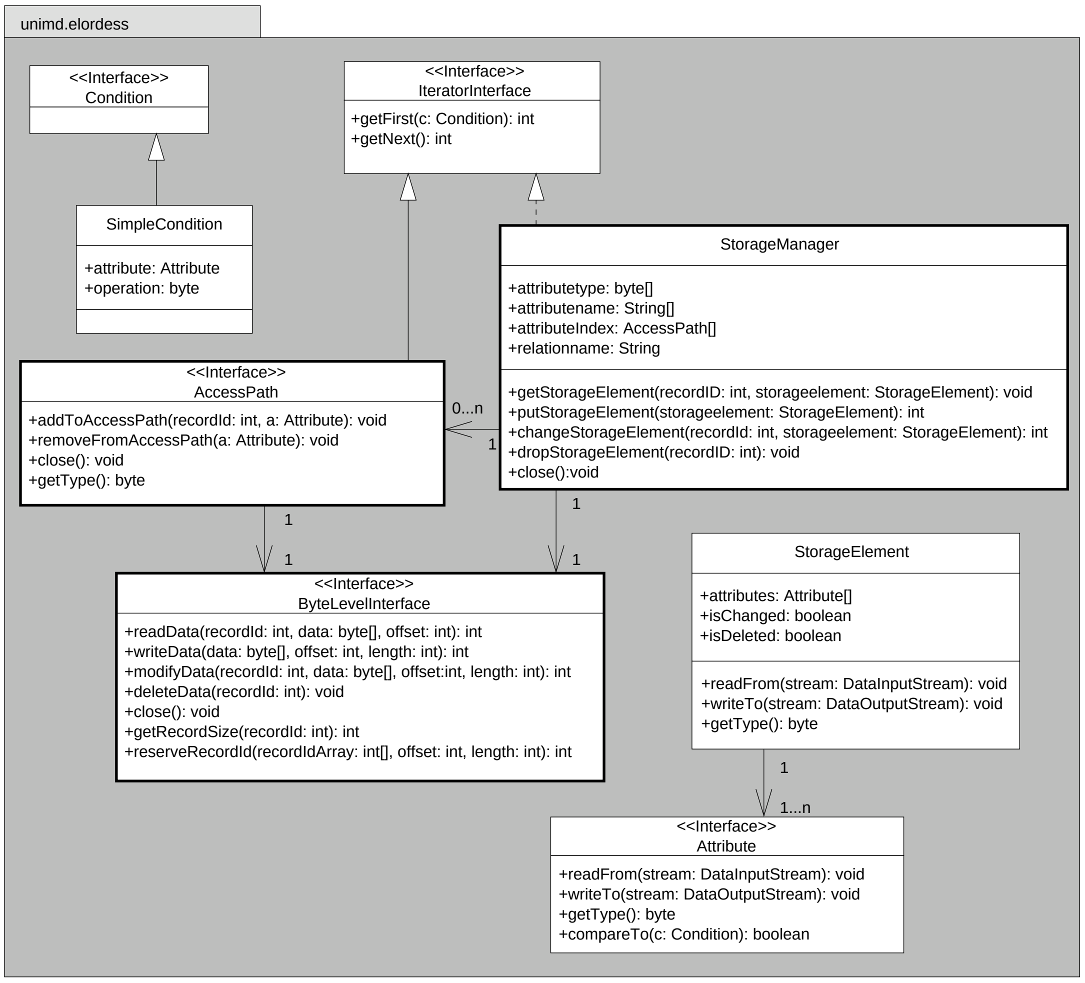
unimd.elordess
<<Interface>>
Condition
<<Interface>>
IteratorInterface
+getFirst(c: Condition): int
+getNext(): int
SimpleCondition
+attribute: Attribute
+operation: byte
StorageManager
+attributetype: byte[]
+attributename: String[]
+attributeIndex: AccessPath[]
+relationname: String
+getStorageElement(recordID: int, storageelement: StorageElement): void
+putStorageElement(storageelement: StorageElement): int
+changeStorageElement(recordId: int, storageelement: StorageElement): int
+dropStorageElement(recordID: int): void
+close():void
<<Interface>>
AccessPath
+addToAccessPath(recordId: int, a: Attribute): void
+removeFromAccessPath(a: Attribute): void
+close(): void
+getType(): byte
0...n
1
1
1
1
1
<<Interface>>
ByteLevelInterface
+readData(recordId: int, data: byte[], offset: int): int
+writeData(data: byte[], offset: int, length: int): int
+modifyData(recordId: int, data: byte[], offset:int, length: int): int
+deleteData(recordId: int): void
+close(): void
+getRecordSize(recordId: int): int
+reserveRecordId(recordIdArray: int[], offset: int, length: int): int
StorageElement
+attributes: Attribute[]
+isChanged: boolean
+isDeleted: boolean
+readFrom(stream: DataInputStream): void
+writeTo(stream: DataOutputStream): void
+getType(): byte
1
1...n
<<Interface>>
Attribute
+readFrom(stream: DataInputStream): void
+writeTo(stream: DataOutputStream): void
+getType(): byte
+compareTo(c: Condition): boolean

Figure 2. Main Classes

Second, the StorageManager holds the catalog information. In the case of relational database architectures the module manages relation names, relation types, attribute names and attribute types. Object-oriented or object-relational DBMS architectures provide names and types for root- and dependent objects. Furthermore, index information has to be accessible.

Stored database objects are only handed out by `RecordID` or in undetermined order for sequential scans. To fetch an object with specified content, an access path defined on the distinguished attribute is required.

AccessPathManager The `AccessPathManager` describes an interface that allows to apply hash-tables, trees, tries or other index-structures to the storage system. The interface specifies abstract methods for adding and removing `Attribute` objects to the access path and retrieving `RecordID`'s for a given condition. The module itself stores all hash-buckets or tree-nodes like other modules using the `ByteLevelInterface`.

Each `AccessPathManager` requires `Attribute` objects implementing special interfaces. The methods specified by these interfaces are used to integrate keys into access paths. For example, B-trees need *lower-than* comparison methods. Hash-indexes require every key independent of its type to be transformed to an integer value which can be used as input for hash function. Tries are unable to perform their job without the ability to obtain a part of the key. Figure 3 shows an example which extends two attributes by supporting methods for hash- and trie-based indexes.

ByteLevelInterface The `ByteLevelInterface` describes an API which works on untyped byte arrays. Methods for reading, writing, deleting and modifying records are provided. Because under certain circumstances – for caching purposes, in disconnected network state, etc. – scheduled or delayed computing of requests will be required, a method for reserving `recordID` enables transparent time-shifted writing.

Every implementor of `ByteLevelInterface` has the permission to split or join given byte pages, as long as it performs the repartitioning transparently for other modules. This may be necessary for cryptographic algorithms which are vulnerable when handling a small amount of data, for mass storage devices with a static block size, or for network protocols with a fixed frame length.

In order to decrease the number of interfaces, tasks like caching, cryptography, or compression working directly on data bytes are specified by the same interface. The interface defines methods for reading, writing, updating and deleting byte arrays. These arrays are read and written to the stable memory by a `PersistenceManager`-class and handled by a couple of different modules like `CachingManager`, `CryptoManager` and `PackingManager`.

CachingManager As described above, supported storage devices show a broad range of characteristics. A `CachingManager` module can be used to support devices with slow read or write speed. Some devices like IBM's microdrive consume a lot of energy by waking up from suspend mode, but sometimes less on normal operations afterwards. This suggests to safe energy by collecting read- or write-requests and performing them in batch mode. In such cases, a special `CachingManager` implementation can be chosen to maximize battery power utilization.

CryptoManager Some applications manage private data which are not allowed to be read by everyone. Particularly mobile devices are vulnerable to be lost or stolen. Hence data security becomes a prominent task on such kind of devices. In this approach an optional `CryptoManager` module supports security management by encrypting and decrypting all written or read data. Varying modules with the same interface can be used to implement distinct security levels by the use of different cryptographic algorithm or different encryption key lengths.

PackingManager On most mobile, lightweight devices only a small amount of memory and low bandwidth network connections are available. Therefore, data compression is an emerging task which will be realized by a separate `PackingManager` module. This module packs the volume of bytes while storing and unpacks it while receiving from other modules. Data compression depends on the processed data and uses different amounts of computing power and main memory. This leads to the need for different compression algorithms performed by different `PackingManager`-modules.

PersistenceManager The secondary storage interface at operating system level is hold by the `PersistenceManager`-Module. This module reads and writes internal records typed

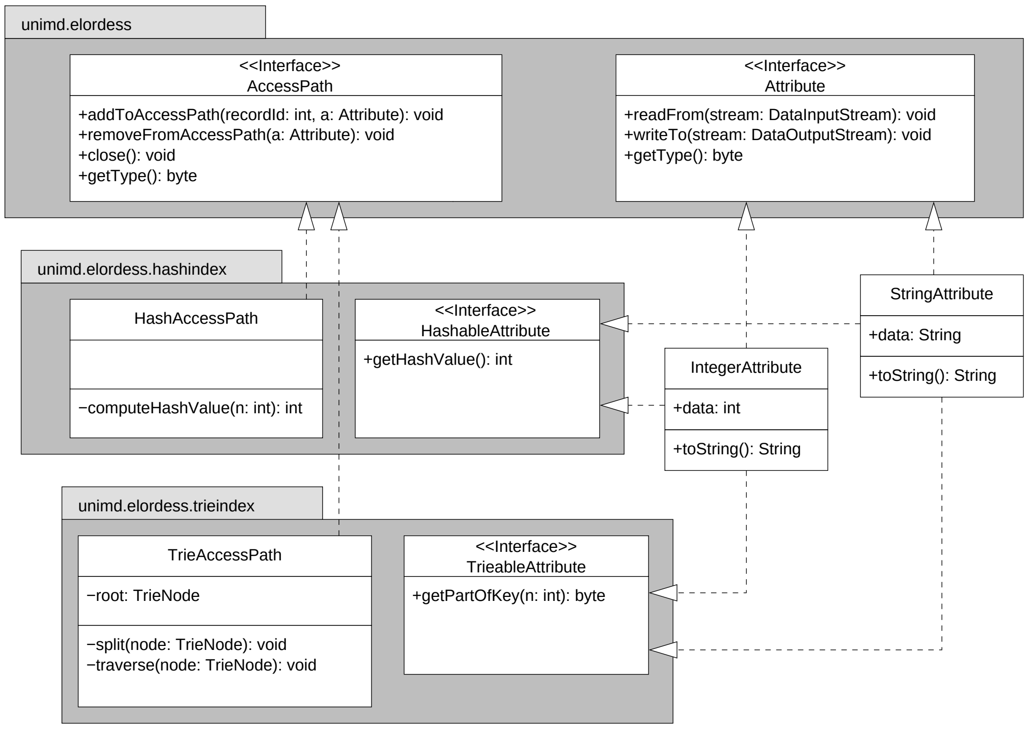

Figure 3. Implementing different access paths.

as `ByteArray` from or to persistent memory. Due to heterogeneity of supported hardware, the `PersistenceManager` is the only direct hardware dependent module class. The `PersistenceManager` offers methods for reading, writing and updating internal records identified by numerical keys named `RecordID`. Operations for opening and closing databases are implicitly invoked by creating or destroying the object instance. Various `PersistenceManager` modules realize persistent storage on networks, disc files, main memory blocks or expansion cards.

The complete set of system modules is shown in Figure 1. Optional modules are depicted as dotted boxes, required modules as solid ones. A minimal configuration required for a working system, consists only of objects of classes derived from `StorageManager` and `PersistenceManager`.

The module configuration is built by the database implementor at design time. At startup, assembling modules is done by the process using the services offered by the modules. This can be performed on static information “hard-wired” in the code. Otherwise, information stored into a designated schema information relation may be used to build up the modules architecture.

6. System Configuration

Using the same interface for compression, cryptography and buffering enables some extended features. If not required, omitting modules is fully transparent on other levels. No request for non-existing functions will be performed on each operation. Secondly, some customization can be done only by changing the order, in which modules are connected among each others. For example, security-challenged applications are able to encrypt the content in the buffer by using the `CryptoManager` “above” to the `CachingManager`.

If used as a service requester to the `CachingManager`, the `CompressionManager` packs the buffer content and saves main memory space, but stresses the CPU resource. Otherwise, under equal circumstances more main memory is needed, but the CPU is utilized less than before. Every

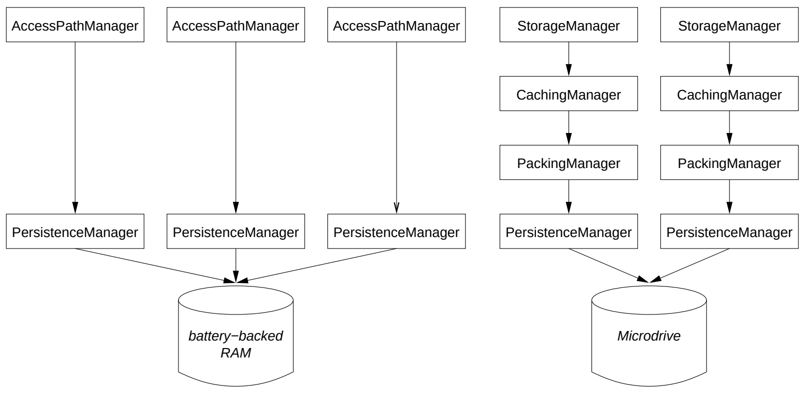

Figure 4. Example system architecture

module is serving only one relation or one access path. If more than one is needed, more object modules have to be created. This concept was chosen to reduce the management complexity.

Figure 4 shows an example system architecture. The hypothetical device offers a large amount of slow, energy-expensive memory on a small harddisc and a little but quick amount of battery-backed RAM. This is a typical constellation for PDAs with expansion slot. All data – in our example represented as two relations – were stored on the Microdrive. To save storage space and achieve better throughput, data will be packed and temporarily buffered. Because storing on harddiscs causes slow access times, indexes are put on battery-backed RAM without any buffering or packing.

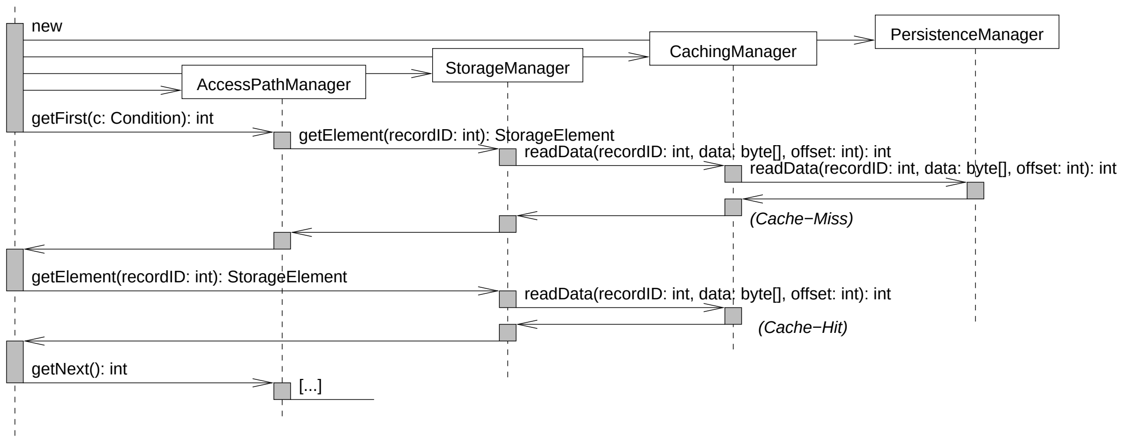
new
PersistenceManager
CachingManager
StorageManager
AccessPathManager
getFirst(c: Condition): int
getElement(recordID: int): StorageElement
readData(recordID: int, data: byte[], offset: int): int
readData(recordID: int, data: byte[], offset: int): int
(Cache−Miss)
getElement(recordID: int): StorageElement
readData(recordID: int, data: byte[], offset: int): int
(Cache−Hit)
getNext(): int
[...]

Figure 5. Set-up and performing sample request.

To explain the functionality of the described system, the processing of a common request is shown. We assume a query on a given attribute using a system with B-tree based access structure. At first, the application process creates the object instances required by the storage manager. Then the application instantiates a condition object describing the requested data items and an empty `StorageElement` object. The condition object is taken as input for the `AccessManager` object. As far as objects exist which match the condition, every call of the `getNext()` method of this module returns a record identifier. This ID is given to the `StorageManager`. The `StorageManager` recursively gets the byte array associated with the record id from a `ByteLevelInterface` module. In fact the `PersistenceManager` is the last called `ByteLevelInterface` and returns the requested byte array. In this way, each module performs its own special task. Finally, the `StorageManager` transforms the given data bytes into typed variables stored in the obtained empty `StorageElement`. Figure 5 shows this process by a sequence diagram.

7. Conclusion and Outlook

In many cases the specific requirements of mobile applications with regard to database support cannot be fulfilled by general purpose database management systems which either try to offer all potentially required functionality or provide only a limited set of functions due to the resource restrictions. One promising approach for solving this problem is an extensible and customizable data management solution enabled to plug in or omit certain modules.

Following this idea, we presented in this paper the storage management component of our DBMS for small and embedded devices. We discussed the overall architecture of this component, which comprises several composable modules implementing specific functions like caching, exploiting access paths or encryption.

In addition, we are currently working on a query engine following a similar approach of customization. In future work, we plan to study techniques for configuration/customization by allowing developers to specify requirements as well as dependencies and using these informations for generating the final system.

References

Alonso, R. and Ganguly, S. (1993). Query Optimization for Energy Efficiency in Mobile Environments. In *Proceedings of the Fifth Workshop on Foundations of Models and Languages for Data and Objects*.

Batory, D. and Thomas, J. (1995). P2: A Lightweight DBMS Generator. Technical Report TR-95-26, University of Texas at Austin, Department of Computer Sciences.

Batory, D. S., Das, D., Singhal, V., Sirkin, M., and Thomas, J. (1992). Database Challenge: Single Schema Database Management Systems. Technical Report CS-TR-92-47, University of Texas, Austin.

Bohannon, P., Lieuwen, D. F., Rastogi, R., Silberschatz, A., Seshadri, S., and Sudarshan, S. (1997). The Architecture of the Dali Main-Memory Storage Manager. *Multimedia Tools and Applications*, 4(2):115–151.

Cha, S. K., Park, J., and Park, B. D. (1997). Xmas: An Extensible Main-Memory Storage System. In Golshani, F. and Makki, K., editors, *Proceedings of the 6th International Conference on Information and Knowledge Management (CIKM-97)*, pages 356–362, New York. ACM Press.

Chaudhuri, S. and Weikum, G. (2000). Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In El Abbadi, A., Brodie, M. L., Chakravarthy, S., Dayal, U., Kamel, N., Schlageter, G., and Whang, K.-Y., editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10–14, 2000, Cairo, Egypt*, pages 1–10, Los Altos, CA 94022, USA. Morgan Kaufmann Publishers.

Chiang, M.-L. and Chang, R.-C. (1999). Cleaning policies in mobile computers using flash memory. *The Journal of Systems and Software*, 48(3):213–231.

Database Architecture Framework Task Group (DAFTG) of the ANSI/X3/SPARC Database System Study Group (1986). Reference Model for DBMS Standardization. *ACM SIGMOD Record*, 15(1):19–58.

Douglis, F., Kaashoek, F., Li, K., Cceres, R., Marsh, B., and Tauber, J. A. (1994). Storage Alternatives for Mobile Computers. In *First Symposium on Operating Systems Design and Implementation*, pages 25–37, Monterey, Californie, US.

Geppert, A. and Dittrich, K. R. (1994). Constructing the next 100 database management systems: like the handyman or like the engineer? *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(1):27–33.

Geppert, A., Scherrer, S., and Dittrich, K. R. (1997). KIDS: Construction of Database Management Systems based on Reuse. Technical Report ifi-97.01, Department of Computer Science, University of Zurich.

Karlsson, J., Lal, A., Leung, C., and Pham, T. (2001). IBM DB2 Everyplace: A Small Footprint Relational Database System. In *17th International Conference on Data Engineering (ICDE' 01)*, pages 230–234, Washington - Brussels - Tokyo. IEEE.

Rudenko, A., Reiher, P., Popek, G., and Kuenning, G. (1998). Saving Portable Computer Battery Power through Remote Process Execution. *Mobile Computing and Communications Review*, 2(1):19–26.

Stonebraker, M. (1986). Inclusion of New Types in Relational Data Base Systems. In *Proceedings of the International Conference on Data Engineering,*, volume IEEE Computer Society Order Number 655, pages 262–269, Los Angeles, CA. IEEE Computer Society Press.

Sybase (2000). The Next Generation Database for Embedded Systems. Whitepaper.

Thomas, J. and Batory, D. (1995). P2: An extensible lightweight DBMS. Technical Report CS-TR-95-04, The University of Texas at Austin, Department of Computer Sciences, Austin, Texas.