Data Management for Embedded Systems: A Cell-based Approach

Syed Saif ur Rahman[1], Marko Rosenmüller[1], Norbert Siegmund[1], Sagar Sunkle[1], Gunter Saake[1], Sven Apel[2]
[1]School of Computer Science, University of Magdeburg, Germany.
{srahman, rosenmue, nsiegmun, sagar.sunkle, saake}@ovgu.de
[2]Department of Informatics and Mathematics, University of Passau, Germany.
apel@uni-passau.de

Abstract

Existing data management systems are very complex and provide a multitude of functionalities. Due to complexity and their monolithic architecture, these data management systems are not suitable for data-centric embedded systems. In order to cope with complexity of data management in such systems, we propose a novel approach to DBMS architecture, called Cellular DBMS*, that is inspired by biological systems.* Cellular DBMS *is composed of multiple customizable embedded database instances, called* Cells*. We illustrate how multiple atomic cells can be used together to provide scalable, resource efficient data management for embedded systems.*

1. Introduction

Data-centric embedded systems (e.g., Sensor Networks) have converged into our daily life. An increasing number of devices are interacting with us to fulfill our very basic needs in an invisible way. All these interactions produce data that need to be managed and processed in an efficient manner. Existing data management solutions are complex and provide many functionalities that are never used. These solutions identically manage all kind of data on heterogeneous hardware. Even a small data management component of such a system is quite large and possesses dependencies on other functionalities.

We propose a DBMS architecture that is inspired by biological systems like tissues of cells or swarms of animals. Our goal is to generate a complete database management system (DBMS) by instantiating multiple customizable databases embedded in a larger DBMS. A cell represents the embedded DBMS customized for a certain task and restricted in its functionality. It works independently or in collaboration with other cells. Cellular DBMS is the superset of the functionalities of all embedded cells including management capabilities for scaling and balancing data accesses and self-tuning concepts. Decisions about functionalities a cell must consist have to be based on the hardware and the nature of the data on the used embedded device.

This work is motivated by the work of Leich et al. [10] who have shown that data management can be tailored according to the needs of embedded devices. Our implementation is based on FAME-DBMS[1] [15], a customizable DBMS for embedded systems that we use to generate the cells. However, the general concept is independent of the used customizable DBMS. In this paper, we outline the design principles of Cellular DBMS and describe the architecture with an example.

2. Motivation

Contemporary DBMS have been developed decades ago and have evolved over time in order to handle large amounts of data on high-end server systems. These systems are complex and provide many functionalities that are tightly coupled with their monolithic engine. Functionalities also possess interdependency, making it difficult to assess the impact of change in one functionality on another. In order to cope with the ever raising complexity of DBMS, Chaudhuri and Weikum argue for a transition from a monolithic DBMS to a DBMS with simple and reusable components of limited functionality and clean inter-component interaction [16]. Cellular DBMS addresses these needs by proposing an architecture of reusable components represented by Cells.

This work is motivated by the work of Leich et. al who proposed to build customizable data management for embedded systems which are for example needed in a wireless sensor network scenario [10]. Sensor networks are important data-centric systems with hardware and software heterogeneity as depicted in Figure 1. Hardware in sensors networks may vary from 8 bit motes to 32 bit microservers with program memory that can vary from 48 kB to 512 kB, whereas data memory may vary from 4 kB to 64 kB [4]. Each node varies in terms of processing power and mem-

[1] http://www.fame-dbms.org/

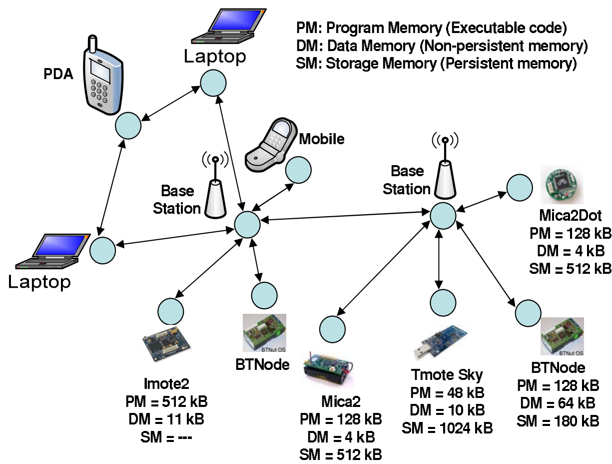

Figure 1. Sensor network scenario.

ory configuration. Considering extreme resource scarcity and high heterogeneity as discussed above, the motivation of our proposed architecture is to make the best use of available resources and exploit the hardware heterogeneity for efficient data management.

As we mentioned above, the needed functionality of a cell depends on the data that is stored on the device. Our classification consists of 4 types: *standing data*, *setup data*, *transactional data*, and *aggregated data*. *Standing data* is generated during deployment and is never changed during its lifetime but often read, e.g., fixed time intervals for sensing the environment. *Setup data* is also initialized during deployment but may be subject to change during its lifetime. For example, this is needed for routing information in a wireless sensor network. If nodes fail due to limited power, new neighbor nodes have to be registered for communication. *Transactional data* is generated during data operations (e.g., add, update, remove) and is often changed during its lifetime, e.g., sensed data in a sensor network. *Aggregated data* is the result of some processing operations. This kind of data can be found on aggregation nodes in a sensor network. Each type of data has its own characteristics in terms of usage frequency and size of data, as shown in Table 1. Since the nature of data is varying and thus also the requirements on data management change, we argue that specialized data management is needed to process each kind of data, i.e., using different types of DBMS cells. Since different kinds of data may occur on a single node we propose to build data management for nodes using a Cellular DBMS that consist of multiple individual cells, each optimized for a particular task. Since data is distributed over multiple nodes in a sensor network, DBMS cells also have to be distributed over nodes and collaborate to build a compound DBMS.

Nature of Data	Standing	Setup	Transactional	Aggregated
Data Size	Small	Small to Medium	Medium to Large	Medium to Large
Read Frequency	High	High	Medium to High	Low to High
Write Frequency	No Write	Low	Medium to High	Low to Medium

Table 1. Data categorization

3. DBMS Product Lines

Software Product Line (SPL) engineering is an efficient and cost-effective approach to produce a family of related programs for a domain [14]. A product line shares a common set of features developed from a common set of software artifacts [2]. It has been shown that a high degree of customizability makes an SPL a suitable candidate for the development of DBMS for embedded systems [10]. Feature-oriented programming is a paradigm for developing software product lines where programs are synthesized by composing features [1]. A feature can be defined as ”A distinguishable characteristic of a concept that is relevant to some stake-holder” [8]. When an SPL is designed in terms of features, creating a program is simply the selection of the required features and composition of the according feature modules [1].

In our approach we propose the use of multiple customizable embedded databases that can be combined to build one larger DBMS. For this purpose, we use FAME-DBMS, a highly customizable embedded database management software product line developed for deeply embedded systems [15]. FAME-DBMS is based on feature-oriented programming. It untangles and modularizes DBMS functionalities as features. A decomposition of DBMS into features, i.e., the functionalities individual DBMS differ in, allows a developer to generate a tailor-made DBMS based on the selection of required features [10]. These different variants are built from the same code base as depicted in Figure 2 for a Cellular DBMS product line. Based on such an SPL, multiple heterogeneous DBMS cells can be generated [15].

4. Design Principles for Cellular DBMS

In our proposed DBMS architecture, a Cellular DBMS consists of many customized atomic embedded databases, i.e., Cells. The behavior of the complete DBMS depends on the collective behavior of all atomic cells. Each DBMS Cell may be customized to perform different tasks. What cells a DBMS contains and what functionalities each cell has, depends on the application scenario.

4.1. Generating DBMS Cells

Each cell is based on RISC-style architecture with simple and limited functionality [16]. The motivation behind

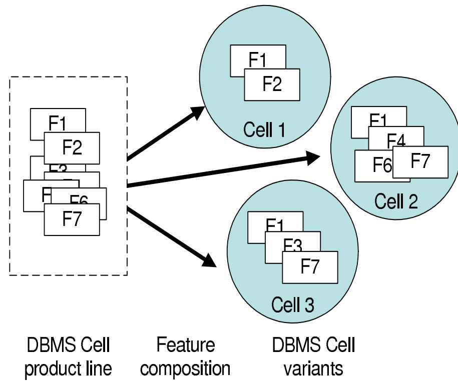

Figure 2. Generating different DBMS cells by composing features (F1–F7) of a DBMS cell product line.

this approach is to ensure that a DBMS can be reduced to a fine-grained atomic unit (i.e., a cell) with predictable behavior and reduced complexity [16]. This approach enables us to assess the behavior of a complete DBMS by accumulating the behavior of all cells. Each cell should be generated according to the nature of stored data, the computation environment, or, more generally, according to needs. Decisions about cell composition require a detailed analysis of data, hardware, software, environment, and usage scenario. Each cell can handle key/value pairs stored in a single table or multiple tables, maintain a data dictionary, and collects data statistics.

For implementation of Cellular DBMS, we used FAME-DBMS [15] and modified it. The *feature model* of Cellular-DBMS, as shown in Figure 3 [2], is based on the FAME-DBMS feature model. It describes the features of an SPL and their relationships [8]. As depicted, the implementation of Cellular DBMS contains of four main features, i.e., In-Memory Data Management, Buffer Manager, Access Path, and OS Abstraction. Each functionality can be implemented differently to achieve benefits, e.g., better performance, described as alternative features. For example, feature Index provides two variants for effective data access, i.e., B+Tree and Hash. This enables us to generate specialized cells by selecting one feature or the other.

Functionality for storing data in a table is provided by feature In-Memory Data Management. This feature contains the functionality of an in-memory embedded database and can alone be used to construct a simple DBMS cell. It performs data management operations in an in-memory environment and does not have any unneeded persistence functionality, resulting in good performance in terms of fast operations [6]. Most sensor nodes are equipped with storage memory that can be used to store data persistently. To scale a Cellular DBMS cell for such nodes feature Persistence is used.

[2]Shown is an excerpt of the feature model with modified features/concepts required for discussion.

The simplest cells, consisting only of feature In-Memory Data Management, support exactly one table. Support for multiple tables can be provided by using multiple DBMS cells, as described below, or by *cloning* the In-Memory Data Management feature [3]. Cloning a feature means to create multiple instances of it. For example, to support two tables we have to create two instances of the In-Memory Data Management feature and each instance handles one of the tables. Whether a feature can be cloned is depicted with cardinalities in Figure 3. For example, there has to be at least one instance of the In-Memory Data Management feature but an arbitrary number of instances is allowed.

Our architecture restricts a cell to handle only a single kind of data, e.g., aggregated data. If a cell supports multiple tables, only the same kind of data should be stored in these tables. This ensure optimization of each cell according to the stored data. If different kinds of data need to be stored, multiple cells have to be used, as we explain in the following.

4.2. Using Multiple Cells

Deployment of cells depends on the data, the available nodes, as well as the hardware of nodes. In the simplest scenario, only a single cell is deployed on an individual node. In a more complex scenario, multiple cells are deployed on a single node or distributed over multiple nodes in a network. The reason for using multiple cells on a single node is the limited functionality of each cell. For example, to store different kinds of data on a single node we have to use different cells.

Composite Cells. In order to avoid code replication when using multiple cells on a single node, we introduce the concept of *composite cells*, which are multiple cells composed into the same binary program or that use the same core implementation provided as a library. Composite cells allow us to reuse the program code between different cells on a single node. As an example for a composite cell consider a node of a sensor network supporting storage in data memory as well as in storage memory. A composite cell supporting this scenario is built by composing two cells, one in-memory cell and one cell for persistent storage. Using this approach we are able to observe the behavior of complete data management on a single node or even individual cells of this node. Each cell of such a composite cell is optimized according to the storage scenario (e.g., using a B+tree for persistent storage) but at the same time reuses the binary code, e.g., the In-Memory Data Management, of other cells.

Achieving code reuse for cells that do not differ in functionality is not different from reusing an embedded DBMS deployed as a library. This, however, is a challenging task when differently configured cells are used and it has to be addressed in future research. The reason why this is hard to achieve is that the program code that has to be executed depends on the configuration of a cell. Well-known concepts like frameworks and plugins could be used to achieve this but are not applicable to embedded systems. Hence, new concepts for achieving proper code reuse for embedded systems have to be found and integrated with current product line engineering techniques.

Composite cells can not only be built from ordinary cells but also from composite cells which results in higher level composite cells. The reason for such an architecture is to provide a hierarchy of data management functionalities. For example, a lower level composite cell might be used to support different tables and higher level composite cells might be used to group lower level cells according to the used storage (e.g., persistent and in-memory). This way it is possible to decompose and structure a complex data storage scenario.

Distributed Cells. As stated above, cells are not only limited to one device, e.g., a node of a sensor network, but can also be distributed over different devices via a network. In a complex distributed sensor network scenario, cells are deployed on multiple nodes and operate concurrently. On each node of such a distributed scenario, a single cell might be used or a composite cells might provide more complex data management. While composite cells may be spread over multiple processes on a single node and interact via direct API calls, distributed cells are spread over multiple nodes in the network and interact through API calls via a network. For distributed deployment, we envision a Cellular DBMS using a global data dictionary and statistics. However, it has to be further analyzed how distributed deployment of interacting cells can be achieved using a software product line of DBMS Cells.

Clean API and Interaction. Providing a consistent API for simple as well as for composite cells is the most important design criteria required for communication between cells. Hence, two communicating cells should not care about the concrete type of another cell. On the other hand, simple cells provide only limited data management functionality and should exhibit a simple API that reflects the limited functionality. This is in contrast to a consistent API and has to be considered when generating cells.

As a solution, we use two different mechanisms. First, we only allow interface extensions but not modifications of an interface. For example, a feature might add a method to the interface of the DBMS but is not allowed to modify the signature of a method. This ensure upward compatibility, i.e., we can use cells with a more complex API when cells with a simple API are expected. The second approach is to generate wrappers for simple cells when more complex cells are expected. For example, if a method for creating an index is expected from an in-memory cell without index support, an empty wrapper method can be generated to provide this method. Wrappers are only used to achieve downward compatibility and more complex wrappers might be required. Furthermore, it has to be analyzed for which scenarios it is not possible to generate such wrappers.

4.3. Resource Balancing

Cell Mobility. In Cellular DBMS we propose the concept of mobile cells. Cell mobility means the capability of Cellular DBMS to move a cell from one processing environment to another. Mobility of cells could be across processes on single system or across systems connected via network. The motivation behind mobility is load balancing and to use resources efficiently. Currently in sensor network scenario, this concept is not viable due to resource constraint, i.e., low communication speed and high power consumption during communication. However, for embedded devices with relatively high speed communication and larger battery power this concept is important for efficient utilization of resources. Cell mobility can be used in many different ways. One scenario is that the embedded device on which a cell is deployed, is heavily loaded with processing. We envision to move that cell to other device that is relatively idle. If all devices are over-consumed, then a new device can be brought into network and then cells can be moved to that device for load balancing.

Virtual Resources. Embedded systems are different from high-end systems by means of resources. In embedded system we normally have resource constraints on a single device but in distributed network of interacting embedded system there are many resources that are available across network and are idle. Cellular DBMS virtually combine these scattered resources as Virtual Resource, i.e., it gives a virtual view of scattered small resource across embedded devices as one single large resource. For example, on three embedded devices we have 10 kB, 6 kB, and 13 kB of free memory. Now if we have to store data that is 18 kB large, none of these devices have enough capacity on its own. In this case, Cellular DBMS approach is capable of storing data distributed across cells on these devices and to give an application a view of one single large resource capable of accommodating 18 kB of data. This concept also gives us a clue that how Cellular DBMS can use cells for fragmenting data on multiple embedded devices or sensor nodes.

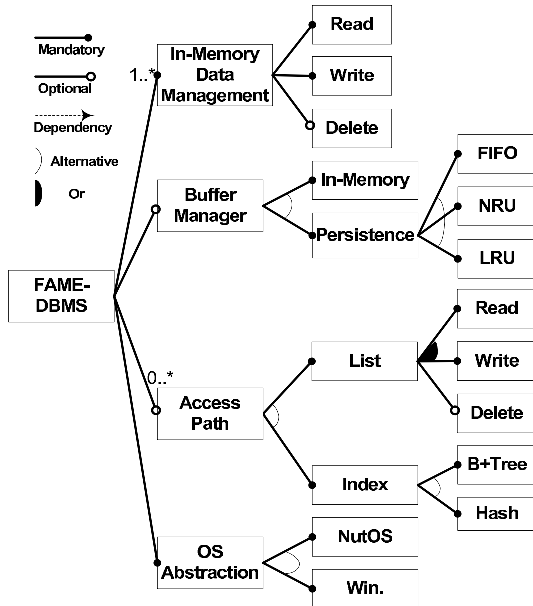

Figure 3. Cellular DBMS feature model.

Features	In-Memory Data Management			Persistence	List			Index	Binary Size (.ELF) Compiler: avr-g++
	Read	Write	Delete	LRU	Read	Write	Delete	B+Tree	
Cell A	X	X	X						32 KB
Cell B	X	X	X	X	X				42 KB
Cell C	X	X	X	X	X	X	X		45 KB
Cell D	X	X	X	X				X	48 KB

Figure 4. Binary size for different Cellular DBMS cells.

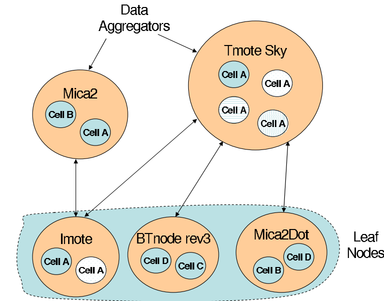

Figure 5. Sample deployment of different Cellular DBMS cells.

5. Discussion

The main concept behind Cellular DBMS is the capability of instantiating and using multiple instances of customizable embedded database cells. We have implemented a Cellular DBMS prototype using the FAME-DBMS SPL. Naturally, different DBMS cells overlap in their functionalities. For example, all DBMS cells perform basic data operations like searching and storing. The main advantage of using an SPL is to maintain the same code-base for common functionalities, and reuse the common functionalities in all cells and to provide specific functionalities, e.g., persistent storage capabilities, only for those cells that need them. In order to discuss the proposed architecture, a Cellular DBMS prototype has been developed by using modified FAME-DBMS prototype as cell. For discussion of our proposed architecture, we will consider storage memory, and program memory as parameters of interest. To explain the idea, how specialized DBMS cells can be beneficial for data-centric embedded systems, four types of DBMS cells are generated based on different feature selections using FAME-DBMS prototype as shown in Figure 4 [3]. For each cell, the binary size is different and depends on the selected features of FAME-DBMS prototype. Each cell is a candidate for a different type of node based on the available program and storage memory as well as type of data it handles. Cell A is suitable for nodes without any storage memory, e.g., Imote node. Cell D is suitable for nodes with relatively large data and storage memory, e.g., BTnode rev3. A sample deployment of these customized cells based on a node's resources is shown in Figure 5. In the sample deployment, Tmote sky contains the smallest program memory and can only handle small cells like Cell A. However, it also contains largest storage memory making Cell B, C and D a good candidate for storing relatively large data on storage memory. In contrast, Imote contains the largest program memory but lacks storage memory, again making Cell A best candidate for deployment. BTnode rev3, Mica2 and Mica2Dot all contain moderate program and storage memory. We argue that, in the demonstrated sample deployment, the Cellular DBMS is a promising solution. The Cell implementation is light-weight and allows for deployment of multiple heterogeneous cells on a single node, enabling specialized handling of data based on available resources and the nature of data.

6. Related Work

Reconfigurability and customizability of software is a main concern in the embedded domain. There have been many approaches [9, 11, 7] to achieve customizability in the embedded domain but all these approaches have high resource requirements. In contrast, an SPL approach used in FAME-DBMS [15] is different from all above-mentioned approaches and promises benefits for the embedded domain as proposed by Leich et. al [10]. We based on this work and extended the idea of customizing a single DBMS to mul-

[3]Binary size contains additional overhead of dependencies and may vary in future work.

tiple, specialized DBMS acting together as a large DBMS. For wireless sensor networks, TinyDB [17] is a query processing system for extracting information from sensor network. Similarly, the COUGAR System main intent is query processing in an ad hoc sensor networks [5]. We argue that TinyDB and COUGAR does not provide the needed configurability as could be achieved with our approach. Berkeley DB is a customizable embedded database system that supports key/value storage, in-memory operation, and provides a small footprint [12] and could be used as a Cell but not as the overall Cellular DBMS consisting of management functionalities to balance queries, etc. Related approaches can also be found in the area of distributed databases [13]. However, the used functionality does not scale down to embedded devices or take customizability into account. Furthermore, the concept of a cell based approach presented in this paper is unique in its own.

7. Conclusion and Future Work

We proposed a novel DBMS architecture based on composition of multiple cells which are atomic, customized embedded DBMS. As demonstrated, these cells provide restricted data management functionality and collaborate to constitute one large *Cellular DBMS*. This cell based approach ensures predictable behavior and efficient utilization of resources by keeping the cells simple. For implementation, we extended FAME-DBMS which uses a software product line approach. As future work, we are going to develop concepts for distribution of heterogeneous interacting DBMS cells. Additionally, we want to identify how balancing data accesses and self-* capabilities concepts can be integrated into Cellular DBMS.

Acknowledgment

Syed Saif ur Rahman is funded by Higher Education Commission of Pakistan and NESCOM, Pakistan. Marko Rosenmüller and Norbert Siegmund are funded by the German Ministry of Education and Science (BMBF), project 01IM08003C. Apel's work is being supported in part by the German Research Foundation (DFG), project number AP 206/2-1. The presented work is part of the ViERforES[4] project.

[4]http://vierfores.de

References

[1] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling stepwise refinement. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[2] P. Clements, L. Northrop, and L. M. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, August 2001.

[3] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged configuration using feature models. volume 3154, pages 266–283, 2004.

[4] J. Elson, S. Bien, N. Busek, V. Bychkovskiy, A. Cerpa, D. Ganesan, L. Girod, B. Greenstein, T. Schoellhammer, T. Stathopoulos, and D. Estrin. Emstar: An environment for developing wireless embedded systems software. Technical report, March 25 2003.

[5] W. F. Fung. Cougar: the network is the database. In *In SIGMOD Conference*, pages 621–621. ACM Press, 2002.

[6] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Trans. on Knowl. and Data Eng.*, 4(6):509–516, 1992.

[7] F. Golatowski, J. Blumenthal, M. H, M. Haase, H. Burchardt, and D. Timmermann. Service-oriented software architecture for sensor networks. In *Proceedings of the International Workshop on Mobile Computing*, pages 93–98, 2003.

[8] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[9] J. Koshy. Synthesizing Scalable System Software for Wireless Sensor Networks. *Computer*, 2007.

[10] T. Leich, S. Apel, and G. Saake. Using Step-Wise Refinement to Build a Flexible Lightweight Storage Manager. volume 3631 of *Lecture Notes in Computer Science*, pages 324–337. Springer Verlag, 2005.

[11] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 85–95, New York, NY, USA, 2002. ACM.

[12] M. Olson, K. Bostic, and M. Seltzer. Berkeley db. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, pages 183–192, 1999.

[13] M. T. Ozsu. *Principles of Distributed Database Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[14] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 1 edition, 2005.

[15] M. Rosenmüller, N. Siegmund, H. Schirmeier, J. Sincero, S. Apel, T. Leich, O. Spinczyk, and G. Saake. Fame-dbms: tailor-made data management solutions for embedded systems. In *SETMDM '08: Proceedings of the 2008 EDBT workshop on Software engineering for tailor-made data management*, pages 1–6, New York, NY, USA, 2008. ACM.

[16] G. Weikum, A. Moenkeberg, C. Hasse, and P. Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *Proceedings of the 28th VLDB Conference*, Hongkong, China, 2002.

[17] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(3):9–18, 2002.