Cellular DBMS: Customizable and autonomous data management using a RISC-style architecture

Dissertation
zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von: M.S. Syed Saif ur Rahman
geb. am 23. December 1980 in Hyderabad, Pakistan

Gutachter:
Prof. Dr. Gunter Saake
Prof. Dr. Kai-Uwe Sattler
Dr. Stefan Manegold

Ort und Datum des Promotionskolloquiums: Magdeburg, den 06. September 2011

Rahman, Syed Saif ur:
Cellular DBMS: Customizable and autonomous data management using a RISC-style architecture
Dissertation, Otto-von-Guericke-Universität Magdeburg, 2011.

Abstract

Database management systems (DBMS) were developed decades ago with consideration for the legacy hardware and data management requirements. Over years, developments in the hardware and the data management have forced DBMS to grow in functionalities. These functionalities got tightly integrated into the DBMS core because of their monolithic architecture. This has resulted in increased complexity of DBMS, which makes them difficult to tune for consistent performance. Furthermore, the decreasing cost of the hardware and the software has resulted in making the human resource a major factor in the total cost of ownership for the data management. There exists a need to revisit existing database architecture using unconventional and unexplored techniques towards more diversified and loosely coupled architectures.

We present *the Cellular DBMS architecture*, which is designed according to the *RISC-style self-tuning database architecture* proposed by Chaudhuri and Weikum in their VLDB 2000 paper. The Cellular DBMS architecture proposes to construct a large DBMS by using multiple RISC-style cells in concert, where each cell is atomic, customized, and autonomous instance of an embedded database. Using the Cellular DBMS architecture, we designed and implemented a customizable and self-tuning storage manager; we termed as *Evolutionary Column-oriented Storage (ECOS)*. ECOS supports the storage model customization at table-level using different variations of the decomposed storage model. It supports the storage structure customization at the column-level using *evolving hierarchically-organized storage structures*. These storage structures automatically evolve themselves with the growth of data considering the workload. Their evolution behavior is defined using *evolution paths*. The Cellular DBMS architecture uses innovative software engineering approaches, such as the software product line, the feature-oriented programming, and the aspect-oriented programming to realize customization and autonomy. We implemented the Cellular DBMS prototype constituting the ECOS storage manager in C++ using FeatureC++ and AspectC++ tools. We evaluated our prototype implementation using a custom micro benchmark to show the benefits of our proposed architecture.

Zusammenfassung

Die Entwicklung der heutigen Datenbankmanagementsysteme (DBMS) begann vor Jahrzehnten, wobei die damalige Hardware und die damaligen Anforderungen an das Datenmanagement zugrunde gelegt wurden. Während der letzten Jahrzehnte haben sich aber sowohl die Hardware als auch die Anforderungen verändert. Dies zwang die Hersteller von DBMS, die gebotene Funktionalität der bis dahin existierenden DBMS auszuweiten. Über die Zeit wurden mehr und mehr Funktionen hinzugefügt, und diese sind wegen der monolithischen Architektur dieser DBMS tief in den Systemkern integriert und miteinander verflochten. Heutige DBMS haben sich zu komplexen Systemen entwickelt, und die vielfältigen abhängigen Funktionalitäten erschweren die Optimierung mit dem Ziel einer konsistenten Performanz. Der Einfluss einer Tuning-Maßnahme auf andere ist schwer abzuschätzen. Außerdem haben sinkende Hardware- und Software-Kosten dazu geführt, dass die Kosten für Administratoren und DBMS-Fachleute ein dominierender Kostenfaktor innerhalb der gesamten Betriebskosten für die Datenverwaltung wurden. Deshalb ist es notwendig, existierende Datenbankarchitekturen zu überdenken und dabei unkonventionelle und bisher unerforschte Techniken zu verwenden, und diese in Richtung breiter gefächerter und lose gekoppelter Architekturen zu entwickeln. Ein solcher Ansatz ist die RISC-style Self-Tuning Datenbankarchitektur, welche von Chaudhuri und Weikum in ihrer VLDB 2000 Arbeit vorgeschlagen worden ist.

Wir haben die Cellular DBMS-Architektur vorgeschlagen und vorgestellt, welche die RISC-style Self-Tuning Datenbankarchitektur umsetzt. Diese Architektur setzt große DBMS um, indem viele Instanzen von RISC-style Zellen (Cells) zusammengesetzt werden, wobei jede Zelle atomar, angepasst und autonom in ihrer Datenmanagementfunktionalität ist. Wir präsentieren einen anpassbaren Speichermanager mit Self-Tuning-Funktionalität, der entsprechend der vorgeschlagenen Cellular DBMS-Architektur implementiert wurde. Den vorgestellten Speichermanager bezeichnen wir als Evolutionary Column-oriented Storage (ECOS). ECOS unterstützt die Anpassbarkeit auf Tabellen- und auf Spaltenebene. Für die Anpassung auf Tabellenebene haben wir vier Varianten des Decomposed Storage Model (DSM)-Modells vorgeschlagen. Zur Anpassbarkeit auf Spaltenebene verwenden wir hierarchisch organisierte Speicherstrukturen, welche als kleinste und minimale Speicherstrukturen für die anfängliche kleine Datenverwaltung initialisiert werden, um sich dann automatisch entsprechend der Datenbankgröße und den Anforderungen des aktuellen Workloads evolutionär zu entwickeln. Wir haben das Konzept des Evolutionsp-

fades (Evolution Path) vorgestellt, welches festlegt, wie sich Speicherstrukturen entwickeln können. Das Design der Cellular DBMS-Architektur legt die Verwendung innovativer Software Engineering-Ansätze nahe, wie zum Beispiel Softwareproduktlinien (SPL), Feature-orientierte Programmierung (FOP) und Aspektorientierte Programmierung (AOP). Wir haben den Cellular DBMS-Prototyp inklusive des ECOS Speichermanagers in C++ unter Verwendung von FeatureC++ und AspectC++ Werkzeugen implementiert. Die prototypische Implementierung wurde unter Verwendung eines angepassten Micro Benchmarks evaluiert, um die Vorteile der vorgeschlagenen Architektur zu demonstrieren.

Dedications

To my lovely parents, wife, siblings, and friends.

Acknowledgments

I am thankful to Prof. Dr. Gunter Saake, who gave me the opportunity to work under his supervision within his workgroup. All that I have learned and achieved during my PhD became possible with his support. He has been very cooperative during ups and downs of my PhD academics. It was his support that enabled me to work on my PhD topic, which was and is too ambitious as a PhD project for a single person with limited time. He gave me full flexibility to bring my full potential into use. He was always available for suggestions and guidance overcoming his busy and tough schedule of work.

I am thankful to Dr. Marko Rosenmüller, who gave me guidance and support during the earlier years of my PhD. He helped me out during the admission process for my PhD. He supported me in quickly adjusting with my new workgroup. I have worked with him during an early stage of my PhD, and I have learned many important things from him.

I am thankful to Dr. Christian Kästner. He became my third mentor during my PhD. It was his guidance that brought me out of the mess that I was in, towards a focused research topic in the most professional manner. His mentoring has a huge positive impact on my research, both in terms of conducting useful research and presenting research results in the best understandable manner. His efforts and guidance have a big contribution in the completion of this PhD research.

I am thankful to Dr. Eike Schallehn. His support played a pivotal role for me to complete this PhD research. As a senior colleague and as an expert in the database domain, he directed me well to stimulate my research motivation and to focus on finishing within my time constraints through finalizing the contributions. The technical quality of my work has a high impact from his suggestions.

I am thankful to Azeem Lodhi, Ateeq Lodhi, Dr. Sagar Sunkle, and Ingolf Geist. They were with me during the complete period of my PhD as good and supportive colleagues. Ingolf Geist also provided me the important last review on my thesis for final corrections. I am also thankful to Andreas Lübcke, Dr. Sven Apel, Norbert Siegmund, Dr. Nasreddine Aoumeur, Michael Soffner, and all other colleagues for their support.

I will also like to acknowledge Higher Education Commission of Pakistan (HEC), National Engineering and Scientific Commission of Pakistan (NESCOM), Deutscher Akademischer Austausch Dienst (DAAD), and Database Research Group of University of Magdeburg for providing the funds and facilities during my PhD.

List of author publications that contributed to this dissertation

- Syed Saif ur Rahman, Eike Schallehn, and Gunter Saake, ECOS: Evolutionary Column-Oriented Storage, In Proceedings of the 28th British National Conference on Databases (BNCOD 2011), pages 18-32, Springer-Verlag, The University of Manchester, United Kingdom, 12 - 14 July 2011.

- Syed Saif ur Rahman, Eike Schallehn, and Gunter Saake, ECOS: Evolutionary Column-Oriented Storage, Technical Report No. FIN-03-2011, Department of Technical and Business Information Systems, Faculty of Computer Science, University of Magdeburg, 2011.

- Syed Saif ur Rahman, Using Evolving Storage Structures for Data Storage, In Proceedings of the International Conference on Frontiers of Information Technology (FIT 2010), pages 3:1-3:6, ACM, Islamabad, Pakistan, 21 - 23 December 2010.

- Syed Saif ur Rahman, Veit Köppen, and Gunter Saake, Cellular DBMS: An Attempt Towards Biologically-Inspired Data Management, Journal of Digital Information Management, Volume 8 Issue 2, ISSN: 0972-7272, April, 2010.

- Syed Saif ur Rahman, Marko Rosenmüller, Norbert Siegmund, Gunter Saake, and Sven Apel, Specialized Embedded DBMS: Cell Based Approach, In Proceedings of the 20th International Workshop on Database and Expert Systems Application (DEXA 2009), pages 9-13, IEEE Computer Society, Linz, Upper Austria, Austria, 31 August - 4 September 2009.

- Syed Saif ur Rahman and Gunter Saake, Cellular DBMS - An Attempt Towards Biologically-Inspired Data Management, Technical Report No. FIN-012-2009, Department of Technical and Business Information Systems, Faculty of Computer Science, University of Magdeburg, 2009.

- Syed Saif ur Rahman, Azeem Lodhi, and Gunter Saake, Cellular DBMS - Architecture for Biologically-Inspired Customizable Autonomous DBMS, In Proceedings of the 2009 First International Conference on 'Networked Digital Technologies' (NDT2009), pages 310-315, IEEE Computer Society, Ostrava, The Czech Republic, 29 - 31 July 2009.

- Syed Saif ur Rahman, Cellular DBMS (Abstract), In Dagstuhl Seminar Proceedings 08281, 08281 Abstracts Collection - Software Engineering for Tailor-made Data Management, 06.07.2008-11.07.2008, `http://drops.dagstuhl.de/opus/volltexte/2008/1579`.

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

AOP	Aspect-oriented Programming
API	Application Programming Interface
DBMS	Database Management System
DMDSM	Dictionary based Minimal Decomposed Storage Model
DSM	Standard 2-copy Decomposed Storage Model
ECOS	Evolutionary Column-oriented Storage
FOP	Feature-oriented Programming
HLC	High-level Composite
HLC B+Tree	High-level Composite using B+Tree
HLC SL	High-level Composite using Sorted List
HOD	Higher Order Derivative
KDSM	Key-copy Decomposed Storage Model
LOC	Lines Of Code
MDSM	Minimal Decomposed Storage Model
NSM	N-Ary Storage Model
RISC	Reduced Instruction Set Computer
SPL	Software Product Line
VDMDSM	Vectorized Dictionary based Minimal Decomposed Storage Model

1 Introduction

> We can't solve problems by using the same kind of thinking we used when we created them.
>
> Albert Einstein

The database management system (DBMS) is one of the backbone technologies for the information technology industry. The relational DBMS technology itself is not new, rather it dates back to 70's when the emergence of System R [Astrahan et al., 1976] and Ingres [Stonebraker et al., 1976] started a new era for data management. The architecture for System R and INGRES was designed considering requirements for the transaction processing systems and the legacy hardware. The legacy hardware of that era was mainly mainframe systems. Maximum processing power available in 70's and 80's was low relative to hardware capacity of current era. Main memories and hard disks were expensive and scarce resources. The tape was considered as the most economical storage medium for high capacity data storage. Taking into account the price ratio of processor, memory, and disk access of that time, Gray and Putzolu [1987] in 80's gave the five minute rule, i.e., "Data referenced every five minutes should be memory resident" and the ten bytes rule, i.e., "Spend 10 bytes of main memory to save 1 instruction per second". Existing commercial DBMS, such as DB2 and Oracle do find their ancestry relation in some form to earlier DBMS, such as System R.

Over the last four decades, we came across a tremendous change in the hardware and the software technology and its usage. The hardware has grown powerful. Now we have abundance of processing power. Main memory and hard disk densities have exploded. The tape is considered dead, whereas we have another effective medium of flash memory to change the traditional memory hierarchy [Graefe, 2008]. The effect of change in the hardware on the data management during next two decades with the reference to the earlier work of Gray and Putzolu [1987], is properly documented by Gray and Graefe [1997] and later by Graefe [2008] at the interval of ten years. Database researchers acknowledge that the impact of change in the hardware on

the data management is high [Agrawal et al., 2009] and they stress on the need of revisiting existing DBMS architecture for better utilization of new hardware features.

One major impact of improved hardware resources in conjunction with the advent of the Internet is the explosion in data sizes, which setup the need for more sophisticated DBMS to handle large data volumes. Over the last four decades, DBMS vendors have equipped their DBMS with more and more functionalities to fulfill the market requirements. However, the wish list for a DBMS is too long. It wants a DBMS to be scalable, speedy, stable, secure, small, simple, self-managing, self-adapting, self-organizing, self-tuning, and this list goes on and on [Manegold et al., 2009]. Existing DBMS now have grown complex with a multitude of functionalities covering hundreds of affecting parameters [Kersten, 2008]. Database researchers blame monolithic architecture of existing DBMS for complexity and call for a revisit in the existing database architecture [Agrawal et al., 2009; Chaudhuri and Weikum, 2000; Kersten et al., 2003].

The complexity of existing DBMS makes them difficult to manage. They are difficult to tune for consistent performance because of the existence of many interdependent functionalities, which makes them highly unpredictable. It requires highly skilled administrators to manage a consistently functioning DBMS. However, highly skilled human resources are expensive. They make the major cost factor for the data management because of the decrease in the hardware cost. Database researchers have proposed and explored autonomic data management approaches to reduce the total cost of ownership for the data management [Chaudhuri and Narasayya, 2007; Weikum et al., 2002]. Autonomic data management includes many self-* functionalities, such as self-tuning, self-managing, self-organizing, self-healing, self-configuring, self-protection, and self-optimization, but this list could be easily extended with more wishes.

We revisit existing database architecture towards a self-tuning RISC-style database architecture according to the work from Chaudhuri and Weikum [2000]. They proposed the RISC-style database architecture where a database is constructed from small RISC-style components with specialized API, limited interaction among components, and built-in self-tuning capabilities. The goal of their design is to make a database system more predictable and easy to self-tune. Revisiting an existing database architecture and to challenge the prevalent assumption on how a DBMS is constructed is a difficult task [Kersten, 2008]. It requires system-oriented research, which is an ambitious task for a PhD research [Chaudhuri and Weikum, 2000]. For example, it took MonetDB seven years to reach a mature system with many excep-

tionally talented resources [Kersten, 2008]. However, our motivation to undertake this work can be best expressed by the words from Kersten [2008]:

> *Taking a side-road is risky, takes quite some years to mature, but also opens vistas of new architectural adventures.*

1.1 Contributions

1. We present *the Cellular DBMS architecture*, a RISC-style database architecture for customizable and autonomous DBMS development that we designed according to suggestions from Chaudhuri and Weikum [2000]. We show that an instance of a customizable embedded database can be used as a RISC-style data manager, which we termed as *cell*. We use the software product line approach to achieve the customization of each cell in a Cellular DBMS by extending the contribution of the FAME-DBMS project[1] [Leich et al., 2005; Rosenmüller et al., 2008]. We also propose structures and mechanisms for autonomic behavior in a database architecture focusing specifically on the self-tuning. We use the aspect-oriented programming to realize autonomy in a DBMS. We also provide details about the realization of a DBMS using the proposed architecture.

2. We present a customizable and self-tuning storage manager realized according to the Cellular DBMS architecture. The storage manager use the decomposed storage model (DSM) [Copeland and Khoshafian, 1985] and its four proposed variations for storage model customization at the table-level. The storage manager also use *hierarchically-organized storage structures*, *evolution*, and *evolution paths* at the column-level to achieve both customization and autonomy. The proposed hierarchically-organized storage structures can be optimized according to a hardware hierarchy. We present a mechanism to increase and decrease the hierarchy of hierarchically-organized storage structures using different storage structures that we dynamically select according to the workload. We show an evolution mechanism that transforms a storage structure from one form to another autonomically. We introduce the mechanism of evolution paths, which defines how hierarchically-organized storage structures evolve.

[1] "Fame-DBMS project", `http://fame-dbms.org/`, Accessed: 21-06-2011

3. We document our experience of the Cellular DBMS prototype implementation. We outline problems that we faced and our design decisions to solve those problems. We present in detail our prototype implementation. We use our prototype implementation statistics to present the effect of an increase in functionalities on the increase in the number of tuning knobs, LOC, and binary size. We also present the effect of an increase in the number of features on the increase in the number of feature derivatives. We explain our evolution mechanism implementation using the aspect-oriented programming by presenting the code snippets.

4. We provide the evaluation results for the following:

   a) The impact of unused functionalities on the performance of a database

   b) The change in the performance of a storage structure with the growth in the database size

   c) The performance gain from the use of evolving hierarchically-organized storage structures

   d) The performance difference of different DSM schemes

   We also present the evolution behavior of evolving hierarchically-organized storage structures. We used Berkeley DB to assess the impact of unused functionalities, whereas other evaluations are performed on the Cellular DBMS prototype. We used a custom micro benchmark for our evaluation.

1.2 Outline

In this section, we provide the outlook of the thesis structure. We distributed the related work content among chapters according to their relevance taking into account the fact that the required related work in this thesis comes from multiple domains. Similarly, we also distributed the motivation for each chapter among them. The outline of the thesis content is as follows:

In **Chapter 2 (Background)**, we introduce the reader with basic terms and concepts that are mandatory for the understanding of the thesis. This chapter introduces concepts from both the data management and the software engineering domains.

In **Chapter 3 (The Cellular DBMS architecture)**, we motivate the reader regarding the need for customization and autonomy support in a database archi-

tecture. This chapter introduces our Cellular DBMS architecture. It outlines the design principles for customization and autonomy in the architecture and provides a detailed explanation from the software engineering perspective about, how these design principles can be realized using the innovative software engineering techniques.

In **Chapter 4 (A customizable and self-tuning storage manager)**, we present a customizable and self-tuning storage manager that we designed and implemented according to the Cellular DBMS architecture. We provide a separate motivation for our design decisions specific to the storage manager implementation. We explain how the relational model can be realized in the Cellular DBMS architecture. We introduce four variations for the DSM scheme to be used for storage model customization at the table-level. We introduce hierarchically-organized storage structures. We present concepts of the evolution and the evolution path that enables a storage manager to self-tune itself with reduced human intervention. We also provide the theoretical explanation for evolving hierarchically-organized storage structures that we used in our storage manager.

In **Chapter 5 (The prototype implementation: Problems faced and lessons learned)**, we document our experience with the Cellular DBMS prototype implementation. It discusses the details about the problems that we faced and our different design decisions to solve those problems. It presents the Cellular DBMS prototype implementation with detailed discussion from the software engineering perspective. It also provides the insight about the source code implementation of the evaluation mechanism in the prototype.

In **Chapter 6 (Evaluation)**, we present the evaluation results for our prototype implementation. It explains the micro benchmark that we used for evaluation and provides a detailed discussion on the presented results.

In **Chapter 7 (Concluding remarks and future work)**, we conclude the thesis by providing the summary of our work and outlining the conclusions that we reached from our experiences and results. We outline the foreseen future work during the research with a detailed explanation on most of them with a focus on their possible solutions.

2 Background

> Completion of the database jigsaw puzzle calls for organizations of its pieces, trying out combinations, and assembling bits and pieces. Staring at the same piece over and over again will not lead to the satisfaction of understanding the complete picture. Our community calls for adventurous academics who explore unknown territory and mark it with paradigm shifts.
>
> Martin L. Kersten
> *The Database Architecture Jigsaw Puzzle (Keynote Talk), ICDE 2008*

In this thesis, we introduce a unique and novel DBMS architecture, which requires knowledge of concepts from both the data management and the software engineering domains. Therefore, we introduce and explain them with many other background concepts in this chapter.

2.1 DBMS architecture

The architecture of a DBMS defines its structure in terms of components, the behavior of components, and the relationship and interaction among them [Özsu and Valduriez, 1999]. Existing DBMS are designed and developed according to the layered DBMS architecture. The layered DBMS architecture stems from earlier work of the Data Independent Accessing Model (DIAM) presented by Senko et al. [1973]. DIAM defines an information system using multiple self-sufficient abstractions, such that each higher level is more abstract than the lower level and provides a simpler environment for solving information system design problems. DIAM defines four layers of abstraction namely entity set model, string model, encoding model, and physical device model. Later on Härder and Reuter [1983a; 1983b] presented a five layer DBMS architecture constructing further on the DIAM. Their five proposed layers are namely file management, propagation control, access path management,

navigational access, and nonprocedural access. The design theme for the layered DBMS architectures is the data independence to overcome the change with changing data management requirements.

2.2 Embedded database

An embedded database is a database management solution that is embedded/tightly integrated into its user-application. This term is also used for data management software for embedded systems [Olson, 2000]. Embedded databases are intended to operate in a management-less environment hidden from the end-user. Two popular open-source embedded databases are Berkeley DB [Olson et al., 1999; Oracle Berkeley DB] and SQLite [SQLite]. MySQL also provides the embedded version of their DBMS servers, which they call the embedded MySQL library [MySQL Database].

2.3 Storage models

The storage model selection is an important design decision for a DBMS architecture. Two most commonly used storage models are *N-Ary Storage Model (NSM)* and *Decomposed Storage Model (DSM)*. The NSM stores data as seen in the relational conceptual schema, i.e., all attributes of a conceptual schema record are stored together [Copeland and Khoshafian, 1985]. Most of popular commercial DBMS, such as DB2, Oracle, MS SQL Server, and MySQL use the NSM. The DSM is a transposed storage model [Batory, 1979] that store all values of the same attribute of the relational conceptual schema relation together [Copeland and Khoshafian, 1985]. Svensson [2008] mentioned the Cantor project [Karasalo and Svensson, 1983, 1986] as the pioneer for this approach. In literature column-oriented [Stonebraker et al., 2005], vertical fragmentation [de Vries et al., 2001], and vertical partitioning [Abadi et al., 2007] are terms used to refer to solutions similar to the DSM.

Copeland and Khoshafian [1985] analyzed both approaches and concluded that neither of the two approaches could be an ideal solution for all domains. The DSM requires relatively more storage space, however, the required storage can be reduced by using compression techniques [Holloway and DeWitt, 2008]. Update and retrieval performance of both models depends on the nature of data and implementation of models. The DSM is known for fast retrieval whereas the NSM is efficient in fast updates [Holloway and DeWitt, 2008]. Copeland and Khoshafian [1985] suggest that many disadvantages of DSM can be avoided by using hardware and software

techniques, such as differential files, multiple disks, and large main-memory. The DSM allows using the CPU cache efficiently [Zukowski et al., 2005]. Zukowski et al. [2008] compared the two approaches on the most recent hardware to assess the CPU performance trade-offs in the block-oriented query processing. Zukowski et al. concluded that it depends on a query to identify, which data layout is better. Furthermore, they recommended on-the-fly conversion between these formats for better performance and stressed for further research on a hybrid data layout using the best of both approaches. Example of hybrid data layout can be found in PAX [Ailamaki et al., 2002], fractured mirrors [Ramamurthy et al., 2002], and MonetDB/X100 [Zukowski et al., 2005].

2.4 Autonomy and self-tuning

By autonomy, we mean the capability of a DBMS to monitor, diagnose, and adjust itself. It is a generic term that covers all functionalities of a DBMS that are required to automatically manage, maintain, tune, or heal a DBMS. In contrast, self-tuning is the specialization of autonomy. By self-tuning, we mean the automation of DBMS tuning activities performed by a DBMS administrator. Shasha and Bonnet [2003] defines tuning as:

> "Database tuning is the activity of making a database application run more quickly. "More quickly" usually means higher throughput, though it may mean lower response time for some applications. To make a system run more quickly, the database tuner may have to change the way applications are constructed, the data structures and parameters of a database system, the configuration of the operating system, or the hardware."

For self-tuning, a DBMS monitors itself for performance tuning related parameters, diagnoses the causes for identified performance degradation, and performs the tuning activities to maintain the required performance (or if possible, performs preventive actions to avoid similar performance degradation in the future). Self-tuning can be performed statically as well as online. Static self-tuning means the self-tuning that requires manual initiation of tuning process. With static self-tuning approaches, the self-tuning advisors provide the database administrator with the advices to tune the DBMS. Then it waits for the administrator to select among the recommended activities for tuning. In contrast, the online self-tuning means the self-tuning that is

performed continuously and automatically [Bruno and Chaudhuri, 2007b]. Online self-tuning approaches are normally tightly integrated with the DBMS functionality, and it requires minimal human intervention for adjusting the DBMS parameters.

2.5 NoSQL databases

NoSQL database is the terminology used for API-based non-relational databases. They are gaining attraction because of their capability to handle unstructured data efficiently in comparison with relational databases. They are also known for taking benefit from the distributed processing using the commodity hardware. Dynamo [DeCandia et al., 2007] and Big Table [Chang et al., 2008] are the two examples for NoSQL databases. They both possess key/value interfaces for data storage.

2.6 Reduced Instruction Set Computer (RISC) and a RISC-style database system

Chaudhuri and Weikum [2000] used the term RISC in their proposal for different self-tuning RISC-style database system architecture. Their presented concepts are inspired from RISC-based central processing unit (CPU) design, which advocates the construction of a CPU using simpler and faster instructions instead of complex and slow instructions [Patterson and Ditzel, 1980]. Use of complex instruction for CPU design has its own benefits, which includes upward compatibility and better marketing opportunities. We believe similar benefits as a factor that derived existing DBMS components towards prevailing complexity. However, Patterson and Ditzel [1980] discussed both approaches in the context of CPU design and presented the benefits of RISC-based design, such as simple and fast instruction, a possibility of careful pruning of an instruction set, and minimizing complexity to maximize performance. We intend to achieve similar benefits in our RISC-style Cellular DBMS architecture.

Chaudhuri and Weikum [2000] suggested the use of RISC-style data managers with narrow functionality, specialized API, small footprint, and limited interaction. Their aim is to reduce the number of tuning knobs for a DBMS to make it more predictable in terms of performance and behavior making it easy to self-tune. They defended their proposal with the notion of "gain/pain" ratio, which suggests to tolerate a moderate degradation in "gain" with the introduction of overheads present in their

approach to reduce the "pain" related to tuning with more predictable performance.

2.7 Related software engineering concepts

Software engineering concepts of the software product line, the feature-oriented programming, and the aspect-oriented programming are relatively unfamiliar in the database domain. Therefore, we introduce and explain them in this section.

2.7.1 Software product line

Software product line (SPL) engineering is an approach to generate related and similar software products using the same code-base [Pohl et al., 2005]. It is inspired from the concept of product line used in industry for production of related and similar products. SPL uses the term of variant to define the different products that are related and similar in some fashion [Kang et al., 1990]. It uses the term of feature to precisely identify the differences and similarities between variants [Zave, 2003]. The use of SPL for developing tailor-made data management solutions have been successfully demonstrated by many researchers. Rosenmüller et al. [2009a] presented the use of SPL for developing tailor-made data management solutions for embedded domain. Saake et al. [2009] presented the benefit of using SPL from the perspective of downsizing the existing data management solutions for embedded systems.

2.7.2 Feature-oriented programming

An SPL can be realized using different software development techniques. However, we confine our discussion with the *feature-oriented programming (FOP)*, which we used to realize our DBMS SPL. FOP is a mechanism to develop a software product line, where software products are manufactured by composing features [Batory et al., 2003]. FOP is formulated for construction of customizable large-scale software systems. In FOP, a feature is a functional unit of a software system [Batory et al., 2003]. It satisfies a user requirement and at the same time provides us with the configuration option to achieve customization. Many researchers have presented the benefit of using FOP for realizing a DBMS SPL. Leich et al. [2005] presented the design and implementation of database storage manager family for resource constrained and heterogeneous embedded system scenario. They showed that high degree of flexibility makes FOP an appropriate mechanism to realize tailor-made data management solutions. As mentioned in Section 2.7.1, both Rosenmüller et al. [2009a]

and Saake et al. [2009] realized SPL using FOP. They showed that fine-grained customization of FOP provides a better solution for tailoring data management for resource constrained systems.

2.7.3 Aspect-oriented programming

Aspect-oriented programming (AOP) [Kiczales et al., 1997] is a methodology that emerged with an aim to separate cross-cutting concerns from core concerns of a source code. By concern, we mean a small manageable piece of source code that is a semantically coherent and identifiable functionality. By cross-cutting concern, we mean a functionality that is scattered and tangled with other functionalities all over the source code. AOP ensures code scalability and maintenance by preventing code tangling and scattering [Kiczales et al., 1997]. Using AOP, a cross-cutting concern is separated from the core source code using a construct of an aspect. An aspect is a modular way to separate the concern code that otherwise is part of different software components. Aspects, such as data persistence, transaction management, and data security, etc., can either be provided by a software component or could be required by it [Kiczales et al., 1997]. Using concepts, such as join-points, pointcuts, and advice; an aspect weaver component brings the program code and aspect code together [Kiczales et al., 2001]. The process of joining the program code and aspect code together is called aspect weaving. Join-points are points in the execution of a program and are events of interest for aspect weaving [Kiczales et al., 2001]. Pointcuts is the collection of join-points and is used for selection of related method-execution points [Kiczales et al., 2001]. An advice is the intended behavior to be weaved [Kiczales et al., 2001]. With all above-mentioned benefits, AOP also has few shortcomings, such as lack of tools support, existing tools are not mature, and difficulty to debug.

2.7.4 Customization

Customization is a mechanism to tailor the software according to the end-user requirements. For a database, customization means the tailoring of a database according to the data management requirements, which requires precise selection of required functionality. A customized database instance constitutes of only the selected functionality, and it is intended for the use of specialized data management.

3 The Cellular DBMS architecture

This chapter shares material with the DEXA'09 paper "Specialized Embedded DBMS: Cell Based Approach" [ur Rahman et al., 2009b], the NDT'09 paper "Cellular DBMS - Architecture for Biologically-Inspired Customizable Autonomous DBMS" [ur Rahman et al., 2009a], and the JDIM'10 paper "Cellular DBMS: An Attempt Towards Biologically-Inspired Data Management" [ur Rahman et al., 2010].

This chapter lays the conceptual foundation for this thesis by introducing the Cellular DBMS architecture from both data management as well as software engineering perspective. It outlines the motivation for the need of customization and autonomy in a DBMS architecture. It presents the design principles for the Cellular DBMS architecture. It explains how the software product line approach is used to achieve customization in it. It also explains how the aspect-oriented programming can be used to implement autonomy functionality in a DBMS.

3.1 Motivation for the customization in an architecture

Existing data management systems are complex [Chaudhuri and Weikum, 2000; Harizopoulos and Ailamaki, 2003]. There are two major reasons behind their complexity. First, the data management systems were developed decades ago. They were developed for legacy hardware and applications. Secondly, over the time the hardware kept changing and the data management needs were also changed. To overcome the changing data management and hardware requirements existing data management solutions were forced to evolve. Functionalities were added to the existing engines over time. Because of monolithic architectures of these engines, each new functionality got tightly coupled with an engine and the inter-dependencies among functionalities made it difficult to later remove the unused functionalities [Chaudhuri and Weikum, 2000; Härder, 2005; Harizopoulos and Ailamaki, 2003]. Furthermore,

functionalities were not only added to satisfy the emerging data management requirements, instead many of them were added to take an edge in the data management market. It resulted in data management solutions, which are full of functionalities; however, many of these functionalities are not required for most of the application scenarios. Moreover, existing DBMS vendors provide their customers a bundled package of their DBMS product. These bundled packages contain many features that customer might never use in their application scenarios. The price of these bundled packages contains the cost of these unnecessary functionalities resulting in high total cost.

All above-mentioned reasons motivated us and many other database researchers to revisit the existing DBMS architecture towards more diversified architecture [Agrawal et al., 2009]. The revisited architecture should be able to reduce the complexity through the capability of removing the unused functionalities. We use the term *customization* to refer to this capability in this thesis. Customization requires loose coupling among functionalities, which could only be possible with reduction in dependencies among them. We revisit existing DBMS architecture to support better customization. Our work is motivated by the proposal of Chaudhuri and Weikum [2000] for rethinking existing database system architecture towards a self-tuning RISC-style database system architecture.

3.2 Motivation for the autonomy in an architecture

In the previous section, we discussed the complexity of existing DBMS, high number of functionalities in them, and tight interdependencies among those functionalities. All above-mentioned reasons contribute to their unpredictable performance [Chaudhuri and Weikum, 2000]. Existing DBMS require continuous administration and tuning to achieve the consistent performance over the time. These administration and tuning tasks become more difficult with the change in data management requirements, workloads, hardware and software platforms, and many other influential parameters. Furthermore, the tuning tasks become more difficult because of the fact that it is difficult to assess the effect of tuning of one knob on another in existing DBMS [Weikum et al., 2002]. Because of this reason, traditional tuning of a DBMS for consistent performance is rather a process of trial and error, instead of a systematic procedure.

Existing DBMS require human resources for administration and maintenance. Experienced technical human resources are expensive. The human resource cost has

Table 3.1: TPC-H LINEITEM table observed statistics, possible customization, and anticipated evolution.

Column Name	Distinct Count	Workload	Data Access	Storage Structure Initial	Storage Structure 1st Evolution	Storage Structure 2nd Evolution
L_ORDERKEY	1500000			Sorted Array	Sorted List	B+-Tree
L_COMMENT	4501941			Sorted Array	Sorted List	Hash Table
L_DISCOUNT	11	Read-Intensive		Sorted Array		
L_SHIPMODE	7			Heap Array		
L_SHIPINSTRUCT	4			Heap Array		
L_RECEIPTDATE	2554			Heap Array	Heap List	
L_COMMITDATE	2466		Ordered	Sorted Array	Sorted List	
L_SHIPDATE	2526		Ordered	Sorted Array	Sorted List	
L_LINESTATUS	2			Heap Array		
L_RETURNFLAG	3			Heap Array		
L_TAX	9	Read-Intensive		Sorted Array		
L_EXTENDEDPRICE	933900	Read-Intensive		Sorted Array	Sorted List	B+-Tree
L_QUANTITY	50	Read-Intensive	Ordered	Sorted Array		
L_LINENUMBER	7			Heap Array		
L_SUPPKEY	10000			Heap Array	Heap List	
L_PARTKEY	200000			Sorted Array	Sorted List	Hash Table

become the major contributor to the total cost of ownership for data management, because of the decrease in hardware and software cost [Weikum et al., 2002]. This motivates the need to reduce the required human intervention for DBMS administration and maintenance. To reduce the human intervention, we have to simplify the DBMS management and tuning tasks. The first suggestion that came as the solution is the possible reduction in the number of tuning knobs [Chaudhuri and Narasayya, 2007; Chaudhuri and Weikum, 2000; Weikum et al., 2002]. Self-tuning is the second solution to reduce human intervention through automating as many tuning tasks as possible.

Here onwards, we make use of the Transaction processing Performance Council benchmark H (TPC-H) schema [TPC-H] for our discussion as needed. Consider the distinct data count of two large columns, i.e., L_ORDERKEY and L_COMMENT in Table 3.1 for the LINEITEM table of the TPC-H schema. For the benchmark scenario, we generate the data altogether to test our data management solutions, and we customize the storage structure to best suit our desired results. However, in

a real world scenario, the data growth is a continuous process. Database designer can predict, how large data can grow and at what rate, but he/she should maintain the database over time. We can elaborate the problem with two possible scenarios. For example, in a first scenario, we suggest a B+-Tree as a suitable storage structure (assume data stored with index) for the L_ORDERKEY column, but what if only after 30 years the expected maximum data size is reached? During the first year, a sorted list could have been good enough to store the data. When we select a complex storage structure for small database management, for each data management operation, we waste resources (cache, memory, and CPU cycles) until and unless data size grows to make the use of the selected storage structure appropriate [ur Rahman, 2010]. For the contrary second possible scenario, a database designer selects a sorted list as a storage structure. However, the data growth is much higher than expected. In a year, the sorted list will become inadequate for the desired performance. The database will need maintenance, which includes changing the storage structure by human intervention.

Another important issue is the change in the workload patterns. It is possible that a workload that was previously populated with write-intensive queries, later in the lifetime of the database becomes more read-intensive. A classical approach would require a manual analysis of the queries. Then according to the results it might require changing the configuration parameters or managing the database structures, such as creating an index, dropping a materialized view, or partitioning a table. All these tasks require human intervention, which is expensive. Therefore, we suggest that an autonomic approach for adjusting data management solutions with changing data management needs is required.

3.3 The Cellular DBMS architecture and the Cell

The Cellular DBMS architecture is based on the RISC-style database system architecture proposed by Chaudhuri and Weikum [2000]. The *Cellular DBMS architecture* proposes to construct a large DBMS by using multiple cells. The *Cell* in the Cellular DBMS architecture is an instance of a small, simple, and customized database. Here onwards, we use the term of cell to represent the smallest possible data management unit of our architecture. By small, we mean that a cell contains few limited functionalities. For example, a database with only in-memory data management capability using an array for the data storage of 4 KB of data. By simple, we mean that the cell exposes a narrow and consistent interface. For example, a

database with simple Put(), Get(), and Delete() function interface for data management operation. Two constraints of making cells small and simple are according to the proposal of RISC-style database system architecture. By *customized*, we mean that each cell is tailored according to its data management needs. For example, for read-optimized data management, a database contains only the data structures that store data in sorted order, whereas for a write-optimized data management, a database contains only the data structures that store data in the insertion order, such as a heap. When we say *composing a cell*, this means precisely selecting the required functionalities and pruning the not-required functionalities from the list of all possible functionalities that a cell can possess. Cell composition is performed at the compile time, i.e., it is static, which means we cannot add or remove the functionalities from the cell at runtime. In the Cellular DBMS architecture, each cell is an atomic unit of data management functionality, i.e., each cell is capable of performing its data management operations independently. By *composing multiple cells*, we mean the composition of all cells in concert to achieve the required data management capability of a complete DBMS.

In the Cellular DBMS architecture, each cell stores *key/value pairs* of data. This design decision enables us to generate a cell of a limited functionality and simple interface, which results in a cell with more predictable performance. By *predictable performance*, we mean the capability of a cell to execute each data management operation in predictable time and with predictable resources. Furthermore, the performance of a cell should also be predictable with the change in data size, i.e., with the growth of data, we should be able to precisely identify the change in required time and resources to complete a data management operation. Moreover, customization and key/value pair storage also reduces the complexity of a cell. The complexity of a cell is dependent on the number of functionalities it contains and the interdependencies among those functionalities. We suggest the reduction in complexity because each cell contains only the required functionalities, which are further simplified with the requirement of storing only key/value pairs of data. However, the overall complexity of a Cellular DBMS is expected to increase with many differently composed cells performing the data management tasks in concert. Here, we have discussed how each cell complexity is reduced with reduction in cell functionalities, and we continue our discussion in this chapter with an assumption that there exists a mechanism to manage and hide the overall Cellular DBMS complexity. In Chapter 4, we explain in detail our mechanism to realize a more complex relational model using these simple key/value pair stores.

3.4 Autonomy in the Cellular DBMS architecture

Autonomy of each cell is an important design principle for the Cellular DBMS architecture. The Cellular DBMS architecture envisions the development of a complete autonomous DBMS by accumulating autonomic behavior of all participating cells. For autonomy, the most fundamental functionalities are monitoring, diagnostics, and tuning [Chaudhuri and Narasayya, 2007; Lightstone et al., 2002]. Autonomy requires runtime transformations. These transformations could be behavioral as well as structural. For an architecture to be autonomous, it should support both autonomic structures and behaviors. In the Cellular DBMS architecture, we introduce different compositions of cells as autonomic structures to enable execution of autonomic behavior. The Cellular DBMS architecture proposes that each cell should be an autonomous data management unit. It should be able to monitor itself. If an abnormal behavior is observed, it should be able to diagnose the cause of the behavior. Finally, it should be able to transform itself structurally or behaviorally or in both forms to achieve and maintain the required normal behavior.

According to our definition of a cell in the previous section, each cell is an instance of a small, simple, and customized database. Here, we extend the cell definition as: each cell is an instance of a small, simple, customized, and *autonomous* database. According to the proposed architecture, monitoring, diagnostic, and tuning components should also be customizable according to the cell functionalities to ensure reduced monitoring overhead, however; it is part of our future work.

According to our discussion in previous section, we customized each cell to few minimal functionalities. By minimal functionality, we also mean to constrain the storage capability of each cell. We suggest that this limitation enables us to execute data management operations in predictable time and with predictable resources. However, for large data storage, we propose to induce more cells with data growth. The Cellular DBMS architecture introduces different compositions of cells that are as follows:

Composite cell A cell can be composed of multiple similar or dissimilar cells related to each other as shown in Figure 3.1. Such composition of cells is termed as the *composite cell*. Each composite cell itself has limited (optimal) data-handling capacity to ensure that it has manageable complexity and predictable performance. With the data growth, more cells could be inducted into the DBMS to extend its data management capacity. Each composite cell maintains a meta-data of cell composition. By meta-data of cell composition, we mean the information related to cell

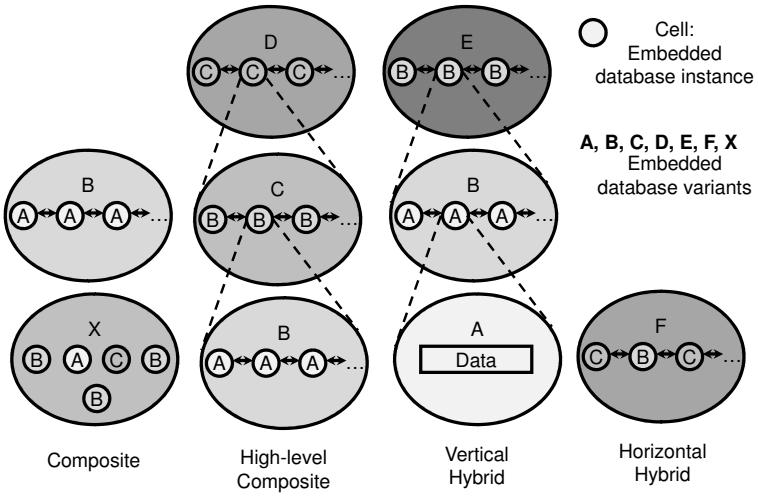

Figure 3.1: Different types of cell.

organization and relationship within a composite cell. This includes configuration information, such as maximum cells limit and references to buffer managers. It also includes statistics, such as cells count, records count, first record, and last record. A composite cell can be used to realize a table structure for the relational model in a Cellular DBMS, where each column is realized by a cell that could be of different type, e.g., one column cell contains un-indexed data management functionality, whereas another column cell uses indexes. We explain in detail our mechanism to realize the relational model in Chapter 4.

High-level composite cell (HLC) In the Cellular DBMS architecture, we propose to build composite cells from simple cells, as well as from composite cells, which results in the *high-level composite cell (HLC)* as shown in Figure 3.1. HLC is initialized as simple cell that transforms to composite cell, which further transforms into HLC. Each transformation increases the hierarchy of HLC and each new level of hierarchy is restricted with definite storage capacity limitation. The Cellular DBMS architecture uses HLC cell for handling large amount of data.

Hybrid cell For diversified data management, the Cellular DBMS architecture introduces the concept of the *Hybrid Cell*. We could have horizontal as well as vertical hybrid cells as shown in Figure 3.1. By horizontal hybrid cell, we mean a composite cell that is composed of different types of cells, such that each type is handling a definite data range. For example, we want to store city codes to be used in the contact book of a mobile phone product. If mobile is to be used in European Union

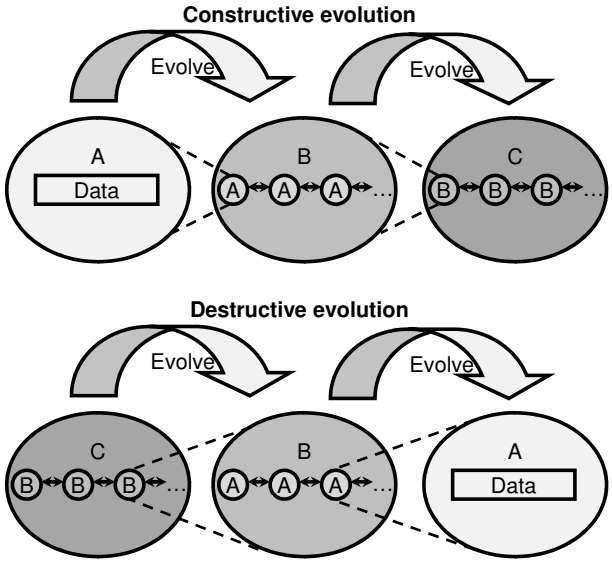

Figure 3.2: Evolving cell.

(EU), frequency to access city codes of EU countries is much higher as compared to city codes of Africa. Using horizontal hybrid cell, we can store data in a composite cell in such a way that EU city codes should be stored in cell with a type that is suitable for faster access time, whereas we store remaining city codes in a cell, which requires less storage space. We can exploit this feature in conjunction with autonomy to move data among different cells based on their usage scenario and available resources.

By vertical hybrid cell, we mean a HLC cell that is composed of different types of cells at different levels of hierarchy. We provide an extensive discussion on vertically hybrid HLC cells in Chapter 4, where it is referred as evolving hierarchically-organized storage structures.

Evolution Evolution in the Cellular DBMS architecture means run-time transformation of a cell from one form into another according to the cell types we defined above. We term a cell that supports evolution as an *evolving cell*. Evolution can be constructive as well as destructive. By constructive evolution, we mean the transformation of a cell from one form into another in such a way that the previous form becomes an atomic integral unit of the new form as shown in Figure 3.2. The new form of such an evolved cell should have a larger data-handling capacity. By destructive evolution, we mean the transformation of a cell from an existing form to a previous form as shown in Figure 3.2.

Distributed cells In the Cellular DBMS architecture, cells are not confined to a single computing resource. Cells can be distributed across a network, or more ambitiously speaking across the Internet. Important distribution criteria could be size and locality of data. Distributed cells interact with each other through API calls over the network. For distributed deployment, we envision a Cellular DBMS using a global data dictionary and statistics as well as distributed monitoring functionality to implement distributed autonomy. However, it has to be further analyzed how distributed deployment of interacting cells can be achieved in a Cellular DBMS. It is a part of our future work.

Cell classification According to the Cellular DBMS architecture, we can also classify cells in two types based on the data they store, i.e., data cell and meta-data cell. A data cell manages data, whereas a meta-data cell stores meta-data.

3.5 Realization of a cell

In the Cellular DBMS architecture, we do not confine our discussion to formal concepts. Instead, we take a step forward to explain the realization of these concepts. In the Cellular DBMS architecture, each cell is realized as an instance of an embedded database. For the Cellular DBMS architecture, we impose three important constraints on the definition of an embedded database (also suggested as characteristics of an embedded database in literature [Olson et al., 1999]), which are as follows:

Small footprint An embedded database should possess a small footprint. By a footprint, we mean a memory space required by a process for execution. Footprint of an embedded database is directly related to the number of functionalities it encloses. Through reducing the functionalities, we are able to shrink the memory required by an embedded database to execute. Reducing the footprint of an embedded database enables us to reduce the overall footprint of a Cellular DBMS. Moreover, it enables us to trace the memory requirement of differently customized embedded databases, which equips us with the capability to assess the need of additional memory with the increase in data size.

Limited set of tasks An embedded database should be able to execute a limited set of tasks. It helps us to limit the functionality of an embedded database, which further

enables us to expose a narrow interface for these functionalities. In accordance with the concept of RISC-style architecture [Chaudhuri and Weikum, 2000; Patterson and Ditzel, 1980], we make use of few, simple, and fast tasks instead of many, complex, and slow tasks. Furthermore, we suggest that it also facilitates us with more predictable execution of these tasks. This design decision is also supported by the results from Harizopoulos et al. [2008], which showed the 20-time performance gain for executing their modified TPC-C benchmark [TPC-C] through pruning the not-needed features from their original transaction processing database system (i.e., Shore [Shore]).

API-based access An embedded database should expose its functionalities using API-based access (i.e., no SQL interface for embedded databases). This design decision empowers us to remove the overheads associated with SQL language parsing, optimization, and execution. We can simply say that for our proposed footprint and task constrained embedded databases, we do not need SQL language and query processing. An API-based mechanism should be sufficient for the data management needs of an embedded database. This design decision is also motivated by the suggestion from Stonebraker et al. [2007], who called SQL language as a "one size fits all" solution. Similarly, Chaudhuri and Weikum [2000] referred to SQL language as painful. They supported their statement with the facts that SQL language provides a high number of features, the complexity of the language is high, and to learn and use most of these features is difficult.

SQL language and query processing also has many benefits that cannot be undermined, what we need is a mechanism to reduce the complexity by careful pruning of not needed features [Rosenmüller et al., 2009b] and increase the usability of the available features [Chaudhuri and Weikum, 2000]. In the Cellular DBMS architecture, we propose to implement the query processing system at the higher-level (i.e., at overall DBMS-level), but as an optional functionality, i.e., it should not be mandatory for a Cellular DBMS to contain a query processor rather a Cellular DBMS could also be a NoSQL database. For the Cellular DBMS architecture, we also envision the need to revisit the query processing system, however, the query processing for the Cellular DBMS architecture is part of our future work and is discussed in detail in Section 7.2.1.

According to our discussion above, we can revise our definition for the architecture, such that:

> The Cellular DBMS architecture proposes to construct a large DBMS by using multiple atomic, customized, and autonomous instances of embedded databases in concert.

3.5.1 Using the software product line to achieve customizability

In our discussion above, we outlined the details of our Cellular DBMS architecture from the perspective of the database domain. In this section, we discuss our architectural details from the software engineering perspective. How the Cellular DBMS architecture utilizes software engineering techniques to realize customizability is the theme of our discussion in this section. We use the software product line (SPL) approach to generate a customizable embedded database, which can be instantiated as a cell. The benefits of using the SPL for tailor-made data management in the embedded domain have already been presented by Leich et al. [2005], Rosenmüller et al. [2009a], and Saake et al. [2009].

The SPL approach allows us to implement an embedded database in such a way that the same code-base can be used to generate different types of embedded databases. By different type of embedded database, we mean that each embedded database differ from another in terms of functionalities it contains. SPL uses the term variant for these types. Here onward, we use the term variant to refer to different type of embedded database. SPL uses the term feature to precisely identify the difference in variants. In our discussion scenario of database domain, heap storage, B+-Tree, transaction management, logging, and similar functionalities are examples of features. Whereas, in-memory embedded database, embedded database with only B+-Tree index, and embedded database with only hash-based index are few examples of embedded database variants. Each of these embedded database variants can be used as cell in a Cellular DBMS. We use the feature-oriented programming (FOP) to implement the SPL of our Cellular DBMS architecture. The FOP is not the only approach to realize an SPL. Component-based programming [Szyperski, 2002], AOP, or simple #ifdef directives could be alternatives to implement an SPL. The details of our prototype implementation are provided in Chapter 5.

In Figure 3.3, we present a sample DBMS SPL. As it can be seen, the DBMS SPL contains all the features that can be selected or removed to generate the software products, which in our case are embedded database variants. A DBMS SPL realizes

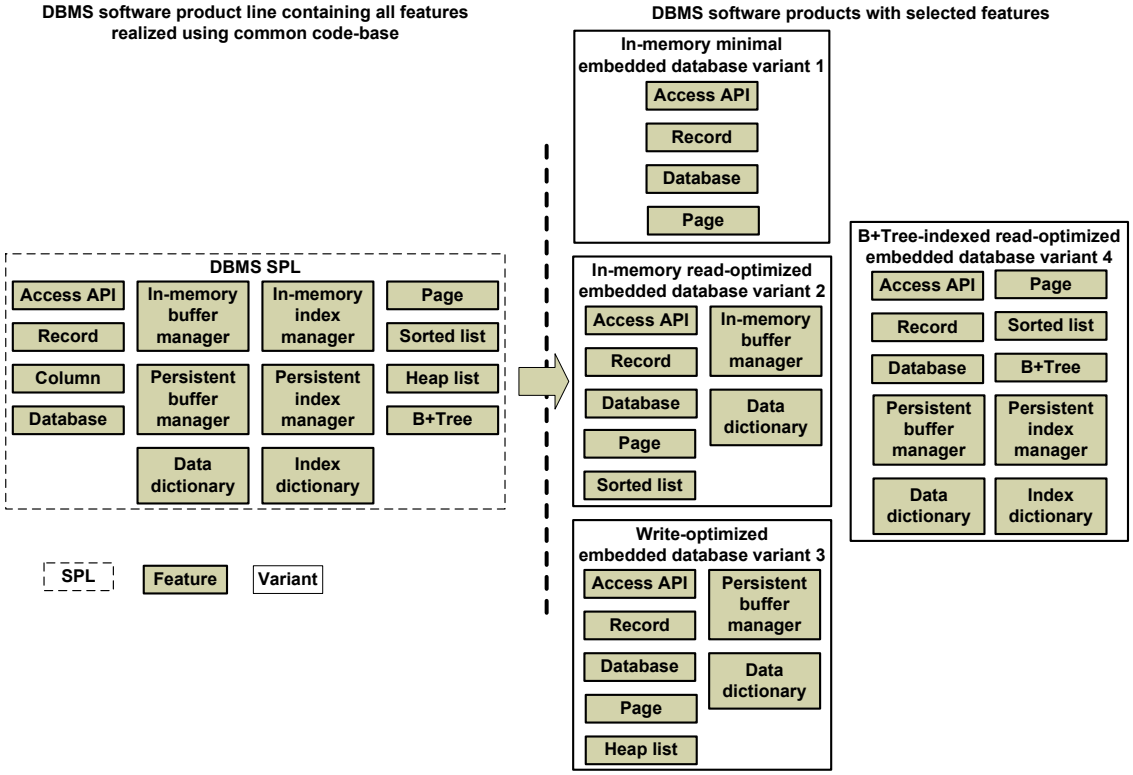

Figure 3.3: Sample DBMS SPL and its few possible variants.

all features using the common code-base. The dashed line in the middle of the figure separates the DBMS SPL from its variants. The arrow in the middle shows the composition process that generates the variants. In our sample scenario of Figure 3.3, we generated four different variants from our DBMS SPL. Each of these four variants differs from each other in terms of features. Each of them is suitable for different data management scenarios. Variant 1 is the minimal possible variant for an embedded database, i.e., it shows that at-minimum every variant should contain four features of access API, record, column, and database. To this point, for our SPL-based design, we make use of the concepts contributed by FAME-DBMS [Rosenmüller et al., 2008] project and its related publications from Leich et al. [2005], Rosenmüller et al. [2009a], and Saake et al. [2009]. We make use of multiple instances of each of these variants as cells in a Cellular DBMS.

In our sample scenario of Figure 3.4, we present the contribution of the Cellular DBMS architecture. It shows, how the Cellular DBMS architecture proposes to make use of multiple instances of embedded database variants generated from a DBMS SPL to construct a large DBMS. In Figure 3.4, we also show a glimpse of, how the Cellular DBMS architecture realizes the relational model. In Figure 3.3,

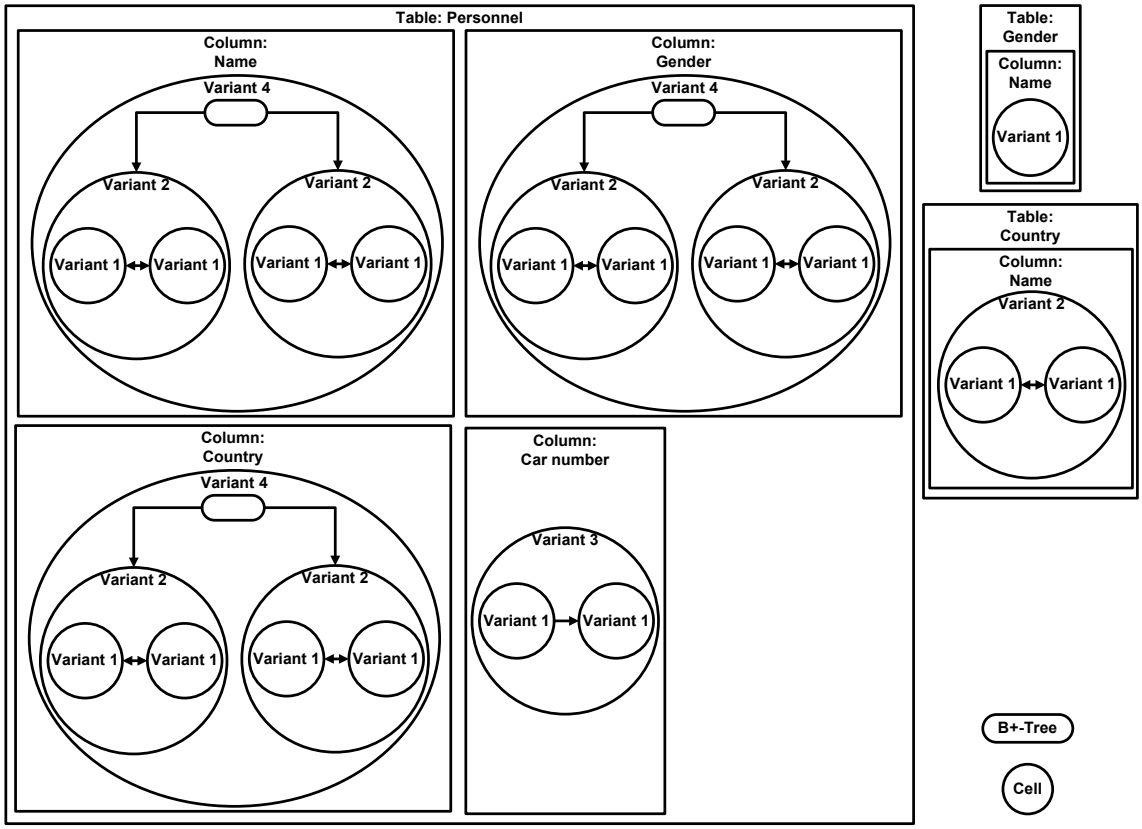

Figure 3.4: Sample Cellular database realization for the relational model using multiple embedded database variants.

we generated four possible embedded database variants from our sample DBMS SPL. In Figure 3.4, we use those variants to show the instantiation of a sample Cellular database, where we use different variants for different columns in a table. Each column can consist of multiple cells according to the cell types we defined in Section 3.4.

In our sample scenario of Figure 3.4, we show the realization of three different tables in a hypothetical sample Cellular database. The three tables are namely Personnel, Gender, and Country. The Gender and Country tables are related to the Personnel table using foreign key relationships, i.e., keys from the Gender and Country tables are used in the Gender and Country columns of the Personnel table to reference the data from the Gender and Country tables. It can be seen that we used smallest possible variant 1 of the Cellular DBMS SPL (shown in Figure 3.3) to instantiate the Gender table with single column. The Gender table stores only two entries, i.e., Male and Female. A simple in-memory array structure should be sufficient to store this data. We used variant 2 to instantiate the Country table, which is expected to store around 203 records. An in-memory sorted list structure

should be sufficient to store the list of countries. The reason of making these two tables in-memory is the high frequency of their use, e.g., frequent retrieval by a GUI during data entry operations, frequent retrieval by a reporting tool during report generation, etc. Moreover, the data in these two tables is not expected to change for long time. The Personnel table in Figure 3.4 contains four columns, three of them instantiated as variant 4, whereas one of them (i.e., Car number) is instantiated as variant 3. The three columns (i.e., Name, Gender, and Country) are instantiated as variant 4, because all of them are expected to grow similar in size with no NULL values in any of them. However, we instantiated the Car number column as variant 3, because it may contain NULL values, i.e., it is expected to be smaller in size as compared to other columns. Furthermore, a person can change car many times requiring the Car number column to be updated frequently. Therefore, we used variant 3, which use the heap list that is efficient for frequent write and update operations.

Benefits of using the software product line approach Here, we summarize the benefits of the SPL approach that we found in literature [Leich et al., 2005; Rosenmüller et al., 2009a; Saake et al., 2009], as well as the ones that we observed from our own experience. A detailed discussion on our experience with the SPL approach can be found in Chapter 5.

- Negligible overhead on database performance for unused functionalities
- Deployment package holds only functionalities for which the client has paid
- Cost effective for client in terms of product price
- Common code-base for all database products
- Easy to manage the DBMS code-base
- Common features of different database products become more mature through rigorous testing and usage
- Better support for hardware and platform heterogeneity

Drawbacks of using the software product line approach The software product line approach also comes-up with few problems that we want to outline here. We provide a detailed discussion of our experience for using the SPL and problems we faced in Chapter 5.

- High initial customization effort
- Large code-base is more prone to bugs because of high interdependencies among features
- Complex product testing process [Kästner et al., 2011]
- High source code redundancy [Schulze et al., 2010]
- SPL is not yet mature, lack of tool support hinders the development and maintenance for large DBMS SPL

3.5.2 Using the aspect-oriented programming to realize autonomy

In the Cellular DBMS architecture, we propose using aspect-oriented programming (AOP) to realize an autonomic behavior in a DBMS. We classify autonomy as a cross-cutting behavior, i.e., it is required by many functionalities in a DBMS. If autonomy is implemented using normal classes and components, then it will be difficult to keep the autonomy source code separate from the other functionalities. It will get tightly coupled with other functionalities. Implementing autonomy using AOP allows us to keep the implementation clean and with proper separation. For monitoring functionality of autonomy, we suggest to exploit the dynamic join point model of AOP, which allows us to perform the monitoring of required data management functionality without implementing monitoring as a separate source code. Our mechanism to implement autonomy is similar to the concept of feedback control loop mechanism suggested by other researchers [Hellerstein, 1997; Weikum et al., 2002]. Furthermore, when included, our monitoring implementation is an integrated monitoring functionality and results in negligible overhead, which is also in agreement with the results from Thiem and Sattler [2009]. The details about our autonomy implementation in the Cellular DBMS prototype is provided in Section 5.3.

3.6 Related work

Use of the "Cellular DBMS" term in literature We found the use of the Cellular DBMS term in literature and industry. However, the use of the Cellular DBMS term in our architecture is different from its former use. We use the term cell for an atomic and autonomic instance of an embedded database variant, whereas, we call our DBMS Cellular, because it is composed of multiple such cells. The Infobionics company[1] uses the term Infobionics Cellular DBMS for their Infobionics Knowledge Server. According to an Infobionics news release[2], "The Infobionics Cellular DBMS places information in individual Data Cells, which can be flexibly compiled via Link Cells into an infinite number of DataSets". However, in patent [Sabry et al., 2003] it is stated as "A system for acquiring knowledge from cellular information. The system has a database comprising a database management module ("DBMS")." The last news release from the company listed on their website is dated 27 January 2009. Internal architectural details of the Infobionics Cellular DBMS are not publicly available, however, based on the available information in the form of patent [Sabry et al., 2003] and press release[2], we found our work quite different in terms of both concept and implementation, because we attempt to work in the direction of revisiting existing database architecture using the RISC-style approach exploiting innovative software engineering approaches, which is quite unique in its own.

Kersten [1998] proposed an architecture for a cellular database system. According to the proposal, each cell is a bounded container, i.e., a workstation or a mobile unit linked into a communication infrastructure. It assumes the Internet as the underlying communication network. This work also envisions a cell as an autonomous DBMS as we do, however, realization of autonomy is different in our approach (discussed in detail in Section 3.4). Furthermore, we suggest freedom of using any customizable embedded database as cell.

Kersten et al. [2003] along with other researchers again tried to draw the focus of database community towards organic databases. In 2006, Kersten and Siebes took a step forward with the concept of an organic database system. They provided the vision of new database architectures as "an Organic Database System where a large collection of connected, autonomous data cells implement a semantic meaningful store/recall information system" [Kersten and Siebes, 2006]. In the Cellular DBMS architecture, we also started with similar inspiration. We wanted to use biological

[1] "The Infobionics Knowledge Server", `http://www.infobionics.com/`, Accessed: 21-06-2011

[2] "Cellular DBMS Seeks Business Intelligence Beta Sites", PRESS RELEASE, infobionics, `http://www.infobionics.com/news/news_2/file_item.pdf`, Accessed: 21-06-2011

inspiration for data management. It is also one reason to call our architecture the Cellular DBMS architecture and the smallest data management unit a cell. For example, the concept of starting the data management with a single cell and then induction of more cells with increasing data size is inspired from the concept of *Binary Fission* [Angert, 2005]. In binary fission, biological cell grows to twice of its starting size and then splits-up into two cells, each cell having a complete copy of its essential genetic material. Not exactly, but similarly each Cellular DBMS cell splits the data into two equal halves. One-half is left in the parent cell where as the other half is moved to a newly induced cell. However, we skipped this dimension of our work in this thesis, to avoid any further difficulty of understanding by inclusion of another domain.

Verroca et al. [1999] used the term Cellular Database for a solution for cellular network data management. Toshio et al. [2002; 2004] proposed a Cellular DBMS that is based on the layer model. It is based on incremental modular abstraction hierarchy. They have applied the cellular model to model web-based information spaces for designing the Cellular DBMS [Toshio and Toshiyasu, 2002].

Embedded databases The Cellular DBMS architecture takes many inspirations from Berkeley DB [Olson et al., 1999], an embedded database system. Key/value pairs, API-based access, main-memory database, and small footprint for database; all these concepts have their counterpart in Berkeley DB. FAME-DBMS [Rosenmüller et al., 2008] is another customizable embedded database developed based on the software product line approach. Our Cellular DBMS prototype implementation emerged from the prototype implementation of FAME-DBMS; however, the concept of the Cellular DBMS architecture can be implemented using any customizable embedded database. We have many unique features in the Cellular DBMS prototype that were not part of the FAME-DBMS prototype, such as column-oriented storage, different cell type implementations, autonomy, evolution, etc. It is not an exhaustive list of features for the Cellular DBMS prototype implementation. Data management of an embedded system is the focus of the FAME-DBMS; in contrast, the Cellular DBMS architecture is not confined to data management for embedded systems. The FAME-DBMS focus derivation of a concrete instance of a database by composing features of a database product line, whereas the Cellular DBMS uses one or more instances of any customizable embedded database and exploits them in concert for data management.

Slim-down approach vs. bottom-up approach for DBMS customization We classify DBMS customization approaches into two categories, i.e., the slim-down approach and the bottom-up approach. The slim-down approach is used for customizing existing DBMS. It requires identification of separable functionalities and their dependencies. For existing data management solutions the slim-down approach is a difficult choice. According to our discussion in this chapter, existing DBMS have many functionalities and these functionalities are tightly integrated with each other. It makes them difficult to customize using the slim-down approach. Harizopoulos et al. [2008] used a similar approach to slim-down an existing Shore version. They reported the drawback of this approach through identification of difficulties in removing all references to unused or pruned functionalities. Rosenmüller et al. [2009a] used the slim-down approach to down-size Berkeley DB. However, they also used the software product line approach to achieve the down-sizing by careful pruning of functionalities that they refactored as features. A bottom-up approach suggests the development of the DBMS from scratch with customization as a main design goal. The Cellular DBMS architecture is designed according to this approach. The bottom-up approach requires additional effort to rebuild the DBMS; however, at the same time it allows the implementation of clean loosely-coupled functionalities for DBMS. The bottom-up approach has also been used by other projects, such as FAME-DBMS. They reported benefits of this approach, such as high customizability [Rosenmüller et al., 2008]. Considering the limitation of slim-down approach for existing DBMS, we suggest that bottom-up approach is a better alternative for creating specialized DBMS using customization.

Other RISC-style data management solutions Chaudhuri and Weikum in their VLDB 2000 paper suggested the transition towards a self-tuning RISC-style database system architecture, however, until the writing of this thesis in 2011, we observed only two data management solutions that made use of their or a similar design. BAT algebra[3] used by MonetDB[4] [Boncz et al., 2008] is one solution designed according to RISC-style architecture, which processes a query with the column-at-a-time approach with each operator operating on one or two columns. The second solution is RDF-3X, an implementation of SPARQL that is a RISC-style engine for RDF from Neumann and Weikum [2008].

[3] "BAT Algebra: the RISC approach to Query Languages" `http://monetdb-xquery.org/MonetDB/Version4/Documentation/monet/index.html`, Accessed: 21-06-2011

[4] "MonetDB", `http://www.monetdb.org/`, Accessed: 21-06-2011

Other approaches for DBMS customization Irmert et al. [2009] presented a component based approach for DBMS adaptation and extension at the runtime. They termed their approach Component Based Runtime Adaptable DataBase (CoBRA) DB. They made use of service-oriented component model [Cervantes and Hall, 2004] to achieve a runtime adaptable environment, where each component implements at-least one service and the components that implement the same service are interchangeable.

AOP for autonomy The use of AOP to implement autonomic behavior is not a new concept. Many researchers in the past have used it successfully to develop autonomic systems. Greenwood and Blair [2004] outlined the case of the use of dynamic AOP for autonomic systems. Truyen and Joosen [2008] demonstrated the applicability of AOP for implementing self-adaptive frameworks. Tesanovic et al. [2004] proposed the concept of aspectual component-based real-time system development (ACCORD) and applied it successfully in the design and development of a component-based embedded real-time database system (COMET). In the Cellular DBMS architecture, we use an AOP based model to implement autonomic behavior at the cell as well as at the DBMS level.

3.7 Summary

This chapter introduced the Cellular DBMS architecture as a customizable and autonomous database architecture. The Cellular DBMS architecture proposed to use multiple instances of customized autonomous embedded database as RISC-style data managers, which we termed as cell. From software engineering perspective, a cell is an instance of a variant of a DBMS SPL. We made each cell small, simple, consistent with an interface, and atomic in its operations according to the suggestion from Chaudhuri and Weikum [2000]. We used the software product line approach to customize embedded databases according to the work from the FAME-DBMS project [Leich et al., 2005; Rosenmüller et al., 2008]. This chapter also introduced the design principles related to autonomy in the Cellular DBMS architecture, which includes the details of the different cell types and an evolution process that is mandatory to realize an autonomic behavior. We also explained our use of AOP to realize autonomy in the Cellular DBMS architecture.

4 A customizable and self-tuning storage manager

This chapter shares material with the FIT'10 paper "Using Evolving Storage Structures for Data Storage" [ur Rahman, 2010] and the BNCOD'11 paper "ECOS: Evolutionary Column-Oriented Storage" [ur Rahman et al., 2011].

In the previous chapter, we outlined and explained all the design principles that we have defined for the Cellular DBMS architecture. In this chapter, we explain our realization of those concepts for a real DBMS implementation. In the previous chapter, we have introduced the Cellular DBMS architecture concepts for a complete DBMS; however, here onwards we confine our discussion at the storage manager level. This chapter also explains how the relational model can be realized using the Cellular DBMS architecture.

We present a customizable and self-tuning storage manager that we designed and implemented according to the Cellular DBMS architecture. We named the storage manager as Evolutionary Column-oriented Storage (ECOS). ECOS supports the storage model customization for each table and the storage structure customization for each column. Each column in ECOS self-tunes itself with the data growth and according to the workload characteristics. ECOS uses the decomposed storage model (DSM) [Copeland and Khoshafian, 1985], however, we also proposed four variations of the standard 2-copy DSM that can be used as an alternative. ECOS uses the mechanism of evolution paths to keep the human intervention for the self-tuning to a minimum.

Storage structure	Complexity class	Data size class	Benefits	Problems
Sorted Array	Simple	Small	Read optimized (Good data reference locality) Cache and space efficient	Write/Update (Requires rearrangement)
Heap Array	Simple	Small	Write optimized	Search time (Poor data reference locality) Complete scan for duplicates
Sorted List	Average	Medium	Read optimized	Write/Update(Requires rearrangement)
Heap List	Average	Medium	Write optimized	Search time (Poor data reference locality) Complete scan for duplicates
Hash Table	Average/Complex	Medium/Large	Write optimized Memory efficient Unordered data access	High space overhead (For dynamic hash tables) It does not preserve order Complete bucket scan for duplicates Range queries
B-Tree	Average/Complex	Medium/Large	Suited for disk use Good for memory Fast search Fast update	Poor cache behavior (Because of pointers)
T-Tree	Average/Complex	Medium/Large	Good for memory Fast search Fast update Ordered data access	Poor cache behavior (Because of pointers)
B+-Tree	Average/Complex	Medium/Large	Suited for disk use Fast search and update More cache conscious Range queries efficient	Not good for main memory

Table 4.1: Storage structures classification (uses the results provided by Lehman and Carey [1986]).

4.1 Motivation

Different storage structures in existing data management solutions have different execution complexity. By storage structure, we mean the data structure used by the storage manager to physically store data and indexes. For example, sorted array, heap list, T-Tree, and B+Tree. By execution complexity, we mean the memory footprint, function calls, branches and mispredictions, cache references and misses, etc., caused by a storage structure during data management operations. We use the term storage manager in its standard meaning for DBMS, i.e., a component to physically store and retrieve data. Data storage efficiency is assumed to be the main goal for a storage manager.

We classify storage structures complexity into three categories, i.e., simple, average, and complex. We argue that simple storage structures are appropriate for small database management. They consume fewer resources in comparison with complex storage structures. With an increase in data size, average complexity storage structures start performing better with appropriate resource consumption in comparison with simple and complex storage structures. For large database management, complex storage structures are the appropriate solutions. To be more concrete with our example, we use a sample classification in Table 4.1, which uses the results pro-

vided by Lehman and Carey [1986]. It can be observed from Table 4.1 that different storage structures are suitable for different workloads and data sizes. Each storage structure exposes different merits and demerits. We cannot find a universal storage structure that can perform optimally for all data sizes and workloads with appropriate resource consumption. To prove our argument, we evaluated different storage structures over different data sizes with the similar workload. The evaluation results are provided in Chapter 6.

The requirement for workload and data size specific customization is also suggested by other research and commercial data management solutions. C-Store [Stonebraker et al., 2005] proposed the use of two different data stores within same DBMS, i.e., read-optimized and write-optimized stores. Another customization C-Store proposed is that the write-optimized store operates in main-memory fashion. Dynamo [DeCandia et al., 2007], a highly available key-value store from Amazon, uses pluggable architecture for storage engine. It enables the choice of the storage engine that best suits the data management need for application, i.e., Berkeley DB can be used to store a database of few kilo bytes, whereas for database of large size, MySQL can be used [DeCandia et al., 2007]. MySQL DBMS also supports storage engine customization at the table-level.

In real world scenarios, we face diversified data management needs. Selecting appropriate storage structures for specific scenarios require extensive tuning. There exists a need for a mechanism that should facilitate appropriate storage structure selection and tuning with minimum human intervention. We make use of concepts of evolving hierarchically-organized storage structures and the evolution path as an alternative solution, which enables us the selection of an appropriate storage structure through customization. They also support automatically adjusting the storage structures with change in the data management needs.

Why hierarchically-organized storage structures? A hierarchical organization of storage structures is a composition of similar or different storage structures in a hierarchy as depicted in Figure 4.1. The hierarchically-organized storage structure is the realization of the concept of HLC cell that we introduce in Chapter 3. Figure 4.1 is an example of the vertically hybrid HLC cell. Initially, these structures do not have any hierarchy, i.e., they are initialized as a simple cell. They increase their hierarchy with data growth through inclusion of new storage structures (induce cells) using the concept of evolution that we introduced in Chapter 3. We suggest that these structures provide us an opportunity for selection of appropriate storage structures

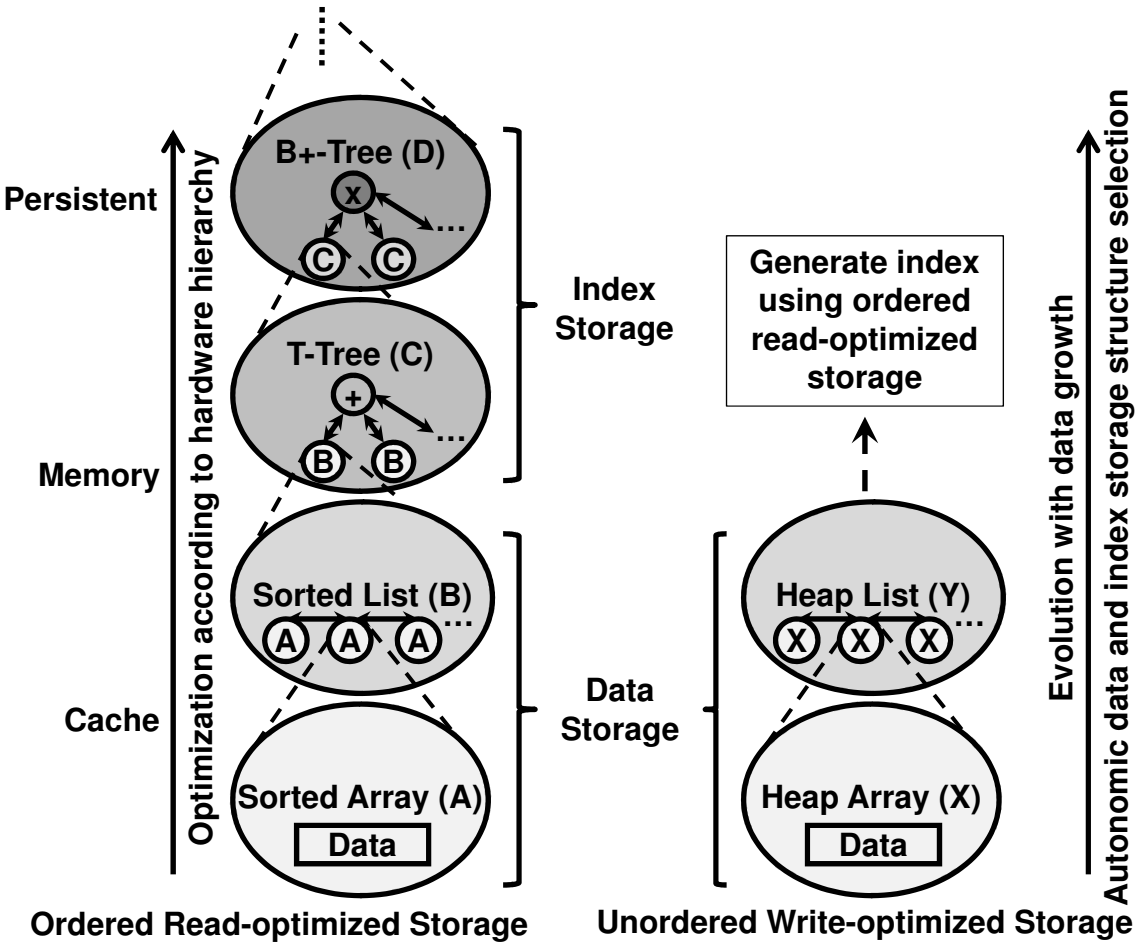

Figure 4.1: Evolving hierarchically-organized storage structures.

along the hierarchy through the analyzes of existing data and gathered statistics for current storage structures. Another benefit is better utilization of the hardware hierarchy. Previously published results from Bender et al. [2000] and Chen et al. [2002] also motivates our decision for the use of hierarchically-organized storage structures. Bender et al. [2000] presented a weight-balanced B-Tree organized according to the van Emde Boas layout, showing that it is capable of achieving near-optimal performance on any memory hierarchy. Chen et al. [2002] presented fractal prefetching B+-Trees, which embed "cache-optimized" trees within "disk-optimized" trees showing better cache performance in comparison with disk-optimized B+-Trees.

Why column-oriented storage model? The column-oriented storage model is derived from the earlier work of DSM [Copeland and Khoshafian, 1985]. DSM is a transposed storage model [Batory, 1979] that stores all values of the same attribute of the relational conceptual schema relation together [Copeland and Khoshafian,

1985]. DSM is a natural choice as a storage model for the Cellular DBMS architecture implementation, and it can be realized using the concept of the composite cell. Copeland and Khoshafian [1985; 1986] concluded many advantages of DSM including:

- Simplicity (Copeland and Khoshafian related it to RISC [Patterson and Ditzel, 1980])
- Less user involvement
- Less performance tuning requirement
- Reliability
- Increased physical data independence and availability
- Support of heterogeneous records

Plattner [2009] suggests that column-oriented storage model is best suited for modern CPU. It allows a DBMS to better utilize CPU cache and parallel processing capabilities. He also suggests that column storage performs superior to row storage with regards to memory consumption. They are also known for their superior performance for analytical data applications [Stonebraker et al., 2005]. The advantages listed above give strong motivation for use of the DSM in a self-tuning storage manager.

Why customization at the column-level? Table 3.1 on page 27 includes some characteristics of the TPC-H schema LINEITEM table. We can observe that distinct data count (cardinality) for all columns is different. We can classify three types of columns according to distinct data count, i.e., large, medium, and small. We further observed (general observation) the TPC-H queries that access LINEITEM table and predicted (using a layman-approach) the workload and data access pattern for columns. We identified that four columns (i.e., L_DISCOUNT, L_TAX, L_EXTENDEDPRICE, and L_QUANTITY) involve read-intensive workload, whereas three columns (i.e., L_COMMITDATE, L_SHIPDATE, and L_QUANTITY) involve ordered data access. The differences in distinct data count, workload, and data access pattern for different columns raise the need for the support of storage structure customization at the column-level. If a storage manager supports column-level customization of storage structure, we can hypothetically customize LINEITEM table columns as shown in Table 3.1.

4.2 Evolutionary Column-oriented Storage (ECOS)

In this section, we explain the concepts of ECOS in detail. We explain the DSM and four DSM based schemes that we proposed to reduce the high storage requirements of the standard 2-copy DSM. We also discuss the concepts of column customization, hierarchical-organization of the storage structures, evolution of the storage structures, and the evolution path.

4.2.1 Table-level customization

ECOS is a customizable self-tuning storage manager. It stores data according to the column-oriented storage model, where each column stores a key/value pair of data. It realizes the column-oriented storage model using the design of the composite cell. ECOS supports customization of the storage model for each table. We use five variations of the DSM for this purpose, which are: Standard 2-copy DSM [Copeland and Khoshafian, 1985], Key-copy Decomposed Storage Model (KDSM), Minimal Decomposed Storage Model (MDSM), Dictionary based Minimal Decomposed Storage Model (DMDSM), and Vectorized Dictionary based Minimal Decomposed Storage Model (VDMDSM). The motivation for proposing and testing different variations of DSM arises from high storage requirements of standard 2-copy DSM. For example, we have eight tables in the TPC-H schema. By table-level customization, we mean selecting an appropriate storage scheme from above-mentioned DSM based schemes for each table. The details for the five variations of DSM are as follows.

Table 4.2: DSM.

Columnk0		Columnk1		Columnk2	
Key	**Value**	**Key**	**Value**	**Key**	**Value**
k1	731	**k1**	20090327	**k1**	Jana
k2	137	**k2**	20071201	**k2**	Tobias
k3	173	**k3**	20010925	**k3**	Christian
k4	371	**k4**	20090327	**k4**	Tobias
k5	317	**k5**	20090327	**k5**	Tobias
k6	713	**k6**	20010925	**k6**	Jana

(a) Columns clustered on key

Columnv0		Columnv1		Columnv2	
Key	**Value**	**Key**	**Value**	**Key**	**Value**
k2	137	**k3**	20010925	**k3**	Christian
k3	173	**k6**	20010925	**k1**	Jana
k5	317	**k2**	20071201	**k6**	Jana
k4	371	**k1**	20090327	**k2**	Tobias
k6	713	**k4**	20090327	**k4**	Tobias
k1	731	**k5**	20090327	**k5**	Tobias

(b) Columns clustered on value

Standard 2-copy DSM The DSM is a transposed storage model [Batory, 1979], which pairs each value of a column with the surrogate of its conceptual schema

record as key [Copeland and Khoshafian, 1985]. It suggests storing two copies of each column, one copy clustered on values, whereas another copy is clustered on keys. We took the DSM as the base storage model and then altered it to propose different schemes. We suggest that the DSM is suitable for read-intensive workloads where data contain a negligible number of duplicates and NULL values, write and updates are minimal relative to read operations and there are negligible storage constraints. An example for the DSM is depicted in Table 4.2. We argue that for a self-tuning storage manager, the 2-copy DSM is the most suitable storage model. It is easy to implement and easy to use, moreover, it does not require human intervention to identify, which column to cluster or index, instead it is done in a uniform way [Valduriez et al., 1986]. To justify our argument, we evaluated the standard 2-copy DSM with four other proposed variations and found it the most appropriate one in terms of performance but at the cost of an additional storage requirement. The evaluation results are presented in Chapter 6.

Table 4.3: KDSM.

Columnk0		Columnk1		Columnk2	
Key	**Value**	**Key**	**Value**	**Key**	**Value**
k1	731	**k1**	20090327	**k1**	Jana
k2	137	**k2**	20071201	**k2**	Tobias
k3	173	**k3**	20010925	**k3**	Christian
k4	371	**k4**	20090327	**k4**	Tobias
k5	317	**k5**	20090327	**k5**	Tobias
k6	713	**k6**	20010925	**k6**	Jana

(a) Columns clustered on key

Columnv0	
Key	**Value**
k2	137
k3	173
k5	317
k4	371
k6	713
k1	731

(b) Columns clustered on value

Key-copy decomposed storage model (KDSM) The KDSM is the first variation of the DSM that we propose to reduce the high storage requirements of the standard DSM. The KDSM stores the data similar to the DSM, i.e., for each column, data is stored in values, whereas keys are unique numeric values that relate attributes of a row together. All columns are clustered on the keys. However, unlike the DSM, we store an extra copy of only key columns (primary key or composite primary key) clustered on values. This design alteration reduces the storage requirement of the KDSM, but it increases the access time for read operations that involve non-key columns in search criteria. However, for read operations with the key column in the search criteria it performs similar to the DSM with less storage requirements. We

propose the use of the KDSM for tables that mostly require querying data using key columns. The KDSM allows a conversion to the DSM by simply creating a copy of the non-key columns clustered on values. We suggest that the KDSM is suitable for data storage where columns have few duplicates and NULL values. An example for the KDSM is shown in Table 4.3.

Table 4.4: MDSM.

Columnk1	
Key	**Value**
k1	20090327
k2	20071201
k3	20010925
k4	20090327
k5	20090327
k6	20010925

(a) Columns clustered on key

Columnk2	
Key	**Value**
k1	Jana
k2	Tobias
k3	Christian
k4	Tobias
k5	Tobias
k6	Jana

Columnv0	
Key	**Value**
k2	137
k3	173
k5	317
k4	371
k6	713
k1	731

(b) Primary key columns clustered on value

Minimal decomposed storage model (MDSM) The MDSM stores the data similar to the DSM except that we do not store any extra copy for any columns thus reducing the high storage requirement of the DSM to a minimum. Instead, the design idea of the MDSM is to store primary key columns clustered on values, whereas non-primary key columns are clustered on key as depicted in Table 4.4. The MDSM performs similar to the DSM and the KDSM for the read operations with search criteria on key column attributes, but it performs worst for the read operations with non-key column attributes in search criteria. The MDSM can be transformed into the KDSM and the DSM by creating an extra copy of the key columns clustered on key and non-key columns clustered on values. However, our results in Chapter 6 suggest that if we do not have any space constraints, this scheme is not recommended.

Dictionary based minimal decomposed storage model (DMDSM) To improve the performance of the MDSM, we introduced the DMDSM, which stores the unique data values for each column separately as a dictionary column. The DMDSM is inspired from the concept of the dictionary encoding scheme, which is frequently used as light-weight compression technique in many column-oriented data management systems [Abadi et al., 2006; Lemke et al., 2010]. In the DMDSM, for each column,

Table 4.5: Dictionary columns for DMDSM and VDMDSM.

Dict. Column 0	
Keyd0	**Valued0**
d02	137
d03	173
d05	317
d04	371
d06	713
d01	731

Dict. Column 1	
Keyd1	**Valued1**
d11	20090327
d12	20071201
d13	20010925

Dict. Column 2	
Keyd2	**Valued2**
d23	Christian
d21	Jana
d22	Tobias

(a) Dictionary columns

Table 4.6: DMDSM.

Columnv0	
Keyv0	**Valuev0**
k2	d02
k3	d03
k5	d05
k4	d04
k6	d06
k1	d01

(a) Primary key columns clustered on value

Columnk1		Columnk2	
Key	**Value**	**Key**	**Value**
k1	d11	**k1**	d21
k2	d12	**k2**	d22
k3	d13	**k3**	d23
k4	d11	**k4**	d22
k5	d11	**k5**	d22
k6	d13	**k6**	d21

(b) Columns clustered on key

Table 4.7: VDMDSM.

Vector Column	
Key	**Value**
v1	d01,d11,d21
v2	d02,d12,d22
v3	d03,d13,d23
v4	d04,d11,d22
v5	d05,d11,d22
v6	d06,d13,d21

(a) Vector column

values are the keys for the data from dictionary column as depicted in Table 4.6. All dictionary columns are clustered on value. All other concepts for the DMDSM are similar to the MDSM, which is presented in Table 4.4. The DMDSM is suitable for tables with many duplicates or NULL values. In this scheme, for columns, database operators always manipulate numeric data for data management operations, which execute much faster on modern hardware [Lemke et al., 2010]. Furthermore, it gives us the provision to exploit our innovative concept of evolving hierarchically-organized storage structures (discussed in Section 4.2.2) to its maximum potential for dictionary columns because they only store non-null unique data and most of them can be stored using simple and small storage structures.

Vectorized dictionary based minimal decomposed storage model (VDMDSM) In the DMDSM, each column stores keys/values, where values are record identifiers from dictionary columns (see Table 4.6). We can optimize this with a better storage scheme by avoiding the storage of keys for every column separately. The VDMDSM is an extension of the DMDSM, such that it stores the values (i.e., dictionary column

keys) for all columns together as a vector in the vector column, i.e., instead of saving each column separately, it generates the vector of all attributes in the row and stores it as a value for vector column as depicted in Table 4.7. Similar to the DMDSM, the VDMDSM provides the opportunity to exploit the benefit of evolving hierarchically-organized storage structures to their full potential for dictionary columns. The VDMDSM is suitable for tables with many duplicate or NULL values.

The VDMDSM needs a special mechanism for searching the data, because the table only contains a single vector column with vectorized data stored in it as values. For searching, first we need to create a search vector to search for the data in the vector column. One approach could be to search in the vector column for each column identifier from the search vector one by one. This could be quite inefficient and will perform similar to the DMDSM with a little benefit in terms of storage space, which could be nullified with the overhead of vectorization. Rather, the goal is to make use of search vector all together to traverse the all related record. A naive implementation could be to scan the vector column for search vector using plain linear search, which we assume as our solution in this thesis. A more appropriate implementation could be to use a special storage structure to store the vector column, which could enable us to extract the data with a minimum scan. However, it is part of our future work.

4.2.2 Column-level customization and storage structure hierarchies

Once we select the appropriate storage model scheme from above-mentioned schemes at the table-level, we move forward to customize the columns. At the column-level, we customize the storage structure for each column. Each column is initially customized as either ordered read-optimized or unordered write-optimized storage structure. For ordered read-optimized storage structures, we store data in sorted order with respect to key or value, whereas for unordered write-optimized storage structure, we store data according to insertion order.

Evolving hierarchically-organized storage structure ECOS utilizes hierarchically-organized storage structures for data and index storage, such that a storage structure at each new level of hierarchy is composed of multiple lower level storage structures as depicted in Figure 4.1, which is according to the design principles we introduced in Chapter 3. The usage of hierarchically-organized storage structures is motivated by

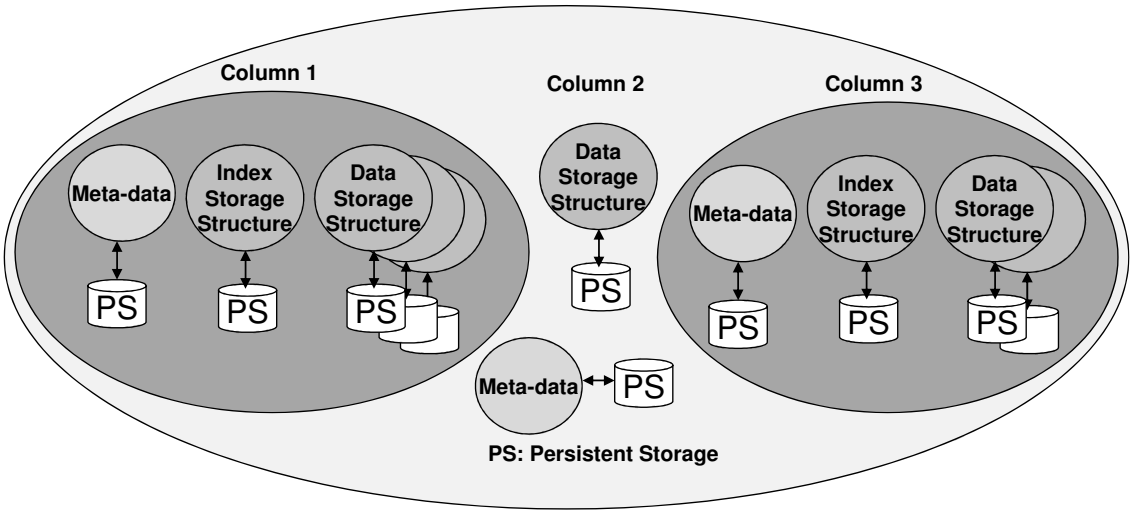

Figure 4.2: Evolutionary column-oriented storage.

the possibility of optimizing the storage structure hierarchy according to hardware hierarchy and data management needs. For example, consider the memory hierarchy in modern hardware. We optimize storage structures for cache, main memory, and persistent storage in the specified order. As shown in Figure 4.1, the lowest level of hierarchy is using array storage structures, which are optimized for cache. On the second level above, T-Tree storage structure is used, which is optimized for main memory. At the third level, B+-Tree is used, which is optimal for persistent storage.

The storage structures that we discuss in this thesis include heap array, sorted array, heap list, sorted list, B+-Tree, T-Tree, and hash table. From heap array/list, we mean a storage structure that always appends new data to existing data in chronological order and uses the linear search algorithm to traverse the data. From sorted array/list, we mean storage structures that always maintain the sort order for the data. For data retrieval sorted array uses the binary search algorithm. B+-tree, T-Tree, and hash table operate according to their de facto standards. Before we continue our discussion, we outline the hierarchically-organized storage structures, which we use further in our discussion. At the lowest level of hierarchy, we use:

Sorted array: It is the simplest storage structure, which is optimized for read-access with minimal space overhead. To use an array, we do not need to instantiate a buffer manager or an index manager.

Heap array: It is also the simplest storage structure, which is optimized for write-access with minimal space overhead.

At the next level above, we use a composite storage structure, which is according to the design of the composite cell introduced in Chapter 3. A sorted array can evolve into any of the below-mentioned three storage structures, whereas a heap array can only be evolved into a heap list.

Sorted list: Sorted list is composed of multiple sorted arrays. It requires the instantiation of a buffer manager for managing multiple sorted arrays.

Heap list: Heap list is composed of multiple heap arrays. It also requires the instantiation of a buffer manager for managing multiple heap arrays.

B+-Tree: B+-Tree is composed of multiple arrays as leaf nodes. It requires the instantiation of a buffer manager for managing multiple arrays as well as an index manager to manage multiple index nodes.

On the higher levels, we use HLC storage structures, which we also introduced in Chapter 3:

HLC SL: HLC SL is a B+-Tree based storage structure, where each leaf node is a sorted list. HLC SL instantiates a buffer manager to manage multiple sorted lists and an index manager to manage multiple index nodes. Each sorted list manages its own buffer manager, which ensures the high locality of data for each sorted list. HLC SL storage structure is depicted in Figure 4.3.

HLC B+-Tree: HLC B+-Tree is a B+-Tree based storage structure, where each leaf node is also a B+-Tree. HLC B+-Tree instantiates a buffer manager to manage multiple B+-Trees and an index manager to manage multiple index nodes. Each B+-Tree at leaf nodes manage its own buffer manager and index manager, which ensures the high locality of data and index nodes for each B+-Tree. HLC B+-Tree storage structure is depicted in Figure 4.3.

We provide a detailed theoretical explanation for evolving hierarchically-organized storage structure in Section 4.4.

Once a column is customized as either ordered read-optimized or unordered write optimized storage, ECOS initializes each column to the smallest possible storage structure according to the design principle of the Cellular DBMS architecture, i.e., ordered read-optimized column is initialized as a sorted array, whereas unordered write-optimized column is initialized as a heap array. According to the Cellular DBMS architecture design principles, ECOS enforces that each storage structure should be atomic and should be directly accessible using an access API. The reason

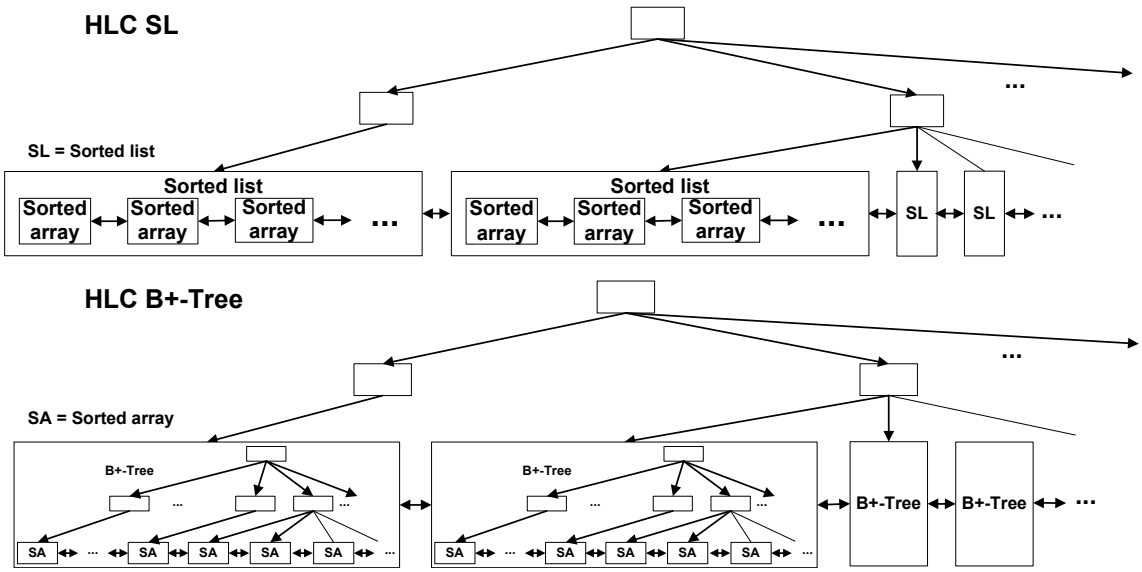

Figure 4.3: HLC SL and HLC B+-Tree storage structures in the Cellular DBMS prototype.

for this approach is that small storage structures consume less memory and generate reduced binary size for small data management [ur Rahman, 2010]. If we can use them directly, then there is no reason to use them as part of complex storage structures (we use storage structure as a common term for both data storage structure and index storage structure), such as B+-Tree or T-Tree; avoiding the overheads of complexity associated with these storage structures. This approach ensures that using smallest suitable storage structures, desired performance is achieved using minimal hardware resources for small database management.

Storage capacity limitation for predictable performance According to the Cellular DBMS architecture design principles, ECOS imposes data storage capacity limitation for each storage structure. We enforce this for more predictable performance and to ensure that storage structure performance does not degrade because of unlimited data growth. In ECOS, once limited storage capacity of a storage structure is consumed, it evolves to a larger more complex storage structure composed of multiple existing ones considering the important decision factors, such as hardware, the data growth, and the workload. For ordered read-optimized data storage, a sorted array is evolved into a sorted list, such that the sorted list is composed of multiple sorted arrays linked together. For unordered write-optimized data storage, a heap array is evolved into the heap list. The evolution of a storage structure is an important event for assessing the next suitable storage structure by analyzing the

existing data and the previously monitored workload.

Similarly, each new storage structure also has a definite data storage capacity limitation and once again as it is consumed, ECOS further evolves and increases the hierarchy of the hierarchically-organized storage structures. For ordered read-optimized data storage, once sorted list storage capacity is consumed it evolves into new storage structure, such that it becomes part of a new index structure. For example, it becomes the data leaf node of a B+-Tree. For ordered read-optimized data storage, ECOS does not perform data management operations separately for data and index structures, instead, each operation interact directly with the index structure. Here onwards, index structure will identify, in which sorted list the data will be stored. For unordered write-optimized storage, operations execute separately on data and index structures, such that first data is inserted into the heap list, and then the index structure is updated with the new key or index value. Index structures for unordered write-optimized storage are based on ordered read-optimized storage and will evolve subsequently.

API consistency to hide complexity and ensure ease of use To hide the complexity of different storage structure over different levels of hierarchy, ECOS keeps the interface for all storage structures consistent. We provide a standard interface to access columns with simple, Put(), Get(), and Delete() functionality with record as argument. It is invisible to an end-user, which storage structure is currently in use for each column.

Automatic partitioning ECOS separates physical storage for each column to reduce the I/O contention for storage of large database. For large columns, it also separates the data for a column into multiple separate physical storage units, which is similar to horizontal partitioning. In Figure 4.2, at a minimum, each column has its own separate physical storage. With the growth of data, each column may spread over multiple physical storage units. For example, for storage structures of Table 3.1, each sorted list or heap list is stored in a separate data file, whereas each B+-Tree or T-Tree is stored in a separate index file. These physical storage units may be stored on the single hard disk, or they may spread across the network. This separation also allows using different compression algorithms for each column (or each physical storage unit) based on the data type.

Meta-data for efficient traversal ECOS proposes to maintain important meta-data for efficient traversal of the hierarchically-organized storage structures, which includes count, minimum key/value, and maximum key/value for each storage structure. This avoids the access to unnecessary data and improves the efficiency of hierarchy traversal. ECOS also proposes to maintain the frequently used important aggregates, e.g., summation, average, etc., as the meta-data at every level of hierarchy. The request for these aggregates should be satisfied by accumulating them using the meta-data to reduce the overhead of accessing each value separately to calculate them again and again.

4.3 Evolution paths

Evolution path is the mechanism to define how ECOS evolves a smallest simple storage structure into a large complex storage structure. The path consists of many storage structure/mutation rules pair entries that ECOS uses to identify, how to evolve the storage structures. Each storage structure can have multiple mutation rules mapped to it. These mutation rules consist of three information elements, i.e., Event, Heredity based selection, and Mutation. The event identifies, when this mutation rule should be executed. Different mutation rules can have the same event, but not all of them execute the mutation. The heredity based selection identifies precisely, when evolution should occur based on the heredity information gathered for existing storage structure. Heredity information comprises the gathered statistics about the storage structure, e.g., workload type, data access pattern, previous evolution details, etc. The mutation defines the actions that should be executed to evolve the storage structure. A sample evolution path is shown in Table 4.8.

We envision that common DBMS maintenance best practices can be documented using the evolution path mechanism. ECOS assumes that DBMS vendors provide the evolution paths that best suit their DBMS internals, with the provision of alteration for a database administrator. The only liability for configuration that lies with database designers and administrator is to have a look at the evolution path for the DBMS and alter it with desired changes, if needed. The evolution process in ECOS is autonomic, and it exploits evolution path to automatically evolve the storage structures, i.e., our approach for self-tuning is online [Bruno and Chaudhuri, 2007b].

Consider the L_ORDERKEY column of the LINEITEM table as shown in Table 3.1. Suppose as a database designer, we design this table. According to our

Table 4.8: Example for evolution paths.

Storage Structure Initial	**Mutation Rules**	**Storage Structure 1st Evolution**	**Mutation Rules**	**Storage Structure 2nd Evolution**
Sorted array	**Event:** Sorted array=Full **Heredity based selection:** Workload=Read intensive Data access=Unordered **Mutation:** => Evolve (Sorted array − >Sorted list)	Sorted list	**Event:** Sorted list=Full **Heredity based selection:** Workload=Read intensive Data access=Ordered **Mutation:** => Evolve (Sorted list − >HLC SL)	HLC SL
Sorted array	**Event:** Sorted array=Full **Heredity based selection:** Workload=Read intensive Data access=Ordered **Mutation:** => Evolve (Sorted array − >B+-Tree)	B+-Tree	**Event:** B+-Tree=Full **Heredity based selection:** Workload=Read intensive Data access=Ordered **Mutation:** => Evolve (B+-Tree − >HLC B+-Tree)	HLC B+-Tree
Sorted array	**Event:** Sorted array=Full **Heredity based selection:** Workload=Write intensive Data access=Unordered **Mutation:** => Evolve (Sorted array − >Heap array)	Heap list according to heap array mutation rules		
Heap array	**Event:** Heap array=Full **Heredity based selection:** Workload=Write intensive Data access=Ordered **Mutation:** => Evolve (Heap array − >Heap list) & Generate secondary index (Sorted list)	Heap list + Index	**Event:** Heap list=Full **Heredity based selection:** Workload=Write intensive Data access=Ordered **Mutation:** => Evolve (Heap list− >Hash table) & Evolve secondary index (Sorted list − >HLC SL)	Hash table + Index
Heap array	**Event:** Heap array=Full **Heredity based selection:** Workload=Write intensive Data access=Unordered **Mutation:** => Evolve (Heap array − >Heap list)	Heap list	**Event:** Heap list=Full **Heredity based selection:** Workload=Write intensive Data access=Unordered **Mutation:** => Evolve (Heap list − >Hash table)	Hash table

application design, we select the L_ORDERKEY column as a part of the primary key. As we already discussed in Section 4, we have to customize each column as either ordered read-optimized or unordered write-optimized. Therefore, we customize the L_ORDERKEY column as ordered read-optimized as a sample case. However, we design according to the domain knowledge, our experiences, and predictions at the initial design time. As a designer, it is difficult to guarantee, how much this column grows, and how long it takes to reach that size. When we customize the column as ordered read-optimized, it is initialized as a sorted array. Now for the L_ORDERKEY column, three initial rows of the sample evolution path of Table 4.8 are relevant.

As we mentioned in Section 4, ECOS limits the storage capacity of each storage structure. Therefore, the initial sorted array has a certain data storage capacity limit. For example, consider it as 4 KB. As long as data is within the 4 KB limits, sorted array is the storage structure for the L_ORDERKEY column, and we gather the heredity information for the column, such as the number of Get(), the number of Put(), the number of Delete(), the number of point Get() (for point queries), the number of range Get() (for range queries), the number of Get() for all records (for scan queries), etc. What heredity information should be gathered may vary from one implementation to another. Here, we simplify our discussion by assuming that a system can identify using heredity information that the workload is either read-intensive or write-intensive and the access to data is either ordered (range) or unordered (point or all).

The moment the storage limit of the sorted array is consumed, an event is raised for notification. This event triggers all three initial mutation rules of Table 4.8. Now heredity based selection identifies, which one of them to execute. We suppose that for the L_ORDERKEY column, the workload is the read-intensive, and the data access is unordered, this scenario executes the first mutation rule of Table 4.8, which evolves the existing sorted array into a sorted list. Now-onwards sorted list is the storage structure for L_ORDERKEY column, and it is also constrained with the storage limit according to the design principle of ECOS. As long as the L_ORDERKEY column data is within the storage limit of the sorted list, heredity information is gathered, and it is used for the next evolution.

It is observed from Table 3.1 that only half of columns in LINEITEM table with high data growth (i.e., eight out of sixteen) evolves during first evolution (i.e., L_ORDERKEY, L_EXTENDEDPRICE, L_RECEIPTDATE, L_COMMITDATE, L_SHIPDATE, L_SUPPKEY, L_PARTKEY, and L_COMMENT). The rest of the

columns can be stored within an array (either heap array or sorted array). Furthermore, only half of the columns, i.e., four out of eight, which are evolved during first evolution evolve again during the second evolution (i.e., L_ORDERKEY, L_COMMENT, L_EXTENDEDPRICE, and L_PARTKEY). The final state of table presented in Table 3.1 shows that each column is using the appropriate storage structure (we assume for explanation) according to the stored data size and observed workload. We can add more parameters for evolution decision, but we only used limited parameters (i.e., data size, workload, and data access) to keep our discussion simple and understandable.

What heredity information should be gathered for each storage structure, and how to improve the efficiency of storage and retrieval of heredity information is a separate topic. Here, we simplify our discussion with an assumption that we have an efficient and precise mechanism for gathering heredity information. As a sample demonstration of how storage structures in the LINEITEM table evolves for the sample evolution path in Table 4.8 is shown in Table 3.1. Before we conclude this section, to avoid any confusion, we want to mention that the terms and concepts of evolution, evolution path, mutation rules, and heredity information used in this report have no relevance with their counterpart in evolutionary algorithms or any other non-relevant domain.

4.4 Theoretical explanation for evolving hierarchically-organized storage structures

In this section, we provide the theoretical explanation of evolving hierarchically-organized storage structures used in ECOS using time and space complexity analysis. As we explained in Section 4, we customize a column as either ordered read-optimized storage structure or unordered write-optimized storage structure. In both categories, many different combinations of storage structures are possible, however, we confine our discussion to the storage structures that we have implemented in our prototype implementation, i.e., sorted array, sorted list, HLC SL, heap array, heap list, and B+-Tree. We use three parameters that are common for both classes of storage structures, which are as follows:

n = Number of key/value pairs in a storage structure
$T(n)$ = Worst-case running time for operations

$S(n)$ = Worst-case space complexity for storage structure
E_i = Evolution overhead, where i is the evolution identifier, such that E_i occurs before E_{i+1} and $E_i < E_{i+1}$

4.4.1 Ordered read-optimized storage structure

For ordered read-optimized storage structure, we explain a storage structure that evolves from a sorted array to a sorted list (of sorted arrays) and then to a HLC SL (a B+-Tree based storage structure with sorted lists as data leaf nodes).

Initial storage structure (Sorted array) For sorted array, we only have one important parameter to consider, which is as follows:

n_{sa} = Maximum number of key/value pairs that can be stored as a sorted array

The time complexity for different data management operations for a sorted array is as follows:

$$\begin{aligned} Get &= \Theta(lg\ n_{sa})\ //Binary\ search \\ Put &= \Theta(n_{sa}) \\ Delete &= \Theta(n_{sa}) \end{aligned}$$

The space complexity for a sorted array is as follows:

$$S(n) = O(n_{sa})$$

As long as $n \leq n_{sa}$, the data storage structure will be the sorted array. When $n > n_{sa}$ evolution occurs, the existing sorted array becomes the part of a new data storage structure, e.g., a sorted list.

First evolution (Sorted array to sorted list) For a sorted list, we have three important parameters to consider, which are as follows:

n_{sl} = Maximum number of key/value pairs that can be stored as a sorted list
l_{sa} = Number of list blocks (sorted array) in a sorted list
n_p = Number of next and previous pointers in a sorted list

The time complexity for different data management operations for a sorted list is as follows:

$$\begin{aligned} Get &= \Theta(lg\ l_{sa}) + \Theta(lg\ n_{sa}) \\ Put &= \Theta(lg\ l_{sa}) + \Theta(n_{sa}) \\ Delete &= \Theta(lg\ l_{sa}) + \Theta(n_{sa}) \end{aligned}$$

The space complexity for a sorted list is as follows:

$$\begin{aligned} l_{sa} &= \frac{n_{sl}}{n_{sa}}\ \textit{//Number of sorted arrays in list} \\ n_p &= l_{sa} * 2\ \textit{//Number of next and previous pointers in a sorted list} \\ => \ n_p &= \frac{n_{sl}}{n_{sa}} * 2\ \textit{//Number of next and previous pointers in a sorted list} \\ \therefore\ S(n) &= O(n_{sl}) + O(\frac{n_{sl}}{n_{sa}} * 2) \end{aligned}$$

As long as $n \leq n_{sl}$, the data storage structure will be the sorted list. When $n > n_{sl}$ evolution occurs, the existing sorted list becomes the part of a new storage structure, e.g., B+-Tree, we term this storage structure as HLC SL.

Second evolution (Sorted list to HLC SL) HLC SL is a B+-Tree based storage structure with a sorted list as leaf nodes for storing data. For HLC SL, we have five important parameters to consider, which are as follows:

n_{bt} = Maximum number of key/value pairs that can be stored in a sorted list using HLC SL
l_{sl} = Number of sorted lists as data leaf nodes
t = Minimum degree of HLC SL B+-Tree, such that $t \geq 2$
k = Maximum number of elements in each node, such at each index node can have k-1 keys and k children where k=2t.
h = Height of the HLC SL B+-Tree

The time complexity for different data management operations for a HLC SL with a sorted list (of sorted arrays) as its data leaf node is as follows:

$$\begin{aligned} Get &= O(t\ log_t\ l_{sl}) + \Theta(lg\ l_{sa}) + \Theta(lg\ n_{sa}) \\ Put &= O(t\ log_t\ l_{sl}) + \Theta(lg\ l_{sa}) + \Theta(n_{sa}) \\ Delete &= O(t\ log_t\ l_{sl}) + \Theta(lg\ l_{sa}) + \Theta(n_{sa}) \end{aligned}$$

The space complexity for the HLC SL with a sorted list (of sorted arrays) as its data leaf node is as follows:

$$\begin{aligned} l_{sl} &= \frac{n_{bt}}{n_{sl}}\ \textit{//Number of sorted list as a data leaf node} \\ => \ S(n_{bt}) &= O(l_{sl})\ \textit{//We store one key for each sorted list} \\ \therefore\ S(n) &= O(l_{sl}) + O(n_{sl}) + O(\frac{n_{sl}}{n_{sa}} * 2) \end{aligned}$$

As long as $n \leq n_{bt}$, the data storage structure will be the HLC SL. When $n > n_{bt}$ evolution may again occur, however, we confine our discussion to this level. Overall ECOS behavior for our example of ordered read-optimized data storage structure with two levels of evolution can be summarized as follows:
Get:

$$T(n) = \begin{cases} \Theta(lg\ n_{sa}) & \text{if } n \leq n_{sa} \\ \Theta(lg\ l_{sa}) + \Theta(lg\ n_{sa}) & \text{if } n \leq n_{sl} \\ O(t\ log_t\ l_{sl}) + \Theta(lg\ l_{sa}) + \Theta(lg\ n_{sa}) & \text{if } n \leq n_{bt} \end{cases}$$

Put:

$$T(n) = \begin{cases} \Theta(n_{sa}) & \text{if } n \leq n_{sa} \\ \Theta(lg\ l_{sa}) + \Theta(n_{sa}) & \text{if } n \leq n_{sl} \\ O(t\ log_t\ l_{sl}) + \Theta(lg\ l_{sa}) + \Theta(n_{sa}) & \text{if } n \leq n_{bt} \end{cases}$$

Delete:

$$T(n) = \begin{cases} \Theta(n_{sa}) & \text{if } n \leq n_{sa} \\ \Theta(lg\ l_{sa}) + \Theta(n_{sa}) & \text{if } n \leq n_{sl} \\ O(t\ log_t\ l_{sl}) + \Theta(lg\ l_{sa}) + \Theta(n_{sa}) & \text{if } n \leq n_{bt} \end{cases}$$

Space complexity

$$S(n) = \begin{cases} O(n_{sa}) & \text{if } n \leq n_{sa} \\ O(n_{sl}) + O(\frac{n_{sl}}{n_{sa}} * 2) & \text{if } n \leq n_{sl} \\ O(l_{sl}) + O(n_{sl}) + O(\frac{n_{sl}}{n_{sa}} * 2) & \text{if } n \leq n_{bt} \end{cases}$$

4.4.2 Unordered write-optimized storage structure

As a second example, we discuss write optimized hierarchically-organized storage structures used in ECOS. For unordered write-optimized storage structure, we explain a heap array that evolves into a heap list, and then we generate a B+-Tree based index structure on the heap list, which further evolves as an ordered read-optimized storage structure.

Initial storage structure (Heap array) For heap array, we only have one important parameter to consider similar to the sorted array, which is as follows:

n_{ha} = Maximum number of key/value pairs that can be stored as a heap array

The time complexity for different data management operations for a heap array is as follows:

$$\begin{aligned} Get &= \Theta(n_{ha}) \text{ //Linear search} \\ Put &= \Theta(1) \\ Delete &= \Theta(1) \text{ //Mark deleted} \\ Defragmentation &= \Theta(n_{ha}) \text{ //Linear} \end{aligned}$$

By defragmentation, we mean an operation, which restructures an array to remove the empty spaces between the data that are generated because of delete operations. The space complexity for a heap array (with defragmentation) is as follows:

$$S(n) = O(n_{ha})$$

As long as $n \leq n_{ha}$, the data storage structure will be the heap array. When $n > n_{ha}$ evolution occurs, the existing heap array becomes part of a new data storage structure, e.g., heap list.

First evolution (Heap array to heap list) For a heap list, we have three important parameters to consider, which are as follows:

n_{hl} = Maximum number of key/value pairs that can be stored as a heap list
l_{hl} = Maximum number of list blocks(heap array) in a heap list
n_p = Number of next and previous pointers in the heap list

The time complexity for different data management operations for a heap list of heap arrays is as follows:

$$\begin{aligned} Get &= \Theta(n_{hl}) \text{ //Linear Search} \\ Put &= \Theta(1) \\ Delete &= \Theta(1) \text{ //Mark deleted} \\ Defragmentation &= \Theta(n_{hl}) \text{ //Linear} \end{aligned}$$

The space complexity for a heap list of heap arrays (with defragmentation) is as follows:

$$\begin{aligned} l_{ha} &= \frac{n_{hl}}{n_{ha}} \text{ //Number of heap arrays in a heap list} \\ n_p &= l_{ha} * 2 \text{ //Number of next and previous pointers in a heap list} \\ => n_p &= \frac{n_{hl}}{n_{ha}} * 2 \text{ //Number of next and previous pointers in a heap list} \\ \therefore S(n) &= O(n_{hl}) + O(\frac{n_{hl}}{n_{ha}} * 2) \end{aligned}$$

It can be observed that we do not get any benefit in terms of performance, when we evolve a heap array to a heap list. However, we should also consider here the possibility of evolving to different storage structure, e.g., hash table. Each evolution is the point to observe the statistics that we gather as long as previous storage structure is usable. These statistics give us insight for the workload on the column. For example, in case of a heap array evolving to a hash table, we have following

time complexity for new hash table storage structure:

$$\begin{aligned} Get &= \Theta(n_{ha}) \text{ //Ignoring the hash calculation and bucket} \\ &\qquad \text{selection overhead} \\ Put &= \Theta(1) \\ Delete &= \Theta(1) \text{ //Mark deleted} \\ Defragmentation &= \Theta(n_{hl}) \text{ //Linear} \end{aligned}$$

However, for our discussion, here we do not evolve a heap list to a hash table. As long as $n \leq n_{hl}$, the data storage structure will be the heap list. When $n > n_{hl}$ evolution occurs, however, in unordered write-optimized storage scenario, we do not evolve a heap list to any other storage structure. Instead, we use the heap list as the primary storage structure for data, and we generate index on it according to the statistics we generated while populating this heap list. Since an index is an ordered data storage structure, we use the evolving storage structure for storing an index as we have discussed above in Section 4.4.1. In this scenario, we assume that according to the gathered statistics, we identify B+-Tree as an appropriate index. Here we mean a standard B+-Tree, i.e., leaf node stores the pointer/identifier to data in the heap list.

Second evolution (Heap list with a B+-Tree as an index) For a heap list with a B+-Tree as an index, we have five important parameters to consider, which are as follows:

n_{ibt} = Maximum number of keys that can be stored in the B+-Tree
l_{hl} = Number of heap lists for data storage
t = Minimum degree of the B+-Tree, such that $t \geq 2$
k = Maximum number of elements in each node, such at each index node can have k-1 keys and k children where k=2t.
h = Height of the tree

The time complexity for different data management operations for a heap list of heap arrays with a B+-Tree as an index is as follows:

$$\begin{aligned} Get &= O(t\ log_t n_{ibt}) + \Theta(1) \\ Put &= O(t\ log_t n_{ibt}) + \Theta(1) \\ Delete &= O(t\ log_t n_{ibt}) + \Theta(1)\ //Mark\ deleted \\ Defragmentation &= O(t\ log_t n_{ibt}) + \Theta(n_{hl}) \end{aligned}$$

The space complexity for a heap list (with defragmentation) of heap arrays with a B+-Tree as an index is as follows:

$$\begin{aligned} S(n) &= O(n_{ibt}) + O(n_{hl}) + O(\frac{n_{hl}}{n_{ha}} * 2) \\ since\ n_{ibt} &= n_{hl}//Number\ of\ keys\ in\ B+\text{-}Tree\ is\ same\ as\ the\ number\ of \\ &\qquad records\ in\ a\ heap\ list \\ \therefore\ S(n) &= O(2 * n_{ibt}) + O(\frac{n_{hl}}{n_{ha}} * 2) \end{aligned}$$

As long as $n \leq n_{ibt}$, the data storage structure will be heap list with the B+-Tree as an index. When $n > n_{ibt}$ evolution may again occur for index storage structure, however, we confine our discussion to this level. Overall ECOS behavior for our example of unordered write-optimized data storage structure with two levels of evolutions is as follows:
Get:

$$T(n) = \begin{cases} \Theta(n_{hl}) & \text{if } n \leq n_{hl} \\ O(t\ log_t n_{ibt}) + \Theta(1) & \text{if } n \leq n_{ibt} \end{cases}$$

Put:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq n_{hl} \\ O(t\ log_t n_{ibt}) + \Theta(1) & \text{if } n \leq n_{ibt} \end{cases}$$

Delete:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq n_{hl} \\ O(t\ log_t n_{ibt}) + \Theta(1) & \text{if } n \leq n_{ibt} \end{cases}$$

Defragmentation:

$$T(n) = \begin{cases} \Theta(n_{hl}) & \text{if } n \leq n_{hl} \\ O(t\ log_t n_{ibt}) + \Theta(n_{hl}) & \text{if } n \leq n_{ibt} \end{cases}$$

Space complexity:

$$S(n) = \begin{cases} O(n_{ha}) & \text{if } n \leq n_{ha} \\ O(n_{hl}) + O(\frac{n_{hl}}{n_{ha}} * 2) & \text{if } n \leq n_{hl} \\ O(2 * n_{ibt}) + O(\frac{n_{hl}}{n_{ha}} * 2) & \text{if } n \leq n_{ibt} \end{cases}$$

It can be observed from above-provided time and space complexity analysis of evolving storage structures that for different database sizes, we obtain different resource consumption (i.e., we take both CPU time and storage space as resources). To simplify our discussion, we take an example of ordered read-optimized storage. It can be observed that space requirement of complex storage structure, such as the B+-Tree is high in comparison with the sorted array. Whereas insertion and deletion CPU time for the sorted array is high. However, as we have mentioned and discussed earlier, we restrict the data storage capacity of storage structure. This ensures that we keep the insertion and deletion time for each storage structure within the acceptable limit.

4.5 Related work

Hardware-oblivious approaches, such as cache-oblivious in-memory query processor EaseDB [He and Luo, 2008], cache-oblivious hashing [Pagh et al., 2010], and cache-oblivious B-Trees [Bender et al., 2000] are also important tools for self-tuning DBMS. Bender et al. [2000] proposed the cache-oblivious B-Trees that perform the optimal search across different hierarchical memories with varying memory levels, cache size, and cache line size. Fractal prefetching B+-Trees proposed by Chen et al. [2002] is the most relevant work for the ECOS and is similar in concept to cache-oblivious B-Trees with an additional concept of prefetching. Fractal prefetching B+-Trees are optimized for both cache and disk performance, which is also a goal for the ECOS. However, the ECOS concepts do not restrict the use of any fixed structure; instead it suggests the use of different storage structures in the hierarchy to support an efficient use of underlying hardware.

Lemke et al. [2010] presented the dictionary based compression technique to speed up the query processing in an in-memory column store. They replaced the original column by an index vector that stores only bit-compressed pointers to a dictionary. They further used the prefix, sparse coding, cluster coding, indirect coding, and run length coding techniques to compress the index vector for the column. These techniques can be used to improve our presented approaches of DMDSM and VD-MDSM.

Database cracking is an innovative approach proposed by Kersten and Manegold [2005]. It proposes the continuous physical reorganization of a database according to the query processing. It cracks the database into manageable pieces according to the user queries to decrease the access time and implementing self-organizing behavior. Our approach is different from the database cracking. ECOS in comparison implements self-organization at the storage manager level. ECOS evolves storage structures with data growth to ensure consistent performance while maintaining minimal resource consumption.

The partitioned B-Trees presented by Graefe [2003] has some similarities with our proposed HLC B+Tree storage structure. However, HLC B+Tree is not the only storage structure possible from our HLC design. In HLC B+Tree, we also have partitions and a separate B+-Tree for each partition, but these partitions are according to the keys or values rather than any artificial key column. Moreover, according to HLC design these partitions could be different storage structures, such as sorted list or T-Tree.

4.5.1 Column-oriented DBMS

There exist many column-oriented DBMS in industry as shown in Table 4.9[1]. We found only few of them important for further discussion based on their similarities with the Cellular DBMS. However, the purpose of this discussion is to introduce readers about the features of other existing column-oriented DBMS, rather than performing a comparison with the Cellular DBMS. The Cellular DBMS implementation is a research prototype and currently only includes a storage manager. Therefore, a comparison with other full-fledged DBMS is part of the future work.

[1]List of column-oriented DBMS is not exhaustive.

DBMS	Web Reference (Accessed: 21-06-2011)
MonetDB	http://www.monetdb.org/
Vertica (Formerly: C-Store)	http://www.vertica.com http://db.csail.mit.edu/projects/cstore/
Infobright (Formerly: Brighthouse)	http://www.infobright.com
HBase	http://hadoop.apache.org/hbase/
Kdb+	http://kx.com/Products/kdb+.php
TokuDB for MySQL	http://www.tokutek.com
Calpont	http://www.calpont.com
The ParAccel Analytic Database	http://www.paraccel.com
EXASolution	http://www.exasol.com
Sybase IQ	http://www.sybase.com/products/datawarehousing/sybaseiq
LucidDB	http://www.luciddb.org

Table 4.9: Column-oriented DBMS.

MonetDB MonetDB[2] [Boncz et al., 2008] is an open-source database system for high-performance applications (e.g., data mining, OLAP, etc.). It is a column-oriented database. MonetDB supports multiple data models simultaneously. MonetDB architecture is based on the RISC-approach for database systems. MonetDB uses MonetDB Interpreter Language (MIL) to abstract internal implementation from higher-level models. To support extensibility, it supports MonetDB Extension Language (MEL), which can be used to extend the MonetDB functionality, e.g., datatypes, commands, etc.

MonetDB/X100 Zukowski et al. [2005] presented X100. A new execution engine for the MonetDB system. X100 uses in-cache vectorized processing that improves execution speed of MonetDB and overcomes its main-memory limitation. It further introduced the ColumnBM storage layer to handle large disk-based datasets using techniques of ultra lightweight compression [Zukowski et al., 2006] and cooperative scans [Zukowski et al., 2007]. The Cellular DBMS architecture gets inspiration from MonetDB/X100 and intends to adapt and integrate the best of MonetDB/X100 concepts with its unique cellular architecture in the future.

C-Store C-Store [Abadi et al., 2008; Stonebraker et al., 2005] is an open-source read-optimized relational DBMS. It is a column-oriented DBMS. Its architecture is

[2] "MonetDB", `http://www.monetdb.org/`, Accessed: 21-06-2011

designed to reduce the number of disk accesses per query. It proposed the use of two different stores within same DBMS, i.e., read-optimized and write-optimized stores, from which the write-optimized store operates in main-memory fashion.

Brighthouse Brighthouse [Ślęzak et al., 2008] is a column-oriented data warehouse with the concept of a meta-data layer called Knowledge Grid. The knowledge grid is used as an alternative to classical indexes. In use of meta-data, the Cellular DBMS architecture finds some similarity with the concept of the Brighthouse; however, they are different. The Cellular DBMS architecture allows the use of common indexes. Meta-data in the Cellular DBMS architecture is not used as an alternative to classical indexes. For database functionality, the Brighthouse uses MySQL's pluggable storage engine platform, whereas the Cellular DBMS architecture can be developed using any customizable embedded database. The Cellular DBMS architecture also gets inspiration from Brighthouse and intends to adapt and integrate the best of the Brighthouse concepts within its unique cellular architecture in the future. One such an important feature of the Brighthouse is the selection of different compression algorithms for different Data Packs, based on the data types and regularities automatically observed over data.

4.5.2 ECOS in comparison with other self-tuning solutions

An automated tuning system (ATS) [Hellerstein, 1997] is a feedback control mechanism that automatically adjusts the tuning knobs using the defined tuning policies, according to the monitoring statistics. ECOS also works in similar fashion as suggested in ATS. ECOS also monitors and adjust storage structures with changing data management needs. Malik et al. [2008] suggested the benefit of online physical design techniques and proposed an online vertical partitioning technique for physical design tuning. Bruno and Chaudhuri [2007b] presented an online algorithm for index tuning. The QUIET tool by Sattler et al. [2003] and COLT self-tuning framework by Schnaitter et al. [2006] are also online self-tuning approaches. Similarly, ECOS also operates in online fashion.

Automated physical design research focuses on finding the best physical design structure for running workload, e.g., indexes, materialized views, partitioning, clustering, and views [Agrawal et al., 2006]. Existing automated physical design tools assume the workload as a set of SQL statements [Agrawal et al., 2006; Sattler et al., 2003]. Most of the existing physical design tools require a synthetic workload from a DBA to select the appropriate indexes and materialized views, which is assumed

to be similar to the real workload [Bruno and Chaudhuri, 2007a]. These tools use query optimizer to identify the appropriate physical design selection from various proposed candidate designs [Papadomanolakis et al., 2007; Sattler et al., 2003]. The size of proposed candidate designs could grow large, which requires pruning using the heuristic approach. Dash and Ailamaki [2010] questioned the quality guarantee for the solution achieved from the heuristically prunned candidate designs and presented CoPhy, a physical design tool developed exploiting the technique of the combinatorial optimization problem, as a solution. Furthermore, the cost of using a query optimizer for automated physical design is huge. Papadomanolakis et al. [2007] mentioned that for index selection algorithms on average 90% of running time is spent in the query optimizer. Bruno and Nehme [2008] also accepted the existence of this overhead and presented a solution of parametric query optimization to reduce the number of calls for query optimizer. ECOS also performs automated physical design, but at the different level, i.e., at the storage manager level. It does not rely on a query optimizer. Furthermore, ECOS design is motivated from the idea of exploring new architectures for developing self-tuning DBMS instead of developing techniques to self-tune the existing ones.

There are several different self-tuning based solutions for commercial DBMS, such as AutoAdmin [Chaudhuri and Narasayya, 2007], Oracle automatic SQL tuning [Dageville et al., 2004], and DB2 design advisor [Zilio et al., 2004]. DB2 design advisor is a physical database design tool to recommend indexes, materialized views, partitioning, and clustering for a given workload [Zilio et al., 2004]. Oracle automatic SQL tuning is an integrated solution with Oracle query optimizer. Using query optimizer it analyzes the SQL statements and procedures and gives the recommendations for the tuning [Dageville et al., 2004].

During late 80's and early 90's, the active database management system (ADBMS) was a hot research topic in the database domain [Consortium, 1996; McCarthy and Dayal, 1989; Paton and Díaz, 1999]. The ADBMS can be seen as a classical variant for existing self-tuning DBMS with a focus on automating different functionality rather than tuning. Many concepts introduced by ADBMS, such as event-condition-action (ECA)-rule model and event driven execution of functionalities has their counterparts in existing self-tuning technologies with different names. Our presented approach of the evolution path is quite similar in concept to the concept of ECA-rule model in ADBMS. However, our realization of the evolution path using different software engineering approaches and its usage for self-tuning data management is different.

4.6 Summary

In this chapter, we presented ECOS; a customizable self-tuning storage manager realized using the concepts from the Cellular DBMS architecture. ECOS allows customization of each table using five different variations of DSM schemes. It also allows customization of storage structures for each column in a table. It uses evolving hierarchically-organized storage structures to realize autonomy for each column. We also introduced the concept of the evolution path, which allows us to reduce the human intervention for long-term DBMS maintenance.

5 The prototype implementation: Problems faced and lessons learned

> The success of research is more and more measured in terms of product impact, and for an academic idea to be intriguing to product developers, major prototype implementation and extensive experimentation is often required. With commercial database system being such a highly complex target, this kind of work becomes less and less rewarding and all too often exceeds the available resources in a university environment.
>
> Surajit Chaudhuri and Gerhard Weikum
> *Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System, VLDB 2000*

This chapter reports our experience with the Cellular DBMS prototype implementation. It discusses the details about the problems that we faced and our different design decisions to solve those problems. It presents the Cellular DBMS prototype implementation with detailed discussion from the software engineering perspective. It also provides the insight about the source code implementation of the evaluation mechanism in the Cellular DBMS prototype.

5.1 Our database system implementation experience

The development of the Cellular DBMS prototype started with the development of the FAME-DBMS prototype[1] [Rosenmüller et al., 2008]. The target for the FAME-DBMS project was data management for low-end embedded systems. Development of data management for embedded systems is different from the traditional data management system, because of resource limitations in embedded devices, such as less memory, low processing power, and limited or sometimes no persistent storage

[1] "Fame-DBMS project", `http://fame-dbms.org/`, Accessed: 21-06-2011

space. Battery time is also an important resource for embedded devices operating on batteries.

The first important decision for developing a data management system for embedded systems is the selection of a programming language. Assembly and C languages are the two most successful languages in this domain. They provide the best performance, but if the size of the project is expected to grow large, they may become a management nightmare. However, the largest data management solution for an embedded system should be within the limited size, because of the memory limitations, which should be easily manageable with Assembly or C language. For the Cellular DBMS prototype, we preferred to use C++ language, estimating the maximum expected size of the software to grow large. Our programming language design decision was also biased because of the availability of FOP in C++, i.e., FeatureC++ [Apel et al., 2005]. If we had the tool available to implement our DBMS prototype using FOP in C, then we might have gone towards C for better performance, and the compatibility of small database variants with embedded systems. Another important decision was about the memory allocation, that either we want to do static or dynamic memory allocation. Many embedded platforms do not support dynamic memory allocation, whereas if few of them do support it, most of them recommend not using it. In contrast, use of dynamic memory allocation for a self-tuning database is imminent. We observed that the programming language selection and the memory allocation decision also have an impact on the binary size and execution footprint of a database.

According to our Cellular DBMS prototype development experience, for a data management solution, we argue that the performance is directly proportional to the resources, i.e., to increase the performance, we have to use more resources. In contrast, if we want to preserve resources, there will be a compromise on the performance. Furthermore, three important resources of CPU cycles, memory, and persistent storage also have a direct relation with each other in terms of their access and utilization. If we want to reduce the access to persistent storage because of high access cost, we have to use more CPU cycles and memory. If we want to reduce the access to memory because of high memory latency, we have to use more CPU cycles. Our research observations are in agreement with the observations documented by Gray, Putzolu, and Graefe [Graefe, 2008; Gray and Graefe, 1997; Gray and Putzolu, 1987]. Considering the high growth rate of processor speed in comparison with memory and persistent storage [Patterson et al., 1997], it is essential to change focus of database development towards better CPU utilization. Many database

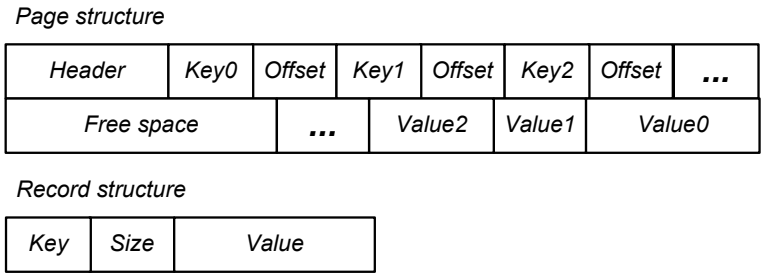

Figure 5.1: Page and record structures in the Cellular DBMS prototype.

researchers are already focused in this research direction, such as [Ailamaki et al., 1999], [Boncz et al., 2005], [Zukowski et al., 2006], and [Ailamaki et al., 2002] are few examples.

We also argue that self-tuning of a data management solution is inversely proportional to the customization, i.e., if we remove all unneeded functionalities during our initial customization, we confine our self-tuning options to a minimum. In contrast, the more functionalities we have, less customized is our data management solution, which means we have more options to self-tune our DBMS. However, in Chapter 3, we have explained in detail the negative impact of a high number of functionalities on a DBMS performance.

According to our above-mentioned observations during the Cellular DBMS prototype development, we found that the term customization should be further classified as hard and soft customization to increase the understandability. By hard customization, we mean the removal of features from a variant. By soft customization, we mean the selection of a required minimal feature at initial design time, such that the unused features are available in the binary footprint and can be used when required. Efficient soft customization requires an implementation, which ensures that when unused, the unused functionalities should have minimal effect on the used functionalities. Moreover, unused functionalities should be usable with minimal overhead. According to our explanation in Chapter 3, the generation of variants through feature selection is an example of hard customization, whereas according to our explanation in Chapter 4, the selection of a small minimal optimal storage structure (such as sorted array) is an example of soft customization. In the Cellular DBMS prototype implementation, we attempted to find a right balance between customization and self-tuning through suggesting precisely what to customize and how to self-tune.

Storage model selection is an important design decision for a DBMS implementation. This design decision has a direct impact on the design of page and record structures of a DBMS. Page and record structures for the Cellular DBMS prototype implementation are depicted in Figure 5.1. Both page and record structures store key/value pairs of data, where keys are fixed sized and values are variable sized. For a simple data management scenario, a key/value pair could be a good enough solution. For an online transaction processing (OLTP) specific solution, the NSM is considered more suitable, whereas for an online analytical processing (OLAP) specific solution, the DSM is considered more suitable [Stonebraker and Cetintemel, 2005]. We used the DSM because of its suitability for self-tuning DBMS implementation. Detailed discussion about our decision to use the DSM is provided in Chapter 4. We observed that the data management need of embedded devices does not need complex data and index storage structures, such as the B+-Tree. Instead, simple data storage structures, such as array, list, queue, and stack can be equally efficient in most of the scenarios. Moreover, after several experimentations with different storage structures, we arrived at the conclusion that any storage structure can be optimized for either read-optimized workload or write-optimized workload at a time, but not for both.

The Cellular DBMS SPL is capable to generate the variants for different types of databases. Our current implementation is capable of generating embedded database, column-oriented storage manager, key/value store, relational database, in-memory database, and persistent database. Therefore, the Cellular DBMS SPL is too generalized. Our experience with this SPL suggests that it is not a practical approach for a real or commercial DBMS SPL. Rather our experience for SPL is in agreement with what Chaudhuri and Weikum [2000] referred as "universality trap". We suggest that DBMS SPL should be specific for certain type of DBMS, such as DBMS SPL for an in-memory column-oriented relational storage manager. The reason for this recommendation is the high difference in the requirements and implementation of different types of databases. For example, algorithm and implementation techniques that are suitable for in-memory database and persistent database are different.

5.2 Prototype implementation details

The Cellular DBMS prototype is implemented using the software product line approach by considering benefits that we discussed in detail in Chapter 3. We used the FOP to realize the Cellular DBMS SPL. The source code of the Cellular DBMS

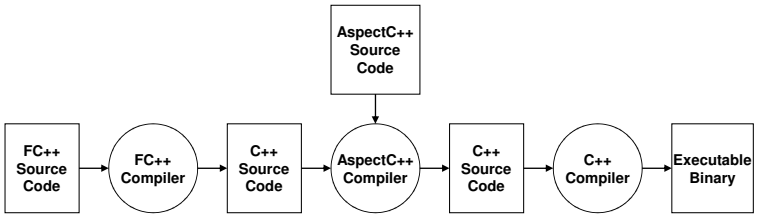

Figure 5.2: Source code transformation.

prototype is written in FeatureC++[2]. "FeatureC++ is a C++ language extension to support FOP" [Apel et al., 2005]. Autonomy is implemented in the Cellular DBMS prototype using the AOP. The source code for autonomy is written using the AspectC++[3], which is a set of C++ language extensions to facilitate AOP with C++. The reason behind using the AOP for autonomy implementation is to keep the monitoring functionality source code separate from other functionalities, which otherwise gets tangled and scattered across the source code of other functionalities making it difficult in long-term to separate it. The code transformation model for our prototype implementation using the FeatureC++, the AspectC++, and the C++ compiler is shown in Figure 5.2. To the best of our knowledge, the Cellular DBMS prototype is the first relational column-oriented storage manager implementation using an SPL approach and FOP.

The storage structures that we have implemented in our Cellular DBMS prototype are sorted array, heap array, sorted list, heap list, B+-Tree, HLC SL, and HLC B+-Tree. Details about these storage structures are provided in Section 4.2.2. We have implemented the DSM storage model and its four variations, i.e., KDSM, MDSM, DMDSM, and VDMDSM. Details about the DSM and its variations are provided in Section 4.2.1. The Cellular DBMS SPL can generate both in-memory database variants as well as persistent storage database variants.

The feature model of the Cellular DBMS prototype is shown in Figure 5.3. "A feature model (a.k.a. domain model or product line variability model) describes the features of a domain or SPL and their relationships" Kästner et al. [2009]. The feature model shows an important characteristic for every feature, i.e., either it is mandatory or optional. Mandatory are features that are always part of a variant, whereas optional features can be removed from a variant. The customizability of an SPL is dependent on the number of optional features it contains. A feature model can also show a relationship among features. In the Cellular DBMS proto-

[2] "FeatureC++", `http://wwwiti.cs.uni-magdeburg.de/iti_db/fcc/`, Accessed: 21-06-2011
[3] "AspectC++", `http://www.aspectc.org/`, Accessed: 21-06-2011

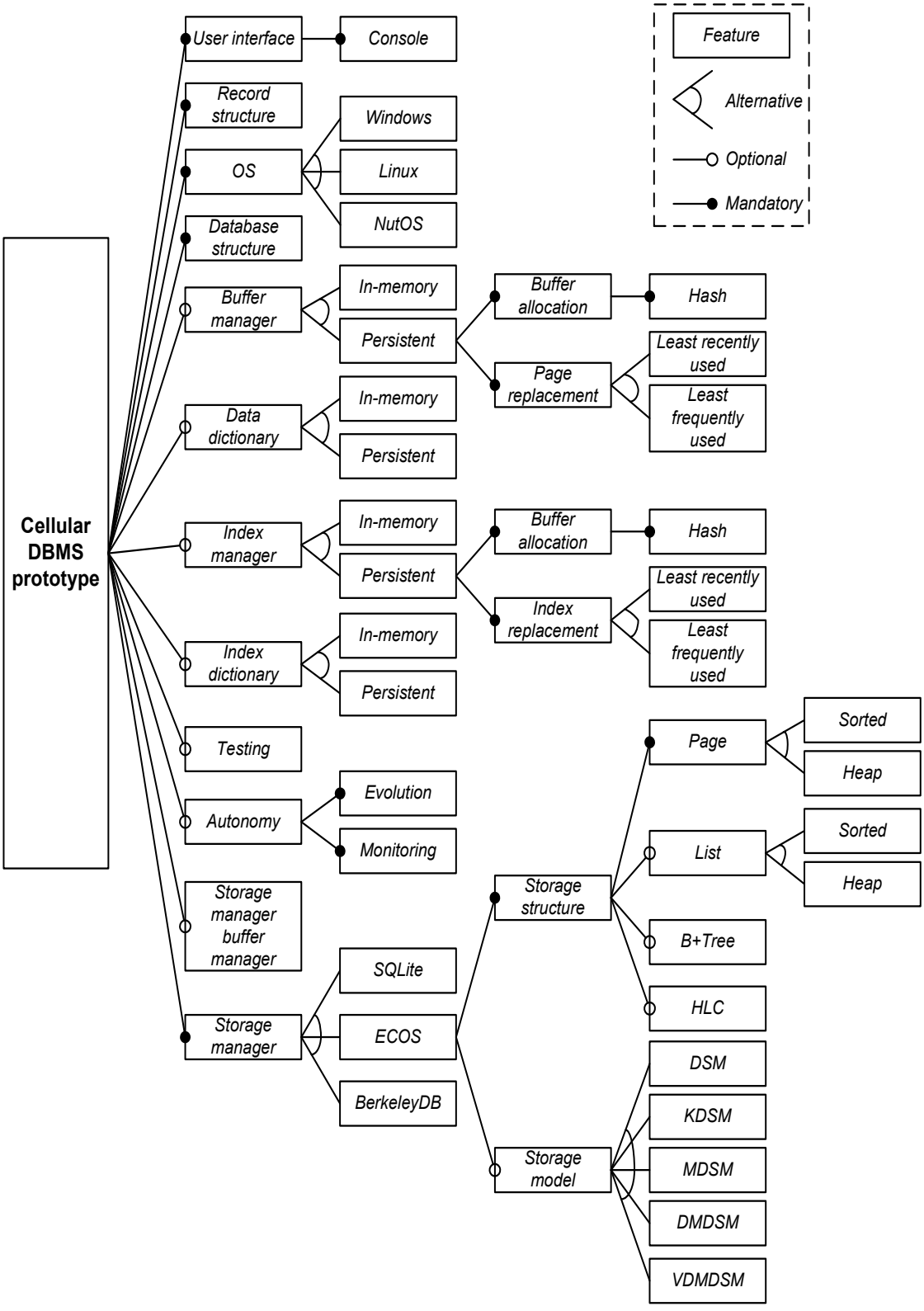

Figure 5.3: The Cellular DBMS prototype feature model.

type feature model, we only used one relationship type that is called alternative relation. Alternative relation means that when two or more features have an alternative relationship, only one of them can be selected in a variant at a time. In Figure 5.3, Record structure is an example of a mandatory feature, Testing is an optional feature, and Sorted and Heap are alternative features to each other.

Figure 5.3 shows all of the high-level features for the Cellular DBMS SPL. We avoided the details about small features to keep the feature model simple. In the Cellular DBMS prototype implementation, we have approximately 151 features. The snippet of the configuration file listing all the features of the Cellular DBMS prototype is provided in Appendix A. However, because of many dependencies among features, we are able to generate only 34 functional database variants using this feature model. The 17 in-memory database variants that we used for discussion and evaluation in this thesis are listed with details in Table 5.1. The configuration file in Appendix A is configured to generate the variant 8 from the variants listed in Table 5.1.

The smallest possible database variants that can be generated from the Cellular DBMS SPL are variants 1 and 2 as listed in Table 5.1. Both, variant 1 and 2, use a single in-memory page as a database. Variant 1 stores sorted data according to key-order, where as variant 2 stored data according to insertion-order. The minimal feature model for database variant 1 is presented in Figure 5.4.

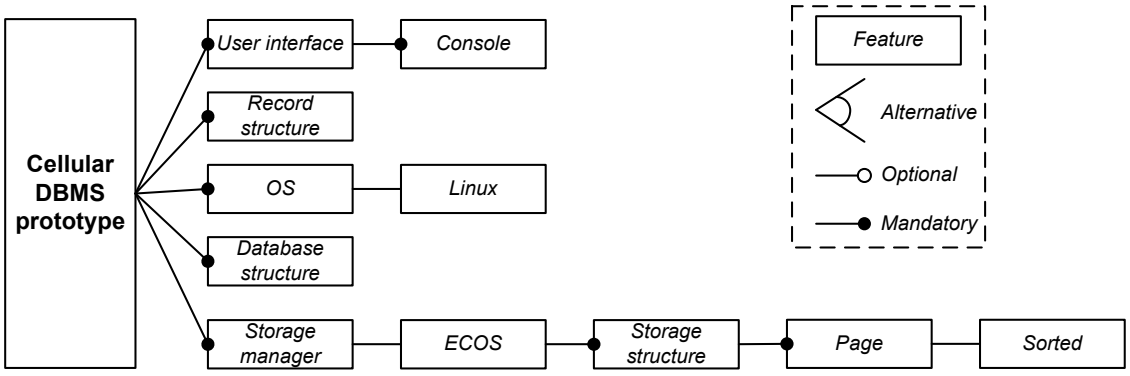

Figure 5.4: The Cellular DBMS prototype minimal variant feature model.

We used the feature derivative approach to handle the optional feature dependencies [Liu et al., 2006]. The optional feature means the functionality that can be removed from the database. The optional feature problem is well known in the SPL domain [Kästner et al., 2009]. It emerges from the existence of optional features that are dependent on each other, i.e., if any one of those dependent optional features is to be selected, then all other dependent optional features should also be

selected or vice versa none of them is selected. For example, in Figure 5.3, it can be observed that both B+-Tree and Index manager are optional in the Cellular DBMS SPL. However, the B+-Tree cannot work without the Index manager feature. If we generate a variant by selecting any one of them then the variant will not compile as the references to the Index manager in B+-Tree will not be available.

A feature derivative is used to separate the dependent source code from all dependent optional features [Liu et al., 2006]. Feature derivative is an additional module created using the dependent code from optional features refactored as a separate module. Feature derivative for dependent optional features is only included in the generation process of source code if and only if all dependent optional features are selected. A detailed discussion about the optional feature problem and use of the feature derivative approach as a solution can be found in work from Liu et al. [2006] and Kästner et al. [2009]. We found 152 feature derivatives for 151 features in the Cellular DBMS prototype implementation, which is quite high. The reason for this high number of feature derivatives is the granularity of features. For example, during initial design, we made Put, Get, and Delete functionality as a separate feature. However, with introduction of relatively complex storage structures, such as B+-Tree, these fine-grained features became useless because they were always needed for all variants. These fine-grained features still exist in our prototype implementation and can be seen in the configuration file presented in Appendix A. We concluded according to our observation that the finer is the granularity of features the higher is the number of feature derivatives. According to our experience, we suggest using FOP with coarser-grained features to avoid the maintainability headaches, which is also compatible with the results presented by Liebig et al. [2010] and problems reported by Kästner et al. [2008].

Table 5.1: Statistics and details for the Cellular DBMS prototype implementation variants.

Variant	Binary size (KB)	Lines of code (LOC)	No. Of features	Tuning knobs	Database type	Storage model	Storage structure
1	103	5665	45	Page size	key/value data store	None	Sorted array
2	82	3795	42	Page size	key/value data store	None	Heap array

Continued on next page...

Variant	Binary size (KB)	Lines of code (LOC)	No. Of features	Tuning knobs	Database type	Storage model	Storage structure
3	123	8027	58	Page size Max. page buffer size Max. pages	key/value data store	None	Sorted list
4	92	4845	53	Page size Max. page buffer size Max. pages	key/value data store	None	Heap list
5	165	9945	53	Page size Max. page buffer size Max. pages Max. index buffer size Max. index nodes Max. index node elements	key/value data store	None	B+-Tree
6	188	13078	76	Page size Max. page buffer size Max. pages Max. storage manager buffer size Max. storage manager nodes Max. HLC index buffer size Max. HLC index nodes Max. HLC index node elements	key/value data store	None	HLC SL
7	217	15020	78	Page size Max. page buffer size Max. pages Max. index buffer size Max. index nodes Max. index node elements Max. storage manager buffer size Max. storage manager nodes Max. HLC index buffer size Max. HLC index nodes Max. HLC index node elements	key/value data store	None	HLC B+-Tree
8	263	17563	101	Page size Max. page buffer size Max. pages Max. storage manager buffer size Max. storage manager nodes Max. HLC index buffer size Max. HLC index nodes Max. HLC index node elements Max. columns	Relational data store	DSM	HLC SL

Continued on next page...

Variant	Binary size (KB)	Lines of code (LOC)	No. Of features	Tuning knobs	Database type	Storage model	Storage structure
9	293	19505	104	Page size Max. page buffer size Max. pages Max. index buffer size Max. index nodes Max. index node elements Max. storage manager buffer size Max. storage manager nodes Max. HLC index buffer size Max. HLC index nodes Max. HLC index node elements Max. columns	Relational data store	DSM	HLC B+-Tree
10	263	17525	101	Page size Max. page buffer size Max. pages Max. storage manager buffer size Max. storage manager nodes Max. HLC index buffer size Max. HLC index nodes Max. HLC index node elements Max. columns	Relational data store	KDSM	HLC SL
11	293	19467	104	Page size Max. page buffer size Max. pages Max. index buffer size Max. index nodes Max. index node elements Max. storage manager buffer size Max. storage manager nodes Max. HLC index buffer size Max. HLC index nodes Max. HLC index node elements Max. columns	Relational data store	KDSM	HLC B+-Tree
12	262	17548	101	Page size Max. page buffer size Max. pages Max. storage manager buffer size Max. storage manager nodes Max. HLC index buffer size Max. HLC index nodes Max. HLC index node elements Max. columns	Relational data store	MDSM	HLC SL

Continued on next page...

Variant	Binary size (KB)	Lines of code (LOC)	No. Of features	Tuning knobs	Database type	Storage model	Storage structure
13	291	19490	104	Page size Max. page buffer size Max. pages Max. index buffer size Max. index nodes Max. index node elements Max. storage manager buffer size Max. storage manager nodes Max. HLC index buffer size Max. HLC index nodes Max. HLC index node elements Max. columns	Relational data store	MDSM	HLC B+-Tree
14	266	17711	101	Page size Max. page buffer size Max. pages Max. storage manager buffer size Max. storage manager nodes Max. HLC index buffer size Max. HLC index nodes Max. HLC index node elements Max. columns	Relational data store	DMDSM	HLC SL
15	296	19653	104	Page size Max. page buffer size Max. pages Max. index buffer size Max. index nodes Max. index node elements Max. storage manager buffer size Max. storage manager nodes Max. HLC index buffer size Max. HLC index nodes Max. HLC index node elements Max. columns	Relational data store	DMDSM	HLC B+-Tree
16	294	19220	106	Page size Max. page buffer size Max. pages Max. storage manager buffer size Max. storage manager nodes Max. HLC index buffer size Max. HLC index nodes Max. HLC index node elements Max. columns	Relational data store	VDMDSM	HLC SL

Continued on next page...

Variant	Binary size (KB)	Lines of code (LOC)	No. Of features	Tuning knobs	Database type	Storage model	Storage structure
17	294	21162	104	Page size Max. page buffer size Max. pages Max. index buffer size Max. index nodes Max. index node elements Max. storage manager buffer size Max. storage manager nodes Max. HLC index buffer size Max. HLC index nodes Max. HLC index node elements Max. columns	Relational data store	VDMDSM	HLC B+-Tree

Liu et al. [2006] introduced the term of the higher order derivatives (HOD) for the feature derivatives that are dependent on more than two optional features. In the Cellular DBMS prototype, we found forty two HOD. Table 5.2 shows the statistics for the number of features and the number of HOD in our prototype source code. We conclude according to our observation that the hierarchy of HOD grows deeper as the number of dependent optional features grows. Moreover, the deeper grows the hierarchy of HOD the more difficult it becomes to manage the source code.

We also conclude from our experience that the use of preprocessor directives is unavoidable even with the use of the FOP and the AOP. In our prototype implementation, we were constrained to use preprocessor directives sixty eight times. All of them are related to platform and compiler. The four preprocessor directives that we used are: #ifdef LINUX, #ifdef btnode3, #ifdef MAKEGXX, #ifdef __cplusplus, and #ifdef __GNUG__. Thirty five of preprocessor directives appeared in feature derivatives. Forty five of directives were used in the testing source code. Seven of directives were used in the user interface source code. According to our observation, we recommend to use preprocessor directives for platform and compiler specific customizations rather than feature refinements of FOP. A refinement means an addition of new elements (such as method or variable) to a class or extending an existing element, and its use for platform and compiler specific customizations will eventually result in many smaller feature derivatives. We also observed that the time for a cross platform DBMS development increases because of the compatibility issues. It is a difficult task to guarantee 100% similarity in DBMS behavior across different platforms and compilers.

Table 5.2: Feature derivatives and higher order feature derivatives for important features in the Cellular DBMS prototype.

Feature	**Feature derivatives**	**Higher order feature derivatives**
Main.UserInterface	10	2
Autonomy.Evolve	15	8
Test	16	2
HPage	4	
Page	4	
SortedList	2	
HeapList	2	
B+-Tree	9	
HLComposite	9	
DSM.*	13	6
KDSM.*	13	6
MDSM.*	13	6
DMDSM.*	13	6
VDMDSM.*	13	6
BufferManager.*	5	

To further analyze the Cellular DBMS SPL source code, we used the CLOC tool[4] to count the LOC. Our Cellular DBMS SPL consists of 42953 LOC excluding comments and whitespaces. Approximately, 23.48% of LOC resides in feature derivatives, i.e., 10086 LOC. Our result is quite close to the evaluation results provided by Liebig et al. [2010] for the analysis of the variability in forty preprocessor-based SPL, which showed that approximately on average 23% of code-base in a software project is variable. It can be observed from Table 5.1 that the largest in-memory database variant in terms of LOC is variant 17, which consists of 21162 LOC, i.e., it only uses approximately 50% of the SPL source code.

According to Table 5.1, the number of features in variant increases from variant 1 to variant 17, i.e., variant 1 is the smallest and variant 17 is the largest database variant. In Figure 5.5, we present the effect of an increase in the number of feature on the LOC and binary size for database variants. It can be observed that LOC and binary size increases with the increase in the number of features. Vice versa our results are also agreeable with the results presented by Liebig et al. [2010], which

[4]CLOC (Count Lines of Code) tool, `http://cloc.sourceforge.net/`, Accessed: 21-06-2011

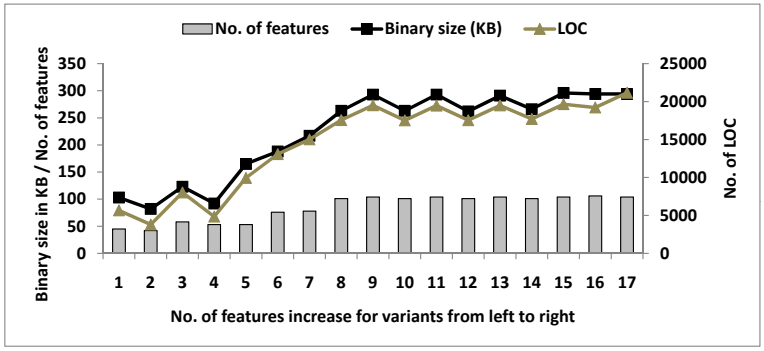

Figure 5.5: Increase in features also increases the LOC and the binary size.

shows that variability of a software system increases with the increase in its source code size. Similarly, in Figure 5.6, we show the effect of an increase in the number of features on the number of tuning knobs. It can be observed that the number of tuning knobs increase with the increase in the number of features.

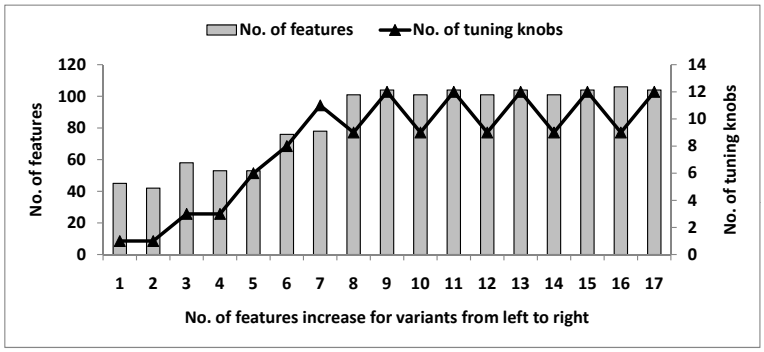

Figure 5.6: Increase in features also increases the tuning knobs.

5.3 Implementation of evolution mechanism

In this section, we explain, how we implemented the evolution mechanism for our ECOS implementation in our Cellular DBMS prototype. Our aim for evolution mechanism implementation was to keep the overheads to be negligible, whereas at the same time we wanted to ensure that the implementation of evolution should not get tightly coupled with standard storage manager implementation. For this purpose, we used the innovative software engineering technique of the AOP to ensure that the evolution behavior can be added or removed from the storage manager without affecting the other storage manager functionalities.

5.3.1 Monitoring functionality implementation

The most important functionality of the evolution mechanism is the monitoring functionality. ECOS monitors existing storage structures to gather the heredity information and to observe the data management operation events (see Section 4.3 for details of the heredity information and events). The code snippet 5.1 of our monitoring functionality is implemented as an aspect (a modular way to separate the common code that otherwise is part of different software components) using the AspectC++ language constructs.

In the code snippet 5.1, the code between line numbers 6 to 11 is an advice code. An advice is used to specify the additional code that could be executed before, after, or at both points (i.e., around) during the flow of a program. For example, on the line number 6 in the code snippet 5.1, the *before* keyword ensures that the advice is executed before the execution of the *ICPutData* function.

5.3.2 Trace functionality implementation

Another important functionality of evolution implementation is the trace functionality, which executes before the execution of data management operation and stores the heredity information, such as column information and record details. The advice defined at the line number 6 in the code snippet 5.1 is a sample trace code, in which we gather the statistics for all *ICPutData* function executions, which for presented sample source code include taking reference to the involved column and record objects. We use this information to call *ICPutData* again, if it fails to execute successfully.

5.3.3 Analysis and fixing functionality implementation

The advice between line numbers 14 to 38 in the code snippet 5.1 define the code that analyzes the execution of different data management functions. It also executes the fixing code if some problem is identified. For example, advice at the line number 14 in the code snippet 5.1 checks the execution result of the *PutData* function of the *Page* implementation class. If some problem is identified, such as *NO_SPACE* at the line number 22 in the code snippet 5.1, it executes the code that analyzes the problem according to the recorded trace data and fixes the problem. All functions used in advices in the code snippet 5.1 are defined in the *Autonom* class. The code snippet 5.2 is the implementation of *Autonom* class.

Listing 5.1: Monitoring implementation code snippet

```
aspect Monitor {
  Autonom a;
  MSG _msgid;

  /*Monitoring for tracing*/
  advice execution("MSG Composite::ICPutData(...)") : before() {
     Composite *c;
     _msgid = (MSG) * tjp->result();
     c = tjp->that();
     a.TraceICPutData((RECORD*) tjp->arg(0), (COLUMN*) tjp->
        arg(1));
  }

  /*Monitoring for possible events, analysis, and fixings*/
  advice execution("MSG Page::PutData(...)") : after() {
    Page *pg;
    if (tjp->result() != NULL) {
      _msgid = (MSG) * tjp->result();
      switch (_msgid) {
        case SUCCESS:
          //Result is SUCCESS
          break;
        case NO_SPACE:
          //Result is NO_SPACE
          pg = tjp->that();
          _msgid = a.AnaFixCheckStorageNoSpace(pg);
          *tjp->result() = _msgid;
          if (_msgid == SUCCESS) {
            a.TraceReset();
          }
          break;
        case NOT_FOUND:
          ...
        default:
          //Unexpected result
          break;
      }
    }
  }
  ...
}
```

Listing 5.2: Autonom class implementation code snippet

```
class Autonom {
  public:
  //Evolve class implements the evolving functionality
  Evolve evo;
  //Trace function for PutData method
  void TraceICPutData(RECORD* _r, COLUMN* _c);
  //Analysis and fixing functions
  MSG AnaFixCheckStorageNoSpace(Page *p);
  ...
};

/*Trace functions*/
void Autonom::TraceICPutData(RECORD* _r, COLUMN* _c) {
  this->evo.r = _r;
  this->evo.c = _c;
  this->evo.isbyval = false;
}

/*Analysis and fixation functions*/
MSG Autonom::AnaFixCheckStorageNoSpace(Page *p) {
  //This is for evolving from Sorted Array to Sorted List
  this->evo._msgid = NO_SPACE;
  //Evolution AnaFix function implements the analysis and fixing
  return this->evo.AnaFix();
}
```

It can be observed from the code snippet 5.2 that the *Autonom* class contains the implementation of trace functions and uses the *Evolve* class to execute the analysis and fixing. However, we use the code snippet 5.1 and 5.2 for a twofold purpose. As the first purpose during analysis we identify, when to evolve the existing storage structure. For example, as shown at the line number 22 in the code snippet 5.1, each case triggers an event for possible evolution of existing storage structure. Furthermore, as the second purpose we also identify, either the existing storage structure should be evolved into new storage structure or not. The *AnaFix* function at the line number 25 in the code snippet 5.2 contains the functionality to take this decision.

The code snippet 5.3 presents our evolution code, in which the function *EvolveColumnIM()* at the line number 1 evolves a sorted array storage structure into a sorted list storage structure. The $c->im$ object refers to a sorted array storage structure and the $c->sl$ refers to a newly instantiated sorted list storage structure. Each storage structure implements an *Evolver* function, such as the one used at the line number 16 in the code snippet 5.3.

The *Evolver* function contains the implementation that makes the existing storage structure, which is provided as an argument, an integral component of the newly instantiated storage structure. For example, the *Evolver* function at the line number 16 in the code snippet 5.3 takes a sorted array as an argument and makes it an integral part of the sorted list. For better understanding, the implementation of the *Evolver* function of the sorted list is also provided at the line number 23 in the code snippet 5.3. The *Evolver* function at the line number 23 in the code snippet 5.3 instantiates a new sorted array and distributes the data of the existing sorted array among them equally, then it makes both new and old sorted arrays a part of the new sorted list. It is a naive implementation that we used to demonstrate the concept, however, the *Evolver* function is an important code fragment. The implementation of an *Evolver* function identifies the associated overhead for an evolution. The interface provided to the end-user or external application by ECOS is simple and consistent. Which storage structure is in use by the column?, when it is evolved?, all these aspects are hidden. A sample code snippet to give an insight for ECOS interface is provided as the code snippet 5.4.

5.4 Summary

In this chapter, we presented our experiences with the database implementation. We outlined problems we faced and solutions we adopted. We provided details about

Listing 5.3: Evolution implementation code snippet

```
MSG Evolve::EvolveColumnIM() {
  //We evolve Sorted array to Sorted list
  //or Heap Array to Heap List
  //This column should be traced during tracing
  if (this->c == NULL) { return UNRESOLVED; }

  //First we change column type to sorted list
  this->c->columntype = SL;
  //Instantiate new Sorted List
  this->c->sl = new StorageManager();
  _msgid = c->sl->CreateDatabase(this->c->im->database);
  if (_msgid != SUCCESS) return _msgid;

  //Now evolve from Array to List
  //Such that existing Array will become an integral unit of ↩
     List
  _msgid = this->c->sl->Evolver(this->c->im);
  if (_msgid != SUCCESS) return _msgid;

  //Now after evolution, redo last buffered operation
  return this->EvolveColumnIMPutData();
}

MSG StorageManager::Evolver(Page* _page) {
  MSG _msgid = pb->EvolvePB(_page);
  if (_msgid != SUCCESS) return _msgid;
  this->tuplecount += _page->CountTuples();
  dd->SetStartPage(_page->GetID());
  dd->SetEndPage(_page->GetID());
  return SUCCESS;
}

MSG PageBuffer::EvolvePB(Page* page) {
  tmpPID++;
  pg[tmpPID - 1] = page;
  pg[tmpPID - 1]->SetID(tmpPID);
  ++usedPageCount;
  fOnUsedPageCountChanged(evenSink, usedPageCount);
  return SUCCESS;
}
```

Listing 5.4: ECOS interface code snippet

```
RECORD* crecords =
(RECORD *) malloc(sizeof (RECORD) * <No. of columns>); ...
crecords[<index>].key = <key>;
crecords[<index>].columnindex = <column index>;
crecords[<index>].size = <No. of bytes for value>;
crecords[<index>].value =
(cbyte*) malloc(sizeof(cbyte) * <No. of bytes for value>); ...
_msgid = cell.GetDataNext(crecords); //Scan ...
_msgid = cell.GetData(crecords);//Get record ...
_msgid = cell.PutData(crecords);//Put record ...
_msgid = cell.DeleteData(crecords);//Delete record ...
```

our Cellular DBMS prototype implementation and presented statistics that gave many insights about the impact of an increase in features on the complexity of the implementation. We concluded that increase in the number of features also increase the number of tuning knobs, LOC, binary size, feature derivatives, and HOD. We showed that even in the presence of FOP, the use of preprocessor directives for cross platform DBMS development is imminent. We recommended to make use of FOP with coarser-grained features and to avoid making a DBMS SPL general for all data management needs.

6 Evaluation

This chapter shares material with the FIT'10 paper "Using Evolving Storage Structures for Data Storage" [ur Rahman, 2010] and the BNCOD'11 paper "ECOS: Evolutionary Column-Oriented Storage" [ur Rahman et al., 2011].

This chapter presents details about the micro benchmark that we used to perform the evaluation of the Cellular DBMS prototype. It also presents and discusses the evaluation results to assess the impact of unused functionalities on a DBMS performance, the performance and resource consumption comparison of different storage structures for different data sizes, the comparison of different DSM based schemes, and most importantly the behavior and related performance improvement from hierarchically-organized storage structures.

6.1 Micro benchmark details

For evaluation of the Cellular DBMS prototype, we set up a micro benchmark with repeated insertion, selection, and deletion of data using API based access method. The data contain keys in ascending, descending, and random order, which also represents their insertion, selection, and deletion order in a database. For different columns, the number of records and the distinct data count (cardinality) is kept different to assess the impact of change in data size using ECOS. We defined seven columns with two unique non-null columns and three columns with varying number of NULL values. We used one of two unique non-null columns as a primary key. We used three different widths for columns, i.e., 16, 85, and 4096 bytes to assess the impact of tuple width on performance of different DSM based schemes. All storage structures used in evaluation operate in main-memory. For the Cellular DBMS prototype evaluation, we used CPU cycles and heap memory as resources. One reason for selecting these parameters is that we evaluated only in-memory database variants. Another reason for selecting these parameters is the change in bottlenecks.

In the last two decades, the processor speed has been increasing at the much faster rate of around 60% per annum in comparison with the memory speed that increases only around 10% per year [Patterson et al., 1997]. Therefore, it is essential for a DBMS to make optimal use of increased processing power and large main memories while avoiding the overheads associated with memory latencies.

We used OpenSuse 11.2 operating on Intel(R) Core(TM)2 Duo CPU E6750 @ 2.66GHz with 4 GB of RAM. It contains two 32 KBytes 8-way set associative L1 instruction and data cache with 64-byte line size; and one 4 MB 16-way set associative L2 cache with 64-byte line size. We used Valgrind tool [Valgrind] to generate cache references and misses, and heap usage. We measured execution speed by taking the average of CPU cycles observed over multiple iterations of micro benchmark. All presented evaluation results are valid for comparison of storage structures and should not be considered as the benchmark for the Cellular DBMS performance comparison with other DBMS. For better visibility of charts, we used few abbreviations that we have listed in Table 6.1.

Table 6.1: List of abbreviations used in figures with their details.

Abbreviation	**Detail**
I	Instruction cache reference
D rd	Data (read) cache reference
D wr	Data (write) cache reference
I1	L1 Instruction cache miss
D1 rd	L1 Data (read) cache miss
D1 wr	L1 Data (write) cache miss
L2i	L2 Instruction cache miss
L2d rd	L2 Data (read) cache miss
L2d wr	L2 Data (write) cache miss

We used a micro benchmark to generate empirical results. We understand the need for empirical results using standard benchmarks, such as TPC-H, however, the existing Cellular DBMS prototype implementation only consist of the ECOS storage manager and can only be tested using a micro benchmark. Furthermore, the Cellular DBMS is a research prototype with many implementation details still in progress. We are using our best effort to provide reliable and repeatable results that can compare the Cellular DBMS with the performance of other existing commercial products; however, it is left as part of the future work.

6.2 Evaluation results

To present the impact of unused functionalities on a DBMS performance, we first used the Berkeley DB as our data management solution. As shown in Figure 6.1, 6.2, 6.3, and 6.4; RECNO, Queue, Hash, and B+-Tree represent the storage structures of the Berkeley DB that we used for evaluation. To evaluate the impact of unused functionalities on a database performance, we tested all four Berkeley DB storage structures with two Berkeley DB configurations, i.e., Default(D) and Minimal(M). The default configuration contains all features of the Berkeley DB, whereas for minimal configuration, we removed all removable features. The flags that we used to generate minimal configuration include: –disable-largefile, –disable-cryptography, –disable-hash, –disable-queue, –disable-replication, –disable-statistics, –disable-verify, –disable-partition, –disable-compression, –disable-mutexsupport, and –disable-atomicsupport. For RECNO and Queue, –disable-queue flag is not used, whereas for Hash, –disable-hash flag is not used. It can be observed that with minimal configuration of the Berkeley DB, storage structures consume much fewer resources showing better performance in comparison with the default configuration of Berkeley DB. Furthermore, it can be observed that for our micro benchmark data size of 4048 records, all simple storage structures perform better than B+-Tree.

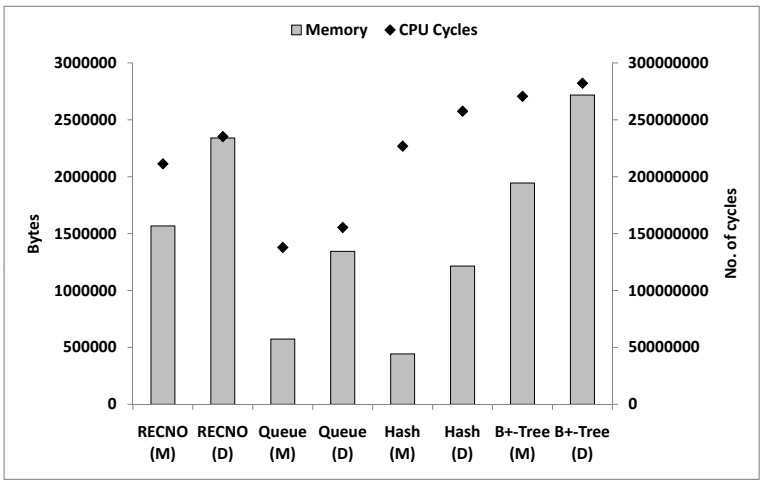

Figure 6.1: Micro benchmark results using the Berkeley DB: Minimal configurations consume less CPU cycles and memory.

We also observed the effect of an increase in data size on performance of different storage structures. We executed our benchmark for different storage structures using different number of records (i.e., single record, 4048 records, 100K records,

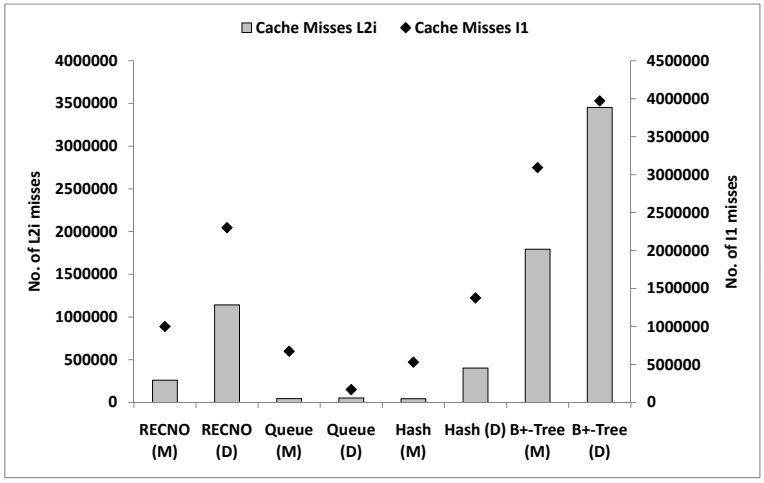

Figure 6.2: Micro benchmark results using the Berkeley DB: Minimal configurations cause less instruction cache misses.

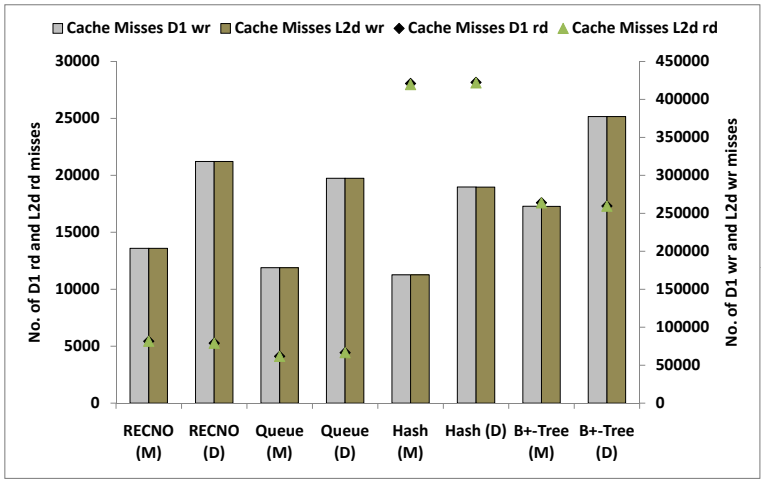

Figure 6.3: Micro benchmark results using the Berkeley DB: Minimal configurations cause less data cache write misses.

and 500K records). It can be observed in Figure 6.5 that for a single record sorted array consumes less CPU cycles in comparison with other storage structures. For 4048 records, array consumes much more CPU cycles in comparison with other storage structures therefore we omitted it in Figures 6.6, 6.7, and 6.8. In Figure 6.6, it can be observed that for 4048 records, sorted list and B+-Tree based storage structures consume a similar amount of memory. However, Figure 6.7 and 6.8 shows that B+-Tree based storage structures perform better for 100K and 500K records. According to the above observation, we suggest the performance gain and reduced resource consumption using the evolving storage structures because evolving storage structures attempt to use minimal/simple storage structures as long as possible

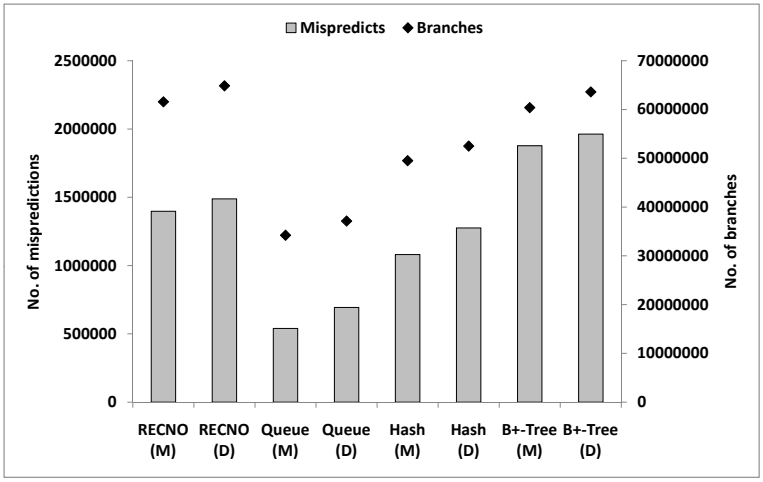

Figure 6.4: Micro benchmark results using the Berkeley DB: Minimal configurations cause fewer branches and their mispredictions.

using the definitions from evolution paths, such as a sorted array for small data management.

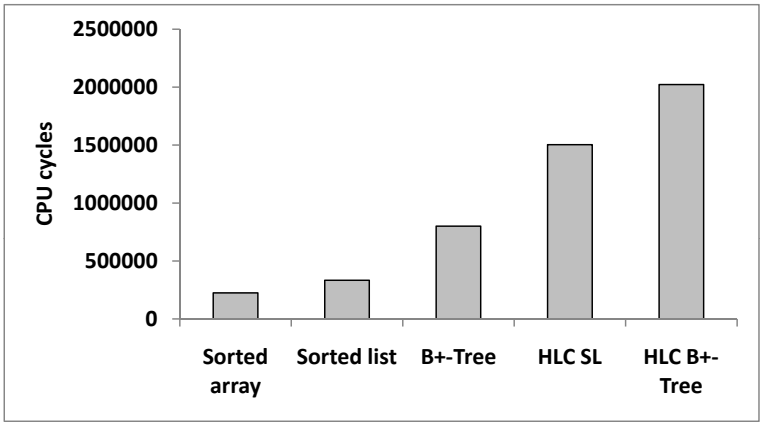

Figure 6.5: Performance comparison of different storage structures for a single record.

To evaluate the performance gain using evolving hierarchically-organized storage structures, we executed our micro benchmark using evolving versions of B+-Tree, HLC SL, and HLC B+-Tree storage structures. It can be observed from Figure 6.9, 6.10, 6.11, and 6.12 that each evolving storage structure version performs better than fixed storage structures in resource consumption and thus exhibit enhanced performance. It shows an important feature of our self-tuning approach, i.e., our approach to self-tuning has negligible overhead. Furthermore, Our design decision to use AOP to implement the self-tuning functionality ensures that the self-tuning is not integrated with any DBMS functionality; rather it can be removed

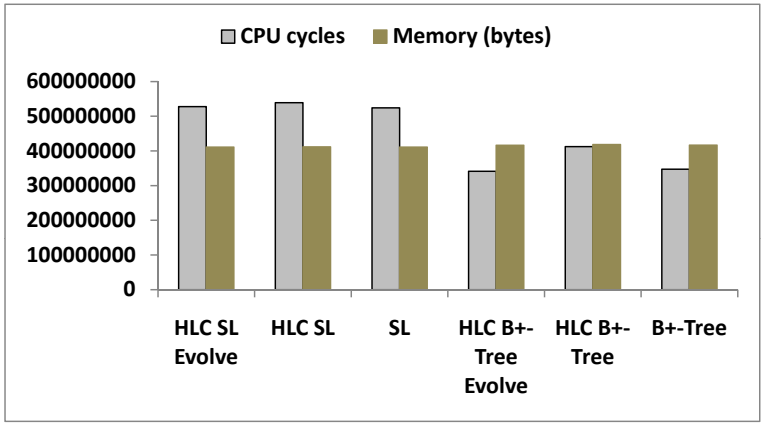

Figure 6.6: Performance comparison of different storage structures for 4048 records.

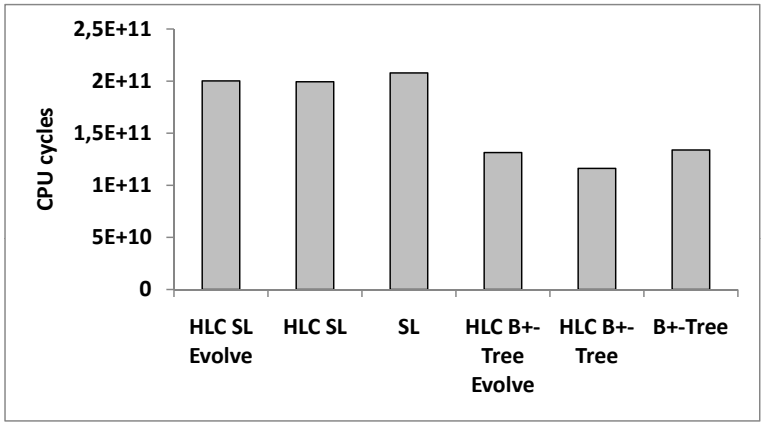

Figure 6.7: Performance comparison of different storage structures for 100K records.

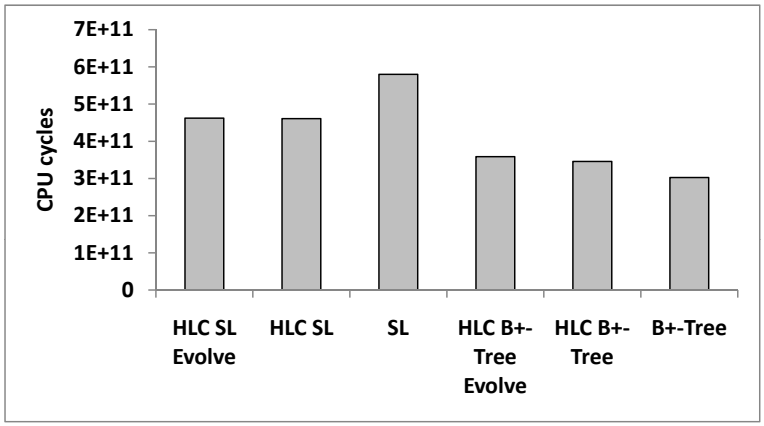

Figure 6.8: Performance comparison of different storage structures for 500K records.

when needed.

To further clarify the evolving storage structure's evolution behavior, we present the evaluation results for evolving HLC SL and evolving HLC B+-Tree storage

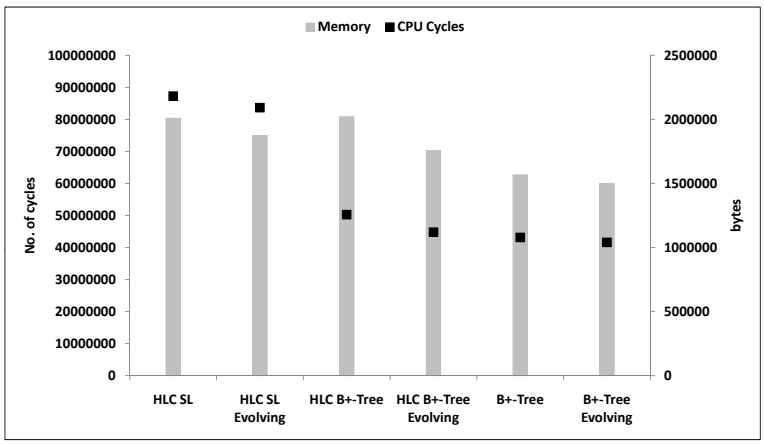

Figure 6.9: Evolving storage structures reduce memory and CPU cycles usage.

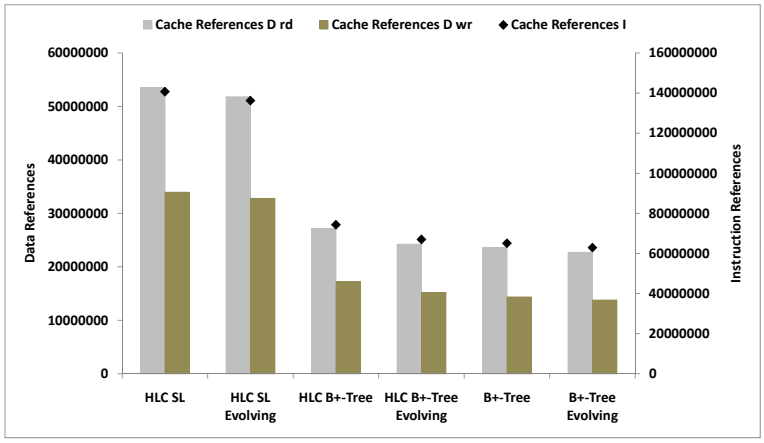

Figure 6.10: Evolving storage structures generate less cache references.

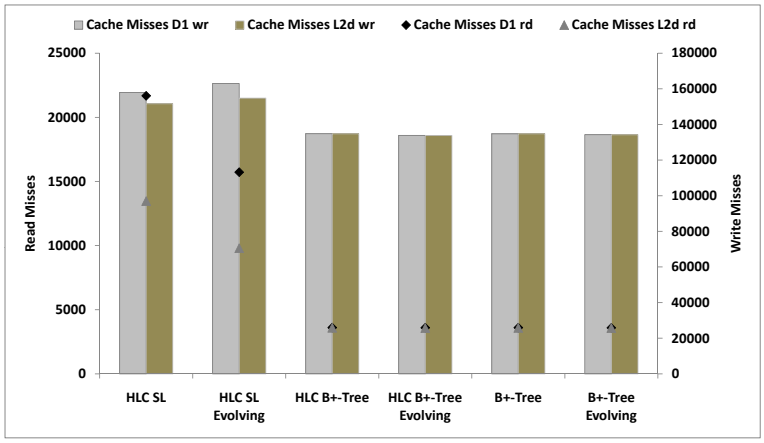

Figure 6.11: Evolving storage structures cause less data cache misses.

structures in Figure 6.13 and 6.14. In both figures, evolving storage structures evolves with the data growth. For both evaluations, we used the same page size for

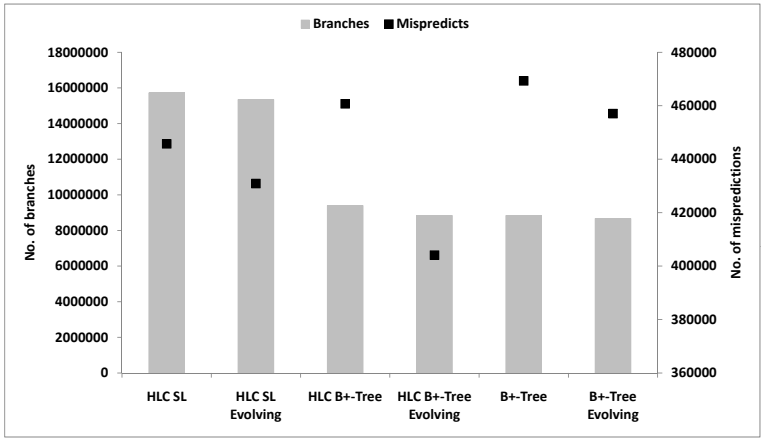

Figure 6.12: Evolving storage structures generate less branches and their misprediction.

HLC SL and HLC B+-Tree storage structures, therefore, first evolution takes place for both of them at the same data size, i.e., around 3500 records. It can be observed that the CPU cycles consumed for data management operations before first evolution are same for both storage structures. During first evolution, HLC SL evolves from a sorted array to a sorted list, whereas HLC B+-Tree evolves from a sorted array to a B+-Tree. It can be observed that both storage structures (i.e., sorted list and B+-Tree) consume different CPU cycles for data management operations. One can argue that a sorted list or a B+-Tree should also have behaved the same for 3500 records as did the sorted array. However, it is not the case. We also presented the behavior of the sorted list and the B+-Tree in both figures, and it can be observed that they do consume more CPU cycles than a sorted array for initial 3500 records.

It can be seen that both HLC SL and HLC B+-Tree storage structures consume more CPU cycles in comparison with sorted list and B+-Tree. This behavior is due to the complexity of these storage structures, which are meant to be used for extremely large data sizes. These two structures (i.e., HLC SL and HLC B+-Tree) automatically partition the data and uses separate buffer and index managers for each partition, which is not the requirement for presented 500K records storage. However, for demonstration of the evolution concept, we forced storage structures to evolve to HLC SL and HLC B+-Tree level for 500K records by defining it in an evolution path. Our HLC B+-Tree structure has some similarities with the partitioned B-Trees presented by Graefe [2003], which are designed to be used in the data warehousing domain and has been found efficient for performing sorting, index creation, and bulk insertion for large data. We could have reduced the CPU cycles requirement of HLC SL and HLC B+-Tree by using a single buffer and index

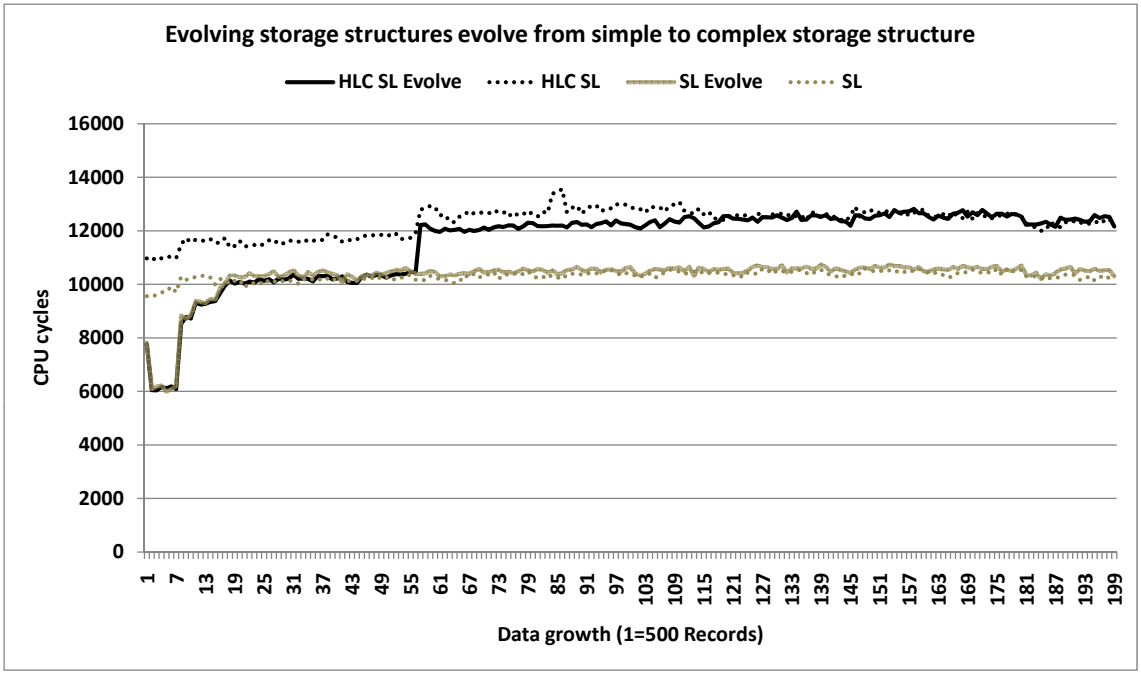

Figure 6.13: Evolving HLC SL storage structure evolution.

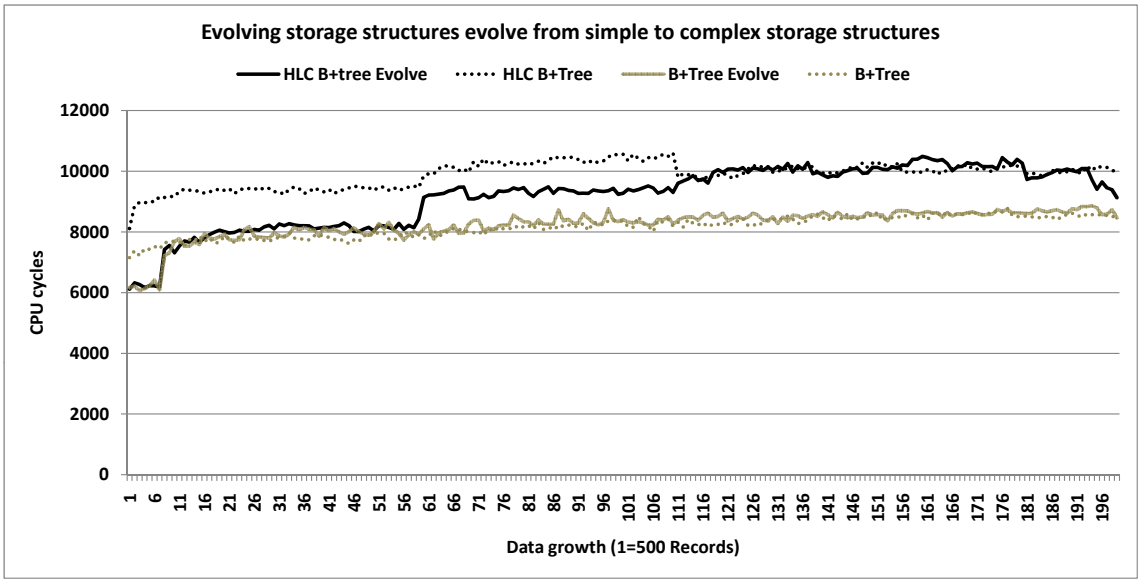

Figure 6.14: Evolving HLC B+-Tree storage structure evolution.

manager for all sorted lists and B+-Trees. However, this leads to a decrease in the locality of data and will eventually affect the cache utilization. Furthermore, this design also leads to interdependencies among all sorted lists and B+-Trees making self-tuning more problematic.

To demonstrate the difference of performance for different DSM based schemes and the performance gains using the evolving storage structures, we executed our micro benchmark in two configurations for all five schemes explained in Section 4. In the first configuration, we instantiated all columns as fixed HLC SL storage structure. In the second configuration, we used evolving HLC SL storage structure, which instantiate all columns as a sorted array on start up and then evolve the column with data growth to a sorted list, and finally to HLC SL (using the evolution path presented in Table 4.8). As different dictionary columns contain the different size of data for two dictionary based schemes, i.e., DMDSM and VDMDSM, in second configuration data of few dictionary columns can be accommodated in a sorted array, few evolve to a sorted list, and rest of the dictionary columns with large data size evolves to HLC SL (a sample scenario is shown in Table 3.1). For three other schemes, i.e., DSM, KDSM, and MDSM only columns with NULL values take benefit from the evolving storage structure's behavior of using minimal storage structure. However, they can still get the benefit of evolving the storage structures differently for each column exploiting the workload characteristics.

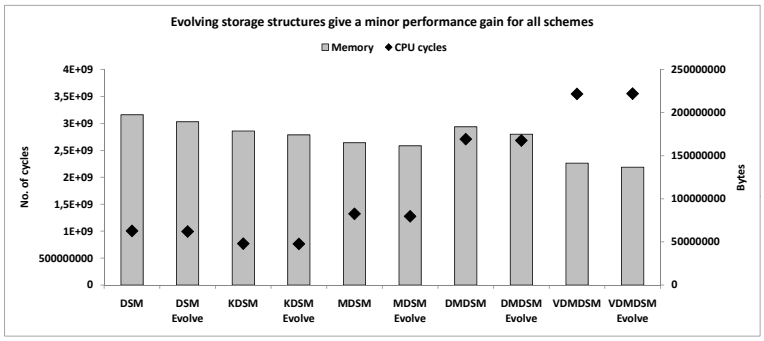

Figure 6.15: Performance comparison of different DSM based schemes in ECOS with a primary key based search criteria.

In Figure 6.15 and 6.16, results for evolving storage structures have evolve keyword appended in front of the scheme name. It can be observed that evolving storage structures perform better than fixed storage structures with minor performance gains. As we have discussed in Chapter 3, our work is based on the ideology from Chaudhuri and Weikum [2000]. They used the notion of "gain/pain ratio" to discuss the overall gain of their proposed approach. They advocate the ideology of less complex, more predictable, and self-tuning RISC-style components with minor compromise on performance to achieve overall improvement in "gain/pain ratio". Our results show the minor performance gain, which should be a good achievement considering the overall benefits we achieve in terms of simplicity, predictability, and

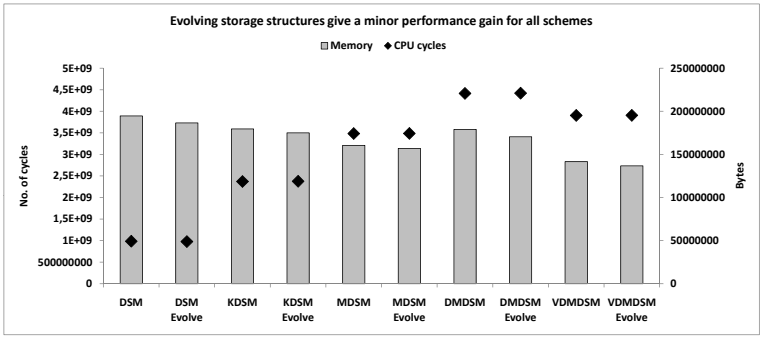

Figure 6.16: Performance comparison of different DSM based schemes in ECOS with a non-key based search criteria.

self-tuning with reduced human intervention.

In four proposed variations of the DSM based schemes in Section 4, i.e., KDSM, MDSM, DMDSM, and VDMDSM, we altered the 2-copy DSM by reducing the duplicate copies for columns, which should affect the time for read operations with search criteria on non-key attributes. For example, consider the KDSM scheme presented in Table 4.3. If data is searched with criteria involving column 0, which has two copies, i.e., Columnk0 clustered on keys, whereas Columnv0 clustered on values. The search can make use of Columnv0 to search for data using binary search. However, if the data is searched with criteria involving column 1 or column 2, which does not has any extra copy clustered on values. The search can only be performed using the linear search, which requires a scan through all records.

The performance of all proposed DSM schemes is also dependent on the number of attributes required by the query. In our micro benchmark, we test with the worst case scenario for all DSM based schemes by extracting all attributes [Holloway and DeWitt, 2008]. The MDSM, DMDSM, and VDMDSM schemes are most affected by this test case scenario, because these schemes also do not store any extra copy clustered on keys for key columns. This requires complete scan for key column attributes for search criteria on non-key attributes.

To assess the impact on performance for proposed changes in different DSM based schemes, we evaluated all five schemes in two configurations, i.e., the first configuration with search criteria involving key attribute as shown in Figure 6.15, and the second configuration with search criteria involving non-key attribute as shown in Figure 6.16. The results show that the DSM and the KDSM perform better for evaluation with search criteria on key-attributes, whereas for evaluation with search criteria on non-key attributes the DSM outperforms the other schemes. It is observed that storage requirement for the DSM is highest, whereas the storage

requirement is the lowest for the VDMDSM.

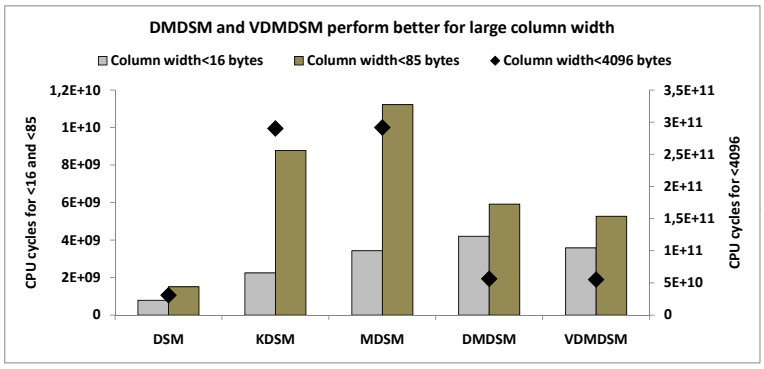

Figure 6.17: Performance improvement for dictionary based DSM schemes for large column width.

The results of Figure 6.15 and 6.16 are based on values with width of 16. We increased the width of value for all columns to 85 and then to 4096 to assess the impact of change in tuple width on performance of different schemes. It can be observed in Figure 6.17 that dictionary based schemes performance is improved and becomes comparable with standard 2-copy DSM scheme for large tuple width. However, KDSM and MDSM still perform poor. The reason of this improvement lies with the light weight compression that we achieve using dictionary columns. The dictionary columns ensure that the duplicate values are only stored once in the dictionary column reducing the amount of data processed during data management operations.

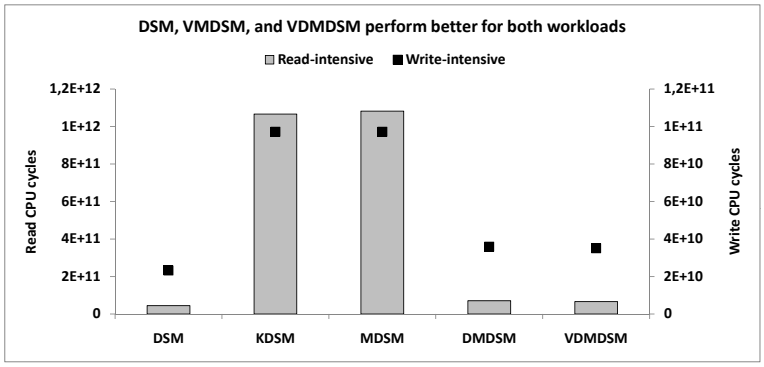

Figure 6.18: Performance comparison of different DSM based schemes in ECOS for read and write intensive workloads.

We also analyzed the performance difference for different DSM schemes on both the read-intensive and write-intensive workloads with the tuple width of 4096. DSM is known to perform well for read-intensive workload [Holloway and DeWitt, 2008].

It can be observed in Figure 6.18 that for DSM, DMDSM, and VDMDSM the read-intensive workload consumes fewer resources in comparison with the write-intensive workload, whereas for KDSM and MDSM, it is opposite with the same workloads. It can also be observed that for both the write-intensive workload and the read-intensive workload differences in performance between the 2-copy DSM and the dictionary based DSM schemes is minimum. This is a promising result for dictionary based schemes, and it shows their potential to act as a better alternative to DSM if their shortcomings can be overcome.

6.3 Summary

In this chapter, we evaluated the Cellular DBMS prototype using a custom micro benchmark. Our results showed that the presence of unused functionalities does affect the performance of the DBMS. We evaluated different storage structures to show their suitability for different database sizes. We also evaluated different DSM based schemes to present their problems and benefits. Our results showed that evolving storage structure provide minor performance gain over fixed storage structures.

7 Concluding remarks and future work

> Database systems are the result of an assemblage of thousands of pieces of code, which work flawlessly and in unison to provide the required functionality. This complexity hinders scientific progress within the time frame allotted for young researchers to complete their PhD research and it drains established groups when they embark on exploring a new route. Likewise, industry is engineering solutions that research projects must beat, making it even harder to enter this field.
>
> Martin L. Kersten
> *The Database Architecture Jigsaw Puzzle (Keynote Talk), ICDE 2008*

This chapter provides the summary of the thesis and present possibilities of the future work with a detailed listing of identified open research problems.

7.1 Summary of the dissertation

We presented the Cellular DBMS architecture, a RISC-style DBMS architecture for customizable and autonomous DBMS development, designed according to suggestions from Chaudhuri and Weikum [2000]. The architecture proposed to construct a large DBMS by using in concert multiple atomic, customized, and autonomous instances of embedded databases, called cells. In the architecture, each cell stores key/value pairs of data. This design decision enabled us to generate a cell of a limited functionality and simple interface, which resulted in a cell with more predictable performance. The architecture envisions the development of a complete autonomous DBMS by accumulating autonomic behavior of all participating cells. In the Cellular DBMS architecture, we introduced different compositions of cells as

autonomic structures to enable execution of autonomic behavior. We proposed to realize an autonomic behavior in DBMS using AOP.

We presented ECOS, a customizable self-tuning storage manager designed according to the Cellular DBMS architecture. ECOS stores data according to the column-oriented storage model, where each column stores key/value pairs of data. ECOS suggested customizations for each table in a database at two levels, i.e., at the table-level and at the column-level. At the table-level, we customized, how columns are stored physically for a logical schema design. We presented four variations of the DSM for table customization. ECOS utilized hierarchically-organized storage structures for data and index storage and suggested customization of each column initially to minimal suitable storage structure. We presented the concept of the evolution path, which defines how ECOS evolves a smallest simple storage structure into a large complex storage structure with the growth of data and change in workload.

The Cellular DBMS prototype is implemented as an SPL in the C++ programming language using the FeatureC++ tool for the FOP. Autonomy is implemented in the Cellular DBMS prototype using the AOP with the AspectC++ tool. The Cellular DBMS SPL can generate both in-memory database variants as well as persistent storage database variants. In the Cellular DBMS prototype, we used the feature derivative approach to handle the optional feature dependencies. Our prototype development experience suggested that the FOP should be used with coarser features, the use of preprocessor directives is imminent for multi-platform DBMS development, and the increase in the number of features also increases the number of tuning knobs and the number of feature derivatives.

For evaluation of the Cellular DBMS prototype, we set up a micro benchmark with synthetic data. Our evaluation results showed that if not-needed functionalities are part of the system, they do affect used functionalities. We also observed that for small databases, small and simple storage structures, such as the array and the sorted list perform better than the B+-Tree or the T-Tree with fewer resources. We evaluated proposed variations of different DSM schemes and found that the standard DSM and the KDSM perform better during evaluation with search criteria on key-attributes, whereas during evaluation with search criteria on non-key attributes, the standard DSM outperformed the other schemes. Furthermore, the storage requirement for the DSM is highest, whereas the storage requirement is lowest for the VDMDSM. We also observed that the dictionary based schemes (DMDSM and VDMDSM) performance was improved and became comparable with standard 2-copy

DSM scheme for large tuple width. Moreover, we also recognized that for DSM, DMDSM, and VDMDSM the read-intensive workload consumes fewer resources in comparison with the write-intensive workload, whereas for KDSM and MDSM, it is opposite for the same workloads. We also showed that the evolving storage structures perform better than fixed storage structures.

7.2 Future work

Our work on the Cellular DBMS architecture proved to be an ambitious project. Over the time with definition and implementation of foundation concepts, we came across many opportunities to extend our work in numerous directions. A very basic requirement of completely defining and implementing the Cellular DBMS architecture for a full-fledged DBMS is in itself a huge task. Considering available resources and time limitation, we have to restrict the scope of our work in this thesis. This resulted in a long list of future work that we left behind to takeover in the future as a research problem. In this section, we attempt to outline the possible future work opportunities that we found during this research, however, the list is not exhaustive.

7.2.1 Query processing

The query processing is a mandatory feature for all existing DBMS. It is an important tool for adhoc decision making. The adaptation of SQL and its popularity among database community is in itself a symbol for its success. However, not all features and capabilities of SQL are commonly used, instead there is a subset of popular features that are in common use [Chaudhuri and Weikum, 2000]. Chaudhuri and Weikum [2000] suggested SQL as painful and a big headache because of its high complexity and difficulty to learn and use it. However, they acknowledged the benefits of its core features, which include execution of selection-projection-join queries and aggregation. Stonebraker et al. [2007] also acknowledged the complexity of SQL. They termed SQL as "one size fits all" solution and stressed on the need of using a simple subset of DBMS specific SQL dialect. Considering the above facts, we suggest to use the customized SQL for the Cellular DBMS architecture, i.e., the features of SQL should be decided according to the features of the DBMS. Existing work from Sunkle et al. [2008] and Rosenmüller et al. [2009b] already provide us with the foundation work in this direction. They used the software product line approach to generate small and simple SQL dialects from the complete SQL standard.

For the Cellular DBMS architecture, we envision to revisit existing approaches for the query processing system in traditional DBMS. We intend to exploit the concept of an in-network acquisitional distributed query processor already in use for sensor networks [Gehrke and Madden, 2004; Madden et al., 2005; Yao and Gehrke, 2002]. The idea is to treat each cell in a Cellular DBMS similar as a sensor node and the concert of all cells for a complete DBMS should be treated as a network of sensor nodes. Later on, with the addition of the transaction management in the Cellular DBMS architecture, we will come up with scenarios in which few cells might not be available for data management operations because of locking. In such as scenario, an in-network acquisitional distributed query processor could benefit in querying data from the available cells providing the end-user with some early results to work on, meanwhile the locked cells get freed. We found the query processing functionality of existing in-network acquisitional distributed query processors, i.e., TinyDB [Madden et al., 2005] and Cougar [Yao and Gehrke, 2002], similar to Chaudhuri and Weikum [2000] concept of the Select-Project-Join (SPJ) query processing engine and Neumann and Weikum [2008] concept of the RISC-style RDF engine.

7.2.2 Mechanisms to adapt storage structures according to evolution paths alteration

In the ECOS, we used the concept of evolution paths to define, how ECOS evolves a smallest simple storage structure into a large complex storage structure using the evolving hierarchically-organized storage structures. We envision possibilities, where we need to alter evolution paths. This alteration could be done manually, or we can devise a mechanism to do it automatically in future. In both cases, how existing storage structures adapt to a new evolution path is still an open question and in this section we provide few suggestions towards possible solutions.

For special data management scenarios, such as intermediate result materialization during the query processing, it is easy to adapt to a new evolution path after alteration. Once the evolution path is altered, during next intermediate result materialization new evolution path will be used. However, for the most common data management scenario, where data only grows, we need a mechanism to alter existing hierarchically-organized storage structures according to new evolution paths. We propose three mechanisms for this purpose, which we termed as Disaster, War, and Preaching.

Disaster Once started, the disaster mechanism completes currently running DBMS operations and queues the new arriving requests. It instantiates the new instance of a hierarchically-organized storage structure according to the new evolution path and transfers the data to the new storage structure all-together. The old storage structure is considered dead after data transfer and is eliminated. Once data transfer is completed, queued requests of DBMS operations are completed with/over the new storage structure. We suggest that the disaster mechanism is optimal for hierarchically-organized storage structures with the small data and light workload. It consumes resources all-together to optimize the storage structure. Furthermore, we hypothesize that it improves the performance for all workloads simultaneously as the complete storage structure is evolved according to new evolution paths all-together. We term the time to adapt the new storage structure according to the updated evolution path as the "Revolution Period".

War The war mechanism instantiates the new instance of a hierarchically-organized storage structure according to the new evolution path and transfers data recursively starting from fine-grained atomic cell-level from the old storage structure to the new one. One by one for each cell, data is transferred to a new cell in the new storage structure, and then the old cell is considered dead and is eliminated. In war mechanism only the current DBMS operations of a single cell are completed, and new operations are queued before the data transfer, which ensures that DBMS operations that do not involve the particular cell can be completed without any delay. We suggest that war mechanism is appropriate for storage structures with medium-sized data storage and workload.

Preaching The preaching mechanism works similar to the war mechanism except the difference that decision to transfer data to a new cell is decided according to the cell state. Preaching mechanism may take longer to change cells, but it attempts to ensure that minimum overhead is incurred for ongoing DBMS operations. A cell waits for data transfer until/unless either it is in an idle state, or it contains the workload well within the threshold defined by the DBMS administrator.

7.2.3 The Cellular DBMS architecture and the multi-core era

We want to extend our existing Cellular DBMS architecture and implementation to exploit parallelism of the many-core architecture using the message passing programming model [Wilson, 2005]. Here we only outline the architecture specific fu-

ture work that we found interesting. We intend to make use of the Intel Single-chip Cloud Computer (SCC) [1], a research microprocessor with 48 IA cores integrated on a single CPU chip. Each core on SCC is optimized for the message passing programming model. Each SCC core can communicate with other cores using shared memory, but there is no hardware coherence for shared memory. SCC suggests the shared memory coherency using software. We observe a need to analyze the impact of using the software coherence on the data management and possible optimization of the software coherence schemes to suit the data management. Furthermore, the message passing protocol of SCC introduces message passing (MP) read and write misses [Howard et al., 2010], which raises the need to analyze the impact of MP read/write misses on data management workloads. If the impact is significant, then we should identify the mechanism to reduce it.

Second important research direction is the efficient use of fine-grained power management provided by SCC for the data management. SCC allows dynamic voltage and frequency scaling (DVFS), which is a mechanism that allows the change in voltage and frequency levels of a core using software instructions. A Cellular DBMS can use the SCC power-management provision to increase or decrease the number of active cores according to the workload reducing the energy consumption. Here, we found two important characteristics of SCC power management that are needed to be analyzed. First one is related to frequency scaling. What will be the impact of frequency scaling on data management when two cores are processing the related/dependent data at different frequencies? The second important issue is the power breakdown. According to [Howard et al., 2010], for the full power breakdown of 125.3W, 69% of power is consumed by the cores and 19% power is consumed by the memory controller (MC) and DDR3-800, whereas for the low power breakdown of 24.7%; cores utilizes only 21% of power and MC with DDR3-800 utilizes 69% of power. These facts show that MC and DDR3-800 power consumption is not reduced in proportion of reduction in power consumption of cores. This motivates us to investigate the impact of using large memories on the power consumption for a DBMS.

Third important research issue is to optimize a DBMS to reduce the overhead of SCC DDR3 access fairness. Access fairness is a mechanism that ensures that all cores get the equally likely access to the memory. SCC DDR3 access fairness[2]

[1] "Single-chip Cloud Computer ", `http://techresearch.intel.com/ProjectDetails.aspx?Id=1`, Accessed: 21-06-2011

[2] "Single-chip Cloud Computer" An experimental many-core processor from Intel Labs, Intel Labs Single-chip Cloud Computer Symposium, March 16, 2010,

results show that round-trip latency per core increases and bandwidth per core decreases with an increase in the number of cores. We can argue that for memory bound processes, increase in the number of cores will result in degraded performance because of under-utilization of the processing cores. We observe many memory bound operations in DBMS, e.g., in-memory join processing. One solution could be to use the mechanisms that reduce the memory usage, such as the one proposed by Zukowski et al. [Zukowski et al., 2005]. Another solution could be to minimize the use of the number of cores for memory bound operations, but this may reduce the benefit of using the SCC.

7.2.4 Multiple storage models

The current Cellular DBMS prototype implementation uses DSM as its storage model. In future, we want to implement more storage models to have better customization options. We plan to add only PAX [Ailamaki et al., 2002] in the Cellular DBMS prototype because of its better cache and memory bandwidth utilization. However, it is needed to be assessed, how much PAX is suitable for a self-tuning storage manager. For NSM, we intend to use existing embedded databases with NSM storage model. Motivation behind this effort is to customize cells to NSM for transaction processing workload, to DSM for decision support workload, and to PAX for a mix of both.

7.2.5 The Cellular DBMS architecture and the cloud data services

The focus of cloud data services is to provide more predictable services with more reliable service level agreements instead of functionalities [Agrawal et al., 2009]. Most often cloud data services provide limited services, i.e., restricted API, minimal query language, limited consistency guarantee, and constraints on resource utilization [Agrawal et al., 2009]. We suggest that the Cellular DBMS architecture is suitable for cloud data services, because its autonomy ensures less human intervention in administration, its customization allows efficient utilization of commodity hardware in shared infrastructure, and its cells based implementation can handle workload variance effectively.

`http://communities.intel.com/servlet/JiveServlet/previewBody/5902-102-1-9037/SCC_Sympossium_Mar162010_GML_final1123.pdf`, Accessed: 21-06-2011

7.2.6 Resource balancing in the Cellular DBMS architecture

In a distributed environment, we envision possibilities of resource balancing using distributed cells in the Cellular DBMS architecture. We have listed down our vision here as part of the future work.

Cell mobility The cell mobility means the capability of a Cellular DBMS to move a cell from one processing environment to another. Mobility of cells could be across processes on a single system or across systems connected via network. The cell mobility becomes possible because of the design principle of the Cellular DBMS architecture, which requires an instance of a small footprint customized embedded database to be used as cell. The motivation behind mobility is to achieve load balancing and to use resources efficiently. The cell mobility can be used in many different ways. For example, one scenario is a distributed network of interconnected embedded devices. Consider a case of an embedded device on which a cell is deployed, and it is heavily loaded with processing. We envision moving a cell to another relatively idle device. If all devices are over-consumed, then a new device can be brought into the network and then cells can be moved to that new device for load balancing. Cell mobility can also be used in other scenarios, such as distributed network of interconnected processors, or interconnected processor cores, e.g., many-core processors.

Virtual resource Embedded systems have become an important part of hardware industry. Most of the digital appliances that we use these days comes up with some form of an embedded system in them. Moreover, these embedded systems also have data management needs. We suggest that the Cellular DBMS architecture could be used for data management on embedded systems, where cells deployed over multiple devices operate in concert to achieve the data management need of the complete systems, providing the end-user a view of single DBMS. Embedded systems are different from high-end systems by means of resources. In an embedded system, we normally have resource constraints on a single device, but in the network of interacting embedded systems, there are many resources that are available across a network and are idle. We envision in the Cellular DBMS architecture to virtually-combine these scattered resources as a virtual resource, i.e., it gives a virtual view of the scattered small resources 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 has an enough ca-

pacity on its own. In this case, the Cellular DBMS architecture approach is capable of storing data distributed across devices using cells and transparently providing to an application a view of a single large resource capable of accommodating 18 KB of data. This concept also gives us a clue that how the Cellular DBMS architecture can use cells for fragmenting data on multiple embedded devices, sensor nodes, or high-end enterprise servers.

7.2.7 Future work from software engineering perspective

In this section, we outline few critical future directions from software engineering perspective for the Cellular DBMS architecture.

Efficient variant testing For an SPL-based DBMS project, over a project timeline, more and more features get introduced in a DBMS SPL and the number of program variants tends to grow. With the introduction of each new feature or a change in functionality of existing feature, every time a DBMS developer encounter the problem of manual testing of the all possible variants for a DBMS SPL. It is a time consuming and error prone process. Our development experience suggested that more time is invested in testing an SPL then to develop it. Right now, we do not have any quantitative results for our claim, however, this issue is well known in SPL community specifically and for software testing generally. We do not plan to add any new features into our DBMS SPL implementation unless we have an automated SPL testing mechanism functional with us. We can benefit from many existing approaches available in literature, such as [Tevanlinna et al., 2004], [Kim et al., 2011], and [Stricker et al., 2010]. However, tool support is still a big problem in this domain.

Minimizing code replication In FOP, it is often the case that we encounter code replication and redundancy among features [Schulze et al., 2010]. Specifically in our case of the Cellular DBMS implementation, using differently composed cells simultaneously while minimizing code replication is an important open issue. A software engineering based solution is needed to solve this problem. Existing work from Rosenmüller et al. on multiple SPL [Rosenmüller and Siegmund, 2010], component families [Rosenmüller et al., 2010], multi-dimensional variability modeling [Rosenmüller et al., 2011b], and flexible feature binding in an SPL [Rosenmüller et al., 2011a] is an important research progress from software engineering perspective that needs to be used in our Cellular DBMS architecture implementation.

7.2.8 Miscellaneous

Finally, here we briefly outline few more future directions of work that we found necessary to mention.

- Our current implementation of a cell evolution is explicitly programmed. An important future direction is to enable implicit learning in the Cellular DBMS architecture for self-* capabilities.

- The transaction management is a critical functionality in existing DBMS. We understand the need of transaction management support in the Cellular DBMS prototype implementation for performance comparison of our approach with other existing DBMS that support transaction. For transaction management in the Cellular DBMS architecture, we intend to exploit the techniques from distributed transaction management systems.

- We want to integrate our existing ECOS prototype implementation into MySQL as a storage engine to evaluate its benefits and usability using a standard query processor. It is also important to generate the standard TPC benchmark results for our prototype implementation. We intend to execute the standard benchmarks (i.e., TPC-C [TPC-C], TPC-H [TPC-H], and TPC-E [TPC-E]) to show the effectiveness and benefit of our approach.

- Weikum et al. [2002] stressed on the need of making the overhead of statistics management in a self-tuning DBMS predictable. For a Cellular DBMS, it is an important design decision to identify, how often and what information should be monitored as heredity information. Additionally, what will be the life time for certain heredity information and how to reduce the storage requirement while at the same time ensuring efficient retrieval.

- How sensitive are tuning knobs to each other? How change in one tuning knob effect the performance of another? A systematic study to assess the impact of tuning knobs on each other in a database system is a well known open research problem [Weikum et al., 2002].

A List of features in the Cellular DBMS prototype

The Listing A.1 presents the configuration file for the Cellular DBMS prototype listing all the 151 features. The symbol # is used to disable features as well as to write comments.

Listing A.1: The configuration file of the Cellular DBMS prototype listing all features

```
Base
Main
#Main.OPC #For single cell
#Main.MPC #For multiple cells
Main.CPC #For composite cells
#Main.HLComposite #For high-level composite cell
Main.UserInterface

#OS.i386 should be used for Windows and Linux
OS.i386
OS.Windows
#OS.Linux
#OS.NutOS

Test
#Test.Innovation

#Page implementation according to sorted array
StorageManager.Page.FLRPage
#StorageManager.Page.FLRPage.Persistent
StorageManager.Page.FLRPage.ByVal
StorageManager.Page.FLRPage.Write
StorageManager.Page.FLRPage.Write.ByVal
```

```
StorageManager.Page.FLRPage.Delete
StorageManager.Page.FLRPage.Delete.ByVal
StorageManager.Page.FLRPage.Multi
StorageManager.Page.FLRPage.Multi.Write
StorageManager.Page.FLRPage.Multi.Status
StorageManager.Page.FLRPage.Multi.Delete
StorageManager.Page.FLRPage.Status

#Page implementation according to heap array
#StorageManager.Page.HPage
#StorageManager.Page.HPage.Persistent
#StorageManager.Page.HPage.ByVal
#StorageManager.Page.HPage.Write
#StorageManager.Page.HPage.Write.ByVal
#StorageManager.Page.HPage.Delete
#StorageManager.Page.HPage.Delete.ByVal
#StorageManager.Page.HPage.Multi
#StorageManager.Page.HPage.Multi.Write
#StorageManager.Page.HPage.Multi.Status
#StorageManager.Page.HPage.Multi.Delete
#StorageManager.Page.HPage.Status

#Structure with key/value definition
StorageManager.Page.RECORD

#Structure with DB definition
StorageManager.Page.DBI

#Buffer manager, multiple pages support, in-memory only
BufferManager.InMemory
#BufferManager.InMemory.MemoryAlloc.Static
BufferManager.InMemory.MemoryAlloc.Dynamic

#Buffer manager, multiple pages support with secondary storage
#BufferManager
#BufferManager.MemoryAlloc.Static
#BufferManager.MemoryAlloc.Dynamic
#BufferManager.PageFind.Hash
```

```
#BufferManager.PageReplace.LRU
#BufferManager.PageReplace.LFU

#Storage Manager Buffer Manager, in-memory
StorageManagerBufferManager.InMemory
#StorageManagerBufferManager.InMemory.MemoryAlloc.Static
StorageManagerBufferManager.InMemory.MemoryAlloc.Dynamic

#Index Buffer Manager, multiple indexs support, in-memory only
IndexBufferManager.InMemory
#IndexBufferManager.InMemory.MemoryAlloc.Static
IndexBufferManager.InMemory.MemoryAlloc.Dynamic

#Index Buffer Manager, index support with secondary storage
#IndexBufferManager
#IndexBufferManager.MemoryAlloc.Static
#IndexBufferManager.MemoryAlloc.Dynamic
#IndexBufferManager.IndexFind.Hash
#IndexBufferManager.IndexReplace.LRU
#IndexBufferManager.IndexReplace.LFU

#Sorted list
StorageManager.SortedList
StorageManager.SortedList.Read
StorageManager.SortedList.Read.ByVal
StorageManager.SortedList.Write
StorageManager.SortedList.Write.ByVal
StorageManager.SortedList.Manage

#Heap list
#StorageManager.HeapList
#StorageManager.HeapList.Read
#StorageManager.HeapList.Read.ByVal
#StorageManager.HeapList.Write
#StorageManager.HeapList.Write.ByVal
#StorageManager.HeapList.Manage

#SQLite
```

```
#StorageManager.SQLite
#StorageManager.SQLite.Read
#StorageManager.SQLite.Read.ByVal
#StorageManager.SQLite.Write
#StorageManager.SQLite.Write.ByVal
#StorageManager.SQLite.Manage

#Berkeley DB
#StorageManager.BDB
#StorageManager.BDB.Read
#StorageManager.BDB.Read.ByVal
#StorageManager.BDB.Write
#StorageManager.BDB.Write.ByVal
#StorageManager.BDB.Manage


#B+-Tree
#StorageManager.BPlusTree
#StorageManager.BPlusTree.Manage
#StorageManager.BPlusTree.Read
#StorageManager.BPlusTree.Read.ByVal
#StorageManager.BPlusTree.Write
#StorageManager.BPlusTree.Write.ByVal
#StorageManager.BPlusTree.Delete
#StorageManager.BPlusTree.Delete.ByVal

#Composite cell, MDSM
#StorageManager.Composite
#StorageManager.Composite.IM #Autonom not needed
#StorageManager.Composite.SL #Autonom feature is mandatory
#StorageManager.Composite.HLC #Autonom and SL_HLC mandatory
#StorageManager.Composite.SL_HLC
#StorageManager.Composite.Read
#StorageManager.Composite.Write
#StorageManager.Composite.Manage

#Structure with COLUMN definition, MDSM
#StorageManager.Composite.COLUMN #Needed to HLComposite
```

```
#Composite class, KDSM
#StorageManager.KDSM
#StorageManager.KDSM.IM #Autonom not needed
#StorageManager.KDSM.SL #Autonom feature is mandatory
#StorageManager.KDSM.HLC #Autonom and SL_HLC mandatory
#StorageManager.KDSM.SL_HLC
#StorageManager.KDSM.Read
#StorageManager.KDSM.Write
#StorageManager.KDSM.Manage

#Structure with COLUMN definition, KDSM
#StorageManager.KDSM.COLUMN #Needed to HLComposite

#Composite class, DSM
StorageManager.DSM
#StorageManager.DSM.IM #Autonom not needed
#StorageManager.DSM.SL #Autonom feature is mandatory
StorageManager.DSM.HLC #Autonom and SL_HLC mandatory
StorageManager.DSM.SL_HLC
StorageManager.DSM.Read
StorageManager.DSM.Write
StorageManager.DSM.Manage

#Structure with COLUMN definition, DSM
StorageManager.DSM.COLUMN #Needed to HLComposite

#Composite class, DMDSM
#StorageManager.DICTCOS
#StorageManager.DICTCOS.IM #Autonom not needed
#StorageManager.DICTCOS.SL #Autonom feature is mandatory
#StorageManager.DICTCOS.HLC #Autonom and SL_HLC mandatory
#StorageManager.DICTCOS.SL_HLC
#StorageManager.DICTCOS.Read
#StorageManager.DICTCOS.Write
#StorageManager.DICTCOS.Manage

#Composite class, VDMDSM
```

```
#StorageManager.VECTCOS
#StorageManager.VECTCOS.IM #Autonom not needed
#StorageManager.VECTCOS.SL #Autonom feature is mandatory
#StorageManager.VECTCOS.HLC #Autonom and SL_HLC mandatory
#StorageManager.VECTCOS.SL_HLC
#StorageManager.VECTCOS.Read
#StorageManager.VECTCOS.Write
#StorageManager.VECTCOS.Manage

#HLC cell, Autonomy.Evolve & StorageManager mandatory
StorageManager.HLComposite
StorageManager.HLComposite.Manage
StorageManager.HLComposite.Read
StorageManager.HLComposite.Read.ByVal
StorageManager.HLComposite.Write
StorageManager.HLComposite.Write.ByVal
StorageManager.HLComposite.Delete
StorageManager.HLComposite.Delete.ByVal

#Handles memory allocation for StorageManager for HLC
StorageManager.Buffers

#Evolving behaviour for cells, Need COLUMN, used by HLC
Autonomy.Evolve

#Monitoring of cells, using aspect
Autonomy.Monitor

#Autonomic behaviour for cells
Autonomy.Autonom

#Class with data dictionary functionality
#DataDictionary
DataDictionary.InMemory

#Class with index dictionary functionality
#IndexDictionary
IndexDictionary.InMemory
```

Bibliography

Daniel J. Abadi, Samuel R. Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proc. Int'l Conf. Management of data (SIGMOD)*, pages 671–682. ACM Press, 2006.

Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proc. Int'l Conf. Very large data bases (VLDB)*, pages 411–422. VLDB Endowment, 2007.

Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *Proc. Int'l Conf. Management of data (SIGMOD)*, pages 967–980. ACM Press, 2008.

Rakesh Agrawal, Anastasia Ailamaki, Philip A. Bernstein, Eric A. Brewer, Michael J. Carey, Surajit Chaudhuri, Anhai Doan, Daniela Florescu, Michael J. Franklin, Hector Garcia-Molina, Johannes Gehrke, Le Gruenwald, Laura M. Haas, Alon Y. Halevy, Joseph M. Hellerstein, Yannis E. Ioannidis, Hank F. Korth, Donald Kossmann, Samuel R. Madden, Roger Magoulas, Beng Chin Ooi, Tim O'Reilly, Raghu Ramakrishnan, Sunita Sarawagi, Michael Stonebraker, Alexander S. Szalay, and Gerhard Weikum. The Claremont report on database research. *Commun. ACM*, 52(6):56–65, 2009. ISSN 0001-0782.

Sanjay Agrawal, Eric Chu, and Vivek Narasayya. Automatic physical design tuning: workload as a sequence. In *Proc. Int'l Conf. Management of data (SIGMOD)*, pages 683–694. ACM Press, 2006.

Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *Proc. Int'l Conf. Very large data bases (VLDB)*, pages 266–277. Morgan Kaufmann Publishers Inc., 1999.

Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal*, 11(3): 198–215, 2002. ISSN 1066-8888.

Esther R. Angert. Alternatives to binary fission in bacteria. *Nature Reviews Microbiology*, 3(3):214–224, 2005.

Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 125–140. Springer-Verlag, 2005.

Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, James N. Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, and Vera Watson. System R: relational approach to database management. *ACM Trans. Database Syst.*, 1:97–137, 1976. ISSN 0362-5915.

Don S. Batory. On searching transposed files. *ACM Trans. Database Syst.*, 4(4): 531–544, 1979. ISSN 0362-5915.

Don S. Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 187–197. IEEE Computer Society, 2003.

Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. In *Proc. Annual Symposium on Foundations of Computer Science (FOCS)*, pages 399–409. IEEE Computer Society, 2000.

Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. Biennial Conf. on Innovative Data Systems Research (CIDR)*, pages 225–237, 2005.

Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51(12):77–85, 2008. ISSN 0001-0782.

Nicolas Bruno and Surajit Chaudhuri. Physical design refinement: The 'merge-reduce' approach. *ACM Trans. Database Syst.*, 32, 2007a. ISSN 0362-5915.

Nicolas Bruno and Surajit Chaudhuri. An Online Approach to Physical Design Tuning. In *Proc. Int'l Conf. Data Engineering (ICDE)*, pages 826–835. IEEE Computer Society, 2007b.

Nicolas Bruno and Rimma V. Nehme. Configuration-parametric query optimization for physical design tuning. In *Proc. Int'l Conf. Management of data (SIGMOD)*, pages 941–952. ACM Press, 2008.

Humberto Cervantes and Richard S. Hall. Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 614–623. IEEE Computer Society, 2004.

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26:4:1–4:26, 2008. ISSN 0734-2071.

Surajit Chaudhuri and Vivek Narasayya. Self-tuning database systems: a decade of progress. In *Proc. Int'l Conf. Very large data bases (VLDB)*, pages 3–14. VLDB Endowment, 2007.

Surajit Chaudhuri and Gerhard Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In *Proc. Int'l Conf. Very large data bases (VLDB)*, pages 1–10. Morgan Kaufmann Publishers Inc., 2000.

Shimin Chen, Phillip B. Gibbons, Todd C. Mowry, and Gary Valentin. Fractal prefetching B+-Trees: optimizing both cache and disk performance. In *Proc. Int'l Conf. Management of data (SIGMOD)*, pages 157–168. ACM Press, 2002.

Corporate Act-Net Consortium. The active database management system manifesto: a rulebase of ADBMS features. *SIGMOD Rec.*, 25:40–49, 1996. ISSN 0163-5808.

George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. *SIGMOD Rec.*, 14:268–279, 1985. ISSN 0163-5808.

Benoit Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zait, and Mohamed Ziauddin. Automatic SQL tuning in Oracle 10g. In *Proc. Int'l Conf. Very large data bases (VLDB)*, pages 1098–1109. VLDB Endowment, 2004.

Debabrata Dash and Anastasia Ailamaki. CoPhy: Automated Physical Design with Quality Guarantees. Technical Report CMU-CS-10-109, Carnegie-Mellon University School of Computer Science, 2010.

Arjen P. de Vries, Nikos Mamoulis, Niels J. Nes, and Martin L. Kersten. Efficient image retrieval by exploiting vertical fragmentation. Technical Report INS-R0109, CWI, 2001.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall,

and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007. ISSN 0163-5980.

Johannes Gehrke and Samuel R. Madden. Query Processing in Sensor Networks. *IEEE Pervasive Computing*, 3(1):46–55, 2004. ISSN 1536-1268.

Goetz Graefe. The Five-Minute Rule 20 Years Later: and How Flash Memory Changes the Rules. *Queue*, 6:40–52, 2008. ISSN 1542-7730.

Goetz Graefe. Sorting And Indexing With Partitioned B-Trees. In *Proc. Biennial Conf. on Innovative Data Systems Research (CIDR)*, 2003.

Jim Gray and Goetz Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *SIGMOD Rec.*, 26:63–68, 1997. ISSN 0163-5808.

Jim Gray and Franco Putzolu. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for CPU time. *SIGMOD Rec.*, 16:395–398, 1987. ISSN 0163-5808.

Philip Greenwood and Lynne Blair. Using Dynamic Aspect-Oriented Programming to Implement an Autonomic System. In *Proc. Dynamic Aspects Workshop (DAW)*, pages 76–88, 2004.

Theo Härder. DBMS Architecture–The Layer Model and its Evolution. *Datenbank-Spektrum*, 5(13):45–57, 2005.

Theo Härder and Andreas Reuter. Concepts for Implementing a Centralized Database Management System. In *Proc. International Computing Symposium (ICS)*, pages 28–60, 1983a.

Theo Härder and Andreas Reuter. Database System for Non-Standard Applications. In *Proc. International Computing Symposium (ICS)*, pages 452–466, 1983b.

Stavros Harizopoulos and Anastassia Ailamaki. A Case for Staged Database Systems. In *Proc. Biennial Conf. on Innovative Data Systems Research (CIDR)*, 2003.

Stavros Harizopoulos, Daniel J. Abadi, Samuel R. Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In *Proc. Int'l Conf. Management of data (SIGMOD)*, pages 981–992. ACM Press, 2008.

Bingsheng He and Qiong Luo. Cache-oblivious databases: Limitations and opportunities. *ACM Trans. Database Syst.*, 33:8:1–8:42, 2008. ISSN 0362-5915.

Joseph L. Hellerstein. Automated Tuning Systems: Beyond Decision Support. In *Proc. Int'l Conf. Computer Measurement Group*, pages 263–270. Computer Measurement Group, 1997.

Allison L. Holloway and David J. DeWitt. Read-optimized databases, in depth. *The VLDB Journal*, 1:502–513, 2008. ISSN 2150-8097.

Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, Fabrice Pailet, Shailendra Jain, Tiju Jacob, Satish Yada, Sraven Marella, Praveen Salihundam, Vasantha Erraguntla, Michael Konow, Michael Riepen, Guido Droege, Joerg Lindemann, Matthias Gries, Thomas Apel, Kersten Henriss, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borkar, Vivek De, Rob Van Der Wijngaart, and Timothy Mattson. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *Proc. Int'l Conf. Solid-State Circuits (ISSCC)*, pages 19–21. IEEE Computer Society, 2010.

Florian Irmert, Thomas Fischer, Frank Lauterwald, and Klaus Meyer-Wegener. The Adaptation Model of a Runtime Adaptable DBMS. In *Proc. British National Conf. on Databases (BNCOD)*, pages 189–192. Springer-Verlag, 2009.

Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Carnegie-Mellon University Software Engineering Institute, 1990.

Ilkka Karasalo and Per Svensson. An overview of Cantor: a new system for data analysis. In *Proc. Int'l Workshop on Statistical and scientific database management (SSDBM)*, pages 315–324. Lawrence Berkeley Laboratory, 1983.

Ilkka Karasalo and Per Svensson. The design of Cantor: a new system for data analysis. In *Proc. Int'l workshop on Statistical and scientific database management (SSDBM)*, pages 224–244. Lawrence Berkeley Laboratory, 1986.

Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 311–320. ACM Press, 2008.

Christian Kästner, Sven Apel, Syed Saif ur Rahman, Marko Rosenmüller, Don S. Batory, and Gunter Saake. On the Impact of the Optional Feature Problem: Analysis and Case Studies. In *Proc. Int'l Conf. Software Product Line (SPLC)*, pages 181–190. SEI, 2009.

Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. Type Checking Annotation-Based Product Lines. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2011. to appear; submitted 8 Jun 2010, accepted 4 Jan 2011.

Martin L. Kersten. The Database Architecture Jigsaw Puzzle. In *Proc. Int'l Conf. Data Engineering (ICDE)*, pages 3–4. IEEE Computer Society, 2008.

Martin L. Kersten. A Cellular Database System for the 21st Century. In *Proc. Int'l Workshop on Active, Real-Time, and Temporal Database Systems (ARTDB)*, pages 39–50. Springer-Verlag, 1998.

Martin L. Kersten and Stefan Manegold. Cracking the Database Store. In *Proc. Biennial Conf. on Innovative Data Systems Research (CIDR)*, pages 213–224, 2005.

Martin L. Kersten and Arno Siebes. Bio-Inspired Data Management. In *Intelligent Algorithms in Ambient and Biomedical Computing*, pages 37–56. Springer-Verlag, 2006.

Martin L. Kersten, Gerhard Weikum, Michael J. Franklin, Daniel A. Keim, Alex Buchmann, and Surajit Chaudhuri. A database striptease or how to manage your personal databases. In *Proc. Int'l Conf. Very large data bases (VLDB)*, pages 1043–1044. VLDB Endowment, 2003.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 220–242. Springer-Verlag, 1997.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 327–353. Springer-Verlag, 2001.

Chang Hwan Peter Kim, Don S. Batory, and Sarfraz Khurshid. Reducing combinatorics in testing product lines. In *Proc. Int'l Conf. Aspect-oriented Software Development (AOSD)*, pages 57–68. ACM Press, 2011.

Tobin J. Lehman and Michael J. Carey. A Study of Index Structures for Main Memory Database ManagementSystems. In *Proc. Int'l Conf. Very large data bases (VLDB)*, pages 294–303. Morgan Kaufmann Publishers Inc., 1986.

Thomas Leich, Sven Apel, and Gunter Saake. Using Step-Wise Refinement to Build a Flexible Lightweight Storage Manager. In *Proc. East-European Conf. on Advances in Databases and Information Systems (ADBIS)*, pages 324–337. Springer-Verlag, 2005.

Christian Lemke, Kai-Uwe Sattler, Franz Faerber, and Alexander Zeier. Speeding up queries in column stores: a case for compression. In *Proc. Int'l Conf. Data Warehousing and Knowledge Discovery (DaWaK)*, pages 117–129. Springer-Verlag, 2010.

Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 105–114. ACM Press, 2010.

Sam S. Lightstone, Guy Lohman, and Danny Zilio. Toward autonomic computing with DB2 universal database. *SIGMOD Rec.*, 31(3):55–61, 2002. ISSN 0163-5808.

Jia Liu, Don S. Batory, and Christian Lengauer. Feature oriented refactoring of legacy applications. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 112–121. ACM Press, 2006.

Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005. ISSN 0362-5915.

Tanu Malik, Xiaodan Wang, Randal Burns, Debabrata Dash, and Anastasia Ailamaki. Automated physical design in database caches. In *Proc. Int'l Conf. Data Engineering (ICDE) Workshop*, pages 27–34. IEEE Computer Society, 2008.

Stefan Manegold, Martin L. Kersten, and Peter A. Boncz. Database architecture evolution: mammals flourished long before dinosaurs became extinct. *Proc. VLDB Endowment*, 2:1648–1653, August 2009. ISSN 2150-8097.

Dennis McCarthy and Umeshwar Dayal. The architecture of an active database management system. *SIGMOD Rec.*, 18:215–224, 1989. ISSN 0163-5808.

MySQL Database. http://www.mysql.com, last accessed: 21-06-2011.

Thomas Neumann and Gerhard Weikum. RDF-3X: a RISC-style engine for RDF. *The VLDB Journal*, 1:647–659, 2008. ISSN 2150-8097.

Michael A. Olson. Selecting and Implementing an Embedded Database System. *Computer*, 33(9):27–34, 2000. ISSN 0018-9162.

Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proc. USENIX Annual Technical Conf.*, pages 43–42. USENIX Association, 1999.

Oracle Berkeley DB. http://www.oracle.com/technology/products/berkeley-db/index.html, last accessed: 21-06-2011.

M. Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems (2nd ed.)*. Prentice-Hall, Inc., 1999. ISBN 0-13-659707-6.

Rasmus Pagh, Zhewei Wei, Ke Yi, and Qin Zhang. Cache-oblivious hashing. In *Proc. ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems of data (PODS)*, pages 297–304. ACM Press, 2010.

Stratos Papadomanolakis, Debabrata Dash, and Anastasia Ailamaki. Efficient use of the query optimizer for automated physical design. In *Proc. Int'l Conf. Very large data bases (VLDB)*, pages 1093–1104. VLDB Endowment, 2007.

Norman W. Paton and Oscar Díaz. Active database systems. *ACM Comput. Surv.*, 31:63–103, 1999. ISSN 0360-0300.

David A. Patterson and David R. Ditzel. The case for the reduced instruction set computer. *SIGARCH Comput. Archit. News*, 8(6):25–33, 1980. ISSN 0163-5964.

David A. Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A Case for Intelligent RAM. *IEEE Micro*, 17:34–44, 1997. ISSN 0272-1732.

Hasso Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *Proc. Int'l Conf. Management of data (SIGMOD)*, pages 1–2. ACM Press, 2009.

Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005. ISBN 3-540-24372-0.

Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. A case for fractured mirrors. In *Proc. Int'l Conf. Very large data bases (VLDB)*, pages 430–441. VLDB Endowment, 2002.

Marko Rosenmüller and Norbert Siegmund. Automating the Configuration of Multi Software Product Lines. In *Proc. Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 123–130. University of Duisburg-Essen, 2010.

Marko Rosenmüller, Norbert Siegmund, Horst Schirmeier, Julio Sincero, Sven Apel, Thomas Leich, Olaf Spinczyk, and Gunter Saake. FAME-DBMS: tailor-made data management solutions for embedded systems. In *Proc. EDBT workshop on Software engineering for tailor-made data management (SETMDM)*, pages 1–6. ACM Press, 2008.

Marko Rosenmüller, Sven Apel, Thomas Leich, and Gunter Saake. Tailor-made Data Management for Embedded Systems: A Case Study on Berkeley DB. *Data and Knowledge Engineering (DKE)*, 68(12):1493–1512, 2009a.

Marko Rosenmüller, Christian Kästner, Norbert Siegmund, Sagar Sunkle, Sven Apel, Thomas Leich, and Gunter Saake. SQL à la Carte – Toward Tailor-made Data Management. In *13. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW)*, pages 117–136, 2009b.

Marko Rosenmüller, Norbert Siegmund, and Martin Kuhlemann. Improving Reuse of Component Families by Generating Component Hierarchies. In *Proc. Workshop on Feature-oriented Software Development (FOSD)*, pages 57–64. ACM Press, 2010.

Marko Rosenmüller, Norbert Siegmund, Sven Apel, and Gunter Saake. Flexible Feature Binding in Software Product Lines. *Automated Software Engineering – An International Journal*, 18(2):163–197, 2011a.

Marko Rosenmüller, Norbert Siegmund, Thomas Thüm, and Gunter Saake. Multi-Dimensional Variability Modeling. In *Proc. Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 11–20. ACM Press, 2011b.

Gunter Saake, Marko Rosenmüller, Norbert Siegmund, Christian Kästner, and Thomas Leich. Downsizing Data Management for Embedded Systems. *Egyptian Computer Science Journal (ECS)*, 31(1):1–13, 2009.

James H. Sabry, Cynthia L. Adams, Eugeni A. Vaisberg, and Anne M. Crompton. Database system for predictive cellular bioinformatics, United States Patent 6631331, October 2003. URL `http://www.freepatentsonline.com/6631331.html`.

Kai-Uwe Sattler, Ingolf Geist, and Eike Schallehn. QUIET: continuous query-driven index tuning. In *Proc. Int'l Conf. Very large data bases (VLDB)*, pages 1129–1132. VLDB Endowment, 2003.

Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. COLT: continuous on-line tuning. In *Proc. Int'l Conf. Management of data (SIGMOD)*, pages 793–795. ACM Press, 2006.

Sandro Schulze, Sven Apel, and Christian Kästner. Code clones in feature-oriented software product lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 103–112. ACM Press, 2010.

Michael E. Senko, Edward B. Altman, Morton M. Astrahan, and P. L. Fehder. Data structures and accessing in data-base systems: III data representations and the data independent accessing model. *IBM Systems Journal*, 12:64–93, 1973. ISSN 0018-8670.

Dennis Shasha and Philippe Bonnet. *Database tuning: principles, experiments, and troubleshooting techniques*. Morgan Kaufmann Publishers Inc., 2003. ISBN 1-55860-753-6.

Shore. http://www.cs.wisc.edu/shore/, last accessed: 21-06-2011.

Dominik Ślęzak, Jakub Wróblewski, Victoria Eastwood, and Piotr Synak. Brighthouse: an analytic data warehouse for ad-hoc queries. *The VLDB Journal*, 1(2): 1337–1345, 2008.

SQLite. http://www.sqlite.org/, last accessed: 21-06-2011.

Michael Stonebraker and Ugur Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proc. Int'l Conf. Data Engineering (ICDE)*, pages 2–11. IEEE Computer Society, 2005.

Michael Stonebraker, Gerald Held, Eugene Wong, and Peter Kreps. The design and implementation of INGRES. *ACM Trans. Database Syst.*, 1:189–222, 1976. ISSN 0362-5915.

Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel R. Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: a column-oriented DBMS. In *Proc. Int'l Conf. Very large data bases (VLDB)*, pages 553–564. VLDB Endowment, 2005.

Michael Stonebraker, Samuel R. Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proc. Int'l Conf. Very large data bases (VLDB)*, pages 1150–1160. VLDB Endowment, 2007.

Vanessa Stricker, Andreas Metzger, and Klaus Pohl. Avoiding redundant testing in application engineering. In *Proc. Int'l Conf. Software Product Line (SPLC)*, pages 226–240. Springer-Verlag, 2010.

Sagar Sunkle, Martin Kuhlemann, Norbert Siegmund, Marko Rosenmüller, and Gunter Saake. Generating Highly Customizable SQL Parsers. In *Proc. EDBT workshop on Software engineering for tailor-made data management (SETMDM)*, pages 29–33. ACM Press, 2008.

Per Svensson. The Evolution of Vertical Database Architectures — A Historical Review (Keynote Talk). In *Proc. Int'l Conf. Scientific and Statistical Database Management (SSDBM)*, pages 3–5. Springer-Verlag, 2008.

Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2002. ISBN 0201745720.

Aleksandra Tesanovic, Dag Nyström, Jörgen Hansson, and Christer Norström. Towards Aspectual Component-Based Development of Real-Time Systems. In *Proc. Int'l Conf. Real-Time and Embedded Computing Systems and Applications (RTCSA)*, pages 558–577. Springer-Verlag, 2004.

Antti Tevanlinna, Juha Taina, and Raine Kauppinen. Product family testing: a survey. *SIGSOFT Softw. Eng. Notes*, 29:12–12, 2004. ISSN 0163-5948.

Alexander Thiem and Kai-Uwe Sattler. An Integrated Approach to Performance Monitoring for Autonomous Tuning. In *Proc. Int'l Conf. Data Engineering (ICDE)*, pages 1671–1678. IEEE Computer Society, 2009.

Kodama Toshio and Kunii Toshiyasu. Development of new DBMS based on the cellular model-from the viewpoint of a data input. *IEIC Technical Report (Institute of Electronics, Information and Communication Engineers)*, 102(208):97–102, 2002. ISSN 0913-5685.

Kodama Toshio, Kunii Toshiyasu, and Seki Yoichi. A Development of a Cellular DBMS Based on an Incrementally Modular Abstraction Hierarchy. *Joho Shori Gakkai Kenkyu Hokoku*, 2004(45):43–50, 2004. ISSN 0919-6072.

TPC-C. http://www.tpc.org/tpcc/, last accessed: 21-06-2011.

TPC-E. http://www.tpc.org/tpce/, last accessed: 21-06-2011.

TPC-H. http://www.tpc.org/tpch/, last accessed: 21-06-2011.

Eddy Truyen and Wouter Joosen. Towards an aspect-oriented architecture for self-adaptive frameworks. In *Proc. AOSD workshop on Aspects, components, and patterns for infrastructure software (ACP4IS)*, pages 1–8. ACM Press, 2008.

Syed Saif ur Rahman. Using evolving storage structures for data storage. In *Proc. Int'l Conf. Frontiers of Information Technology (FIT)*, pages 3:1–3:6. ACM Press, 2010.

Syed Saif ur Rahman, Azeem Lodhi, and Gunter Saake. Cellular DBMS - Architecture for Biologically-Inspired Customizable Autonomous DBMS. In *Proc. Int'l Conf. Networked Digital Technologies (NDT)*, pages 310–315. IEEE Computer Society, 2009a.

Syed Saif ur Rahman, Marko Rosenmüller, Norbert Siegmund, Gunter Saake, and Sven Apel. Specialized Embedded DBMS: Cell Based Approach. In *Proc. Int'l Workshop on Database and Expert Systems Applications*, pages 9–13. IEEE Computer Society, 2009b.

Syed Saif ur Rahman, Veit Köppen, and Gunter Saake. Cellular DBMS: An Attempt Towards Biologically-Inspired Data Management. *Journal of Digital Information Management*, 8(2):117–128, 2010. ISSN 0972-7272.

Syed Saif ur Rahman, Eike Schallehn, and Gunter Saake. ECOS: Evolutionary Column-Oriented Storage. In *Proc. British National Conf. on Databases (BNCOD)*, page <To appear>. Springer-Verlag, 2011.

Patrick Valduriez, Setrag Khoshafian, and George P. Copeland. Implementation Techniques of Complex Objects. In *Proc. Int'l Conf. Very large data bases (VLDB)*, pages 101–110. Morgan Kaufmann Publishers Inc., 1986.

Valgrind. http://www.valgrind.org, last accessed: 21-06-2011.

Fabrizio Verroca, Carlo Eynard, Giorgio Ghinamo, Gabriele Gentile, Riccardo Arizio, and Mauro D'Andria. A Centralised Cellular Database to Support Network Management Process. In *Proc. Workshops on Data Warehousing and Data Mining*, pages 311–322. Springer-Verlag, 1999.

Gerhard Weikum, Axel Moenkeberg, Christof Hasse, and Peter Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *Proc. Int'l Conf. Very large data bases (VLDB)*, pages 20–31. VLDB Endowment, 2002.

Brent Wilson. Introduction to parallel programming using message-passing. *J. Comput. Small Coll.*, 21:207–211, October 2005. ISSN 1937-4771.

Yong Yao and Johannes Gehrke. The Cougar Approach to In-Network Query Processing in Sensor Networks. *SIGMOD Rec.*, 31(3):9–18, 2002. ISSN 0163-5808.

Pamela Zave. *An experiment in feature engineering*, pages 353–377. Springer-Verlag, 2003. ISBN 0-387-95349-3.

Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. DB2 design advisor: integrated automatic physical database design. In *Proc. Int'l Conf. Very large data bases (VLDB)*, pages 1087–1097. VLDB Endowment, 2004.

Marcin Zukowski, Peter A. Boncz, Niels Nes, and Sándor Héman. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Engineering Bulletin*, 28(2):17–22, 2005.

Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proc. Int'l Conf. Data Engineering (ICDE)*, page 59. IEEE Computer Society, 2006.

Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. Cooperative scans: dynamic bandwidth sharing in a DBMS. In *Proc. Int'l Conf. Very large data bases (VLDB)*, pages 723–734. VLDB Endowment, 2007.

Marcin Zukowski, Niels Nes, and Peter A. Boncz. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *Proc. Int'l Workshop on Data Management on New Hardware (DaMoN)*, pages 47–54. ACM Press, 2008.