



Nr.: FIN-01-2008

**EDBT'08 WORKSHOP ON SOFTWARE
ENGINEERING FOR TAILOR-MADE
DATA MANAGEMENT (PROCEEDINGS)**

Nantes, France, March 29, 2008

Editors:

Sven Apel (Passau)

Marko Rosenmüller (Magdeburg)

Gunter Saake (Magdeburg)

Olaf Spinczyk (Dortmund)



Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Technical Report

Impressum (§ 10 MDStV):

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Gunter Saake
Postfach 4120
39016 Magdeburg
E-Mail: saake@cs.uni-magdeburg.de

<http://www.cs.uni-magdeburg.de/Preprints.html>

Auflage: 70

Redaktionsschluss: Februar 2008

Herstellung: Dezernat Allgemeine Angelegenheiten,
Sachgebiet Reproduktion

Bezug: Universitätsbibliothek/Hochschulschriften- und
Tauschstelle

EDBT'08 Workshop on Software Engineering for Tailor-made Data Management (Proceedings)

Nantes, France, March 29, 2008

Editors: Sven Apel (*University of Passau*)
Marko Rosenmüller (*University of Magdeburg*)
Gunter Saake (*University of Magdeburg*)
Olaf Spinczyk (*Dortmund University of Technology*)

Foreword

Tailor-made data management software (DMS) is not only important in the field of embedded systems. A DMS that incorporates only required functionality bears the potential to strip down the code base and to improve reliability and maintainability. The desire for tailor-made data management solutions is not new: concepts like kernel-systems or component toolkits have been proposed 20 years ago. However, a view on the current practice reveals that nowadays DMS are either monolithic DBMS or special-purpose systems developed from scratch for specific platforms and scenarios, e.g., for embedded systems and sensor networks.

While monolithic DBMS architectures hinder a reasonable and effective long-term evolution and maintenance, special-purpose solutions suffer from the conceptual problem to reinvent the wheel for every platform and scenario or to be too general to be efficient. A mere adaptation of present solutions is impossible from the practical point of view, e.g., it becomes too expensive or simply impractical, which is confirmed by the current practice. Especially in the domain of embedded and realtime systems there are extra requirements on resource consumption, footprint, and execution time. That is, contemporary data management solutions have to be tailorable to the specific properties of the target platform and the requirements and demands made by the stakeholders and the application scenario.

We and others (see, e.g., VLDB'03 Panel "A Database Striptease") noticed that the world of data management is in flux. Computing paradigms such as Ubiquitous and Pervasive Computing and new dimensions of systems such as ultra-large systems (ULS - SEI report 2006) arise at the horizon. To keep track with these developments something really has to change in developing data management solutions. The problem of monolithic software architecture is not exclusive to DBMS. For example, in the domain of operating systems or middleware similar problems occurred. In the last twenty years (especially the last 5 years) researchers in these domains made significant progress in designing software towards customizable and well-structured architectures without sacrificing reasonable performance or resource consumption characteristics. Especially, work on software product lines, components, patterns, features, and aspects is promising in this respect. These techniques should also be applicable to DMS.

The main goal of this workshop is to gather people from different fields related to software engineering and data management to find out who is working on related topics and what is the current state of the art.

Contents

FAME-DBMS: Tailor-made Data Management Solutions for Embedded Systems	1
<i>M. Rosenmüller, N. Siegmund, H. Schirmeier, J. Sincero, S. Apel, T. Leich, O. Spinczyk, G. Saake</i>	
A Relational File System as an Example for Tailor-made DMS	7
<i>K. Koll</i>	
Flexible Transaction Processing in the Argos Middleware	12
<i>A. Arntsen, M. Mortensen, R. Karlsen, A. Andersen, A. Munch-Ellingsen</i>	
Tailor-made Lock Protocols and their DBMS Integration	18
<i>S. Bächle, T. Härder</i>	
Database Servers Tailored to Improve Energy Efficiency	24
<i>G. Graefe</i>	
Generating Highly Customizable SQL Parsers	29
<i>S. Sunkle, M. Kuhleemann, N. Siegmund, M. Rosenmüller, G. Saake</i>	
Architectural Concerns for Flexible Data Management	35
<i>I. E. Subasu, P. Ziegler, K. R. Dittrich, H. Gall</i>	
A New Approach to Modular Database Systems	41
<i>F. Irmert, M. Daum, K. Meyer-Wegener</i>	

FAME-DBMS: Tailor-made Data Management Solutions for Embedded Systems

Marko Rosenmüller¹, Norbert Siegmund¹, Horst Schirmeier²,
Julio Sincero³, Sven Apel⁴, Thomas Leich⁵, Olaf Spinczyk², Gunter Saake¹

¹University of Magdeburg, {rosenmue, nsiegmun, saake}@ovgu.de

²Dortmund University of Technology, {horst.schirmeier, olaf.spinczyk}@tu-dortmund.de

³University of Erlangen-Nuremberg, sincero@cs.fau.de

⁴University of Passau, apel@uni-passau.de

⁵METOP Research Institute, thomas.leich@metop.de

ABSTRACT

Data management functionality is not only needed in large-scale server systems, but also in embedded systems. Resource restrictions and heterogeneity of hardware, however, complicate the development of data management solutions for those systems. In current practice, this typically leads to the redevelopment of data management because existing solutions cannot be reused and adapted appropriately. In this paper, we present our ongoing work on FAME-DBMS, a research project that explores techniques to implement highly customizable data management solutions, and illustrate how such systems can be created with a software product line approach. With this approach a concrete instance of a DBMS is derived by composing features of the DBMS product line that are needed for a specific application scenario. This product derivation process is getting complex if a large number of features is available. Furthermore, in embedded systems also non-functional properties, e.g., memory consumption, have to be considered when creating a DBMS instance. To simplify the derivation process we present approaches for its automation.

1. INTRODUCTION

Traditionally, research on data management software is discussed in the context of large-scale *database management systems (DBMS)* like Oracle, IBM DB2 or Microsoft SQL Server. In recent years, data management has also been shown increasingly important for embedded systems [17]. Embedded systems are used as control units in cars, cell phones, washing machines, TV sets, and many other devices of daily use. Visions of pervasive and ubiquitous computing [26] and smart dust [25] emphasize the importance of embedded systems for the future. Two factors make these systems special and challenging for data management: First, embedded devices have usually restricted computing power and memory in order to minimize production costs and energy consumption. Second, embedded systems are strongly

heterogeneous, meaning that most systems differ in software and hardware. Software for these systems is usually implemented specifically for a single system and a special application scenario.

For new application scenarios data management is often reinvented to satisfy resource restrictions, new requirements, and rapidly changing hardware [7]. This practice leads to an increased time to market, high development costs, and poor quality of software [9, 15]. A general data management infrastructure could avoid this by separating data management and application logic [13]. Considering the limited resources and special requirements on data management, traditional DBMS are not suited for embedded environments [23, 6].

In this paper, we present our ongoing work on the FAME-DBMS research project¹. Our goal is to develop, extend, and evaluate techniques and tools to implement and customize DBMS. Such techniques have to account for the special requirements of embedded systems. For that, we employ the *software product line (SPL)* approach based on static composition of features. With an SPL approach and techniques that enable to modularize also crosscutting features we can attain high variability which is needed for embedded systems. However, variability also increases the configuration space (i.e., the number of possible variants of a DBMS) and requires assistance to derive and optimize a concrete DBMS. We present two techniques to partially automate this *product derivation* process. First, to identify features of a DBMS SPL, needed for a particular client application, we use static program analysis. Second, we propose to partially automate the configuration process by using *non-functional constraints*, i.e., constraints that are used to restrict the non-functional properties of a DBMS like performance or memory consumption. We present first results and show what problems arise and what challenges are still ahead.

¹The project is funded by German Research Foundation (DFG), Projects SA 465/32-1 and SP 968/2-1. <http://www.fame-dbms.org/>

2. TAILOR-MADE DATA MANAGEMENT

Resource constraints and a diversity in hardware of embedded systems forces developers to tailor software to a large number of application scenarios. Data management is needed in embedded systems but often reimplemented because existing solutions lack customizability. SPLs enable to develop software that can be tailored to different use cases with minimized development effort. This technology should also be applicable to tailor data management solutions for embedded systems.

2.1 Software Product Lines

Developing software that contains only and exactly the functionality required can be achieved using an SPL. Existing implementation techniques like *components* are inappropriate to support fine-grained customizability also of crosscutting concerns if they are used in isolation [17]. Using static composition, e.g., based on C/C++ preprocessor statements, achieves high customizability while not affecting performance. However, C/C++ preprocessor statements degrade readability and maintainability of source code [22, 5].

To avoid such problems new techniques – that are applicable to embedded systems – have to be explored and developed. In the FAME-DBMS project, we study *aspect-oriented programming (AOP)* [14] and *feature-oriented programming (FOP)* [3, 18] for implementing SPLs. In contrast to components, AOP and FOP also support modularization of crosscutting concerns. By using *AspectC++*² and *FeatureC++*³, both language extensions of C++, we are able to use these paradigms for embedded systems.

Here we concentrate on FOP and FeatureC++, however, most of the presented concepts also apply to AOP. Using FOP, features are implemented as increments in functionality of a base program [3]. For *product derivation* (i.e., creating a concrete instance of a product line) a base program has to be composed with a set of features. This results in a number of different variants of an application.

2.2 Downsizing Data Management

There are solutions to apply the SPL approach to data management software. One possibility is to design a DBMS product line from scratch, starting with domain analysis and implementing and testing its features. Alternatively, instead of beginning from scratch, one can refactor existing data management systems, e.g., using FOP. When starting with existing, tested, and optimized data management systems and incrementally detaching features to introduce variability this results in a stripped-down version that contains only the core functionality. Additional features can be added when required for a certain use case. This approach, also known as *extractive adoption* [8], reduces the required effort and risk which makes it especially attractive for companies that want

to adopt SPL technology for their products. In the FAME-DBMS project, we chose this approach to compare downsized versions with the original application which makes it useful as a research benchmark.

In a non-trivial case study we refactored the C version of the embedded database engine Berkeley DB into features. The footprint of Berkeley DB is fairly small (484 KB) and there are already a few static configuration options available. Even though, it is still too large for deeply embedded devices and contains several features like TRANSACTIONS that might not be required in all use cases. Therefore, we transformed the Berkeley DB code from C to C++ and then refactored it into features using FeatureC++. We used behavior preserving refactorings to maintain performance and to avoid errors.

Our case study has shown that the transformation from C to FeatureC++ (1) has no negative impact on performance or resource consumption, (2) successfully increases customizability (24 optional features) so that we are able to create far more variants that are specifically tailored to a use case, and (3) successfully decreases binary size by removing unneeded functionality to satisfy the tight memory limitations of small embedded systems.

The results are summarized in Figure 1. Before refactoring, the binary size of Berkeley DB embedded into a benchmark application was between about 400 and 650 KB, depending on the configuration (1–6). After transformation from C to FeatureC++, we could slightly decrease the binary size (Figure 1a) while maintaining the original performance (Figure 1b)⁴. By extracting additional features that were not already customizable with preprocessor statements we are able to derive further configurations. These are even smaller and faster if those additional features are not required in a certain use case (Configurations 7 and 8 in Figure 1). This illustrates the practical relevance of downsizing data management for embedded systems.

2.3 FAME-DBMS

Decomposing Berkeley DB showed us that FOP and FeatureC++ are appropriate for implementing DBMS for embedded systems. We could furthermore show that a fine granularity, also of core functionality like the storage management, can be achieved using FOP [16]. However, when decomposing Berkeley DB we recognized that it is a complex task to decompose an existing DBMS that is not designed for a fine-grained decomposition. Thus, further decomposition of Berkeley DB was not possible in reasonable time because remaining functionality was heavily entangled. For example, decomposition of the B-tree index functionality is hardly possible without reimplementing large parts of it.

We argue that a DBMS SPL for embedded systems has to be designed with an adequate granularity. This means

²<http://www.aspectc.org/>

³http://www.witi.cs.uni-magdeburg.de/iti_db/fcc/

⁴In Figure 1b Configuration 8 was omitted since it uses a different index structure and cannot be compared to Configurations 1–7.

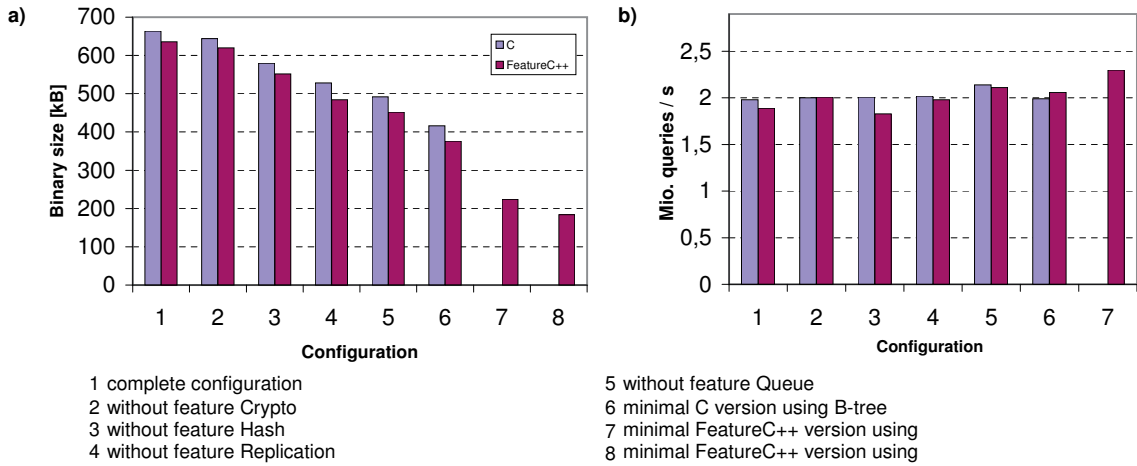


Figure 1: Binary size and performance of different C and FeatureC++ variants of Berkeley DB.

that different levels of granularities have to be supported: Functionality used in highly resource constrained environments should be decomposed with a fine granularity to ensure high variability. In contrast, functionality that is only used in larger systems, where resources are not highly limited, may not be further decomposed or should only be decomposed with a coarse granularity to avoid increasing complexity. Thus, the granularity is the key for a trade-off between complexity and variability.

Another reason for decomposing a DBMS into features is to impose a clean structure (e.g., for source code, documentation, etc.). However, since a decomposition can increase the complexity the benefit of such a decomposition has to be estimated. This is not the case for features that are only included to aggregate already decomposed features and thus do not further increase the complexity.

To analyze the granularity of an appropriate DBMS decomposition in more detail, we are currently implementing a DBMS product line from scratch. The decomposition of this FAME-DBMS prototype is depicted in Figure 2. While we are using a fine-grained decomposition for features that are also used in small systems (e.g., the B-tree index) we use a coarse granularity for features like TRANSACTION which is decomposed into a small number of features (e.g., alternative commit protocols). To further structure the DBMS product line aggregating features are used. For example, feature STORAGE aggregates different features but does not provide own functionality. Using FOP this structuring can be applied to all software artifacts including the source code.

3. AUTOMATED PRODUCT DERIVATION

Fine granularity of a decomposition of a DBMS is important to achieve customizability but it also increases the development effort. Furthermore, the product derivation process is getting more complex if there is a large number of features. The developer of an application that uses a DBMS has to tailor the DBMS for the specific needs of his applica-

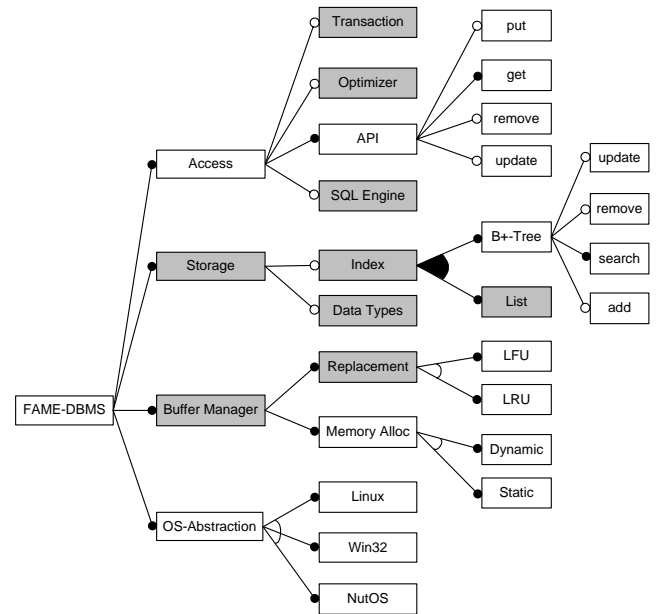


Figure 2: Feature diagram of the FAME-DBMS prototype. Gray features have further subfeatures that are not displayed.

tion which imposes significant configuration effort. Furthermore, the application developer needs detailed knowledge of the DBMS domain for this configuration process. Thus an automated product derivation is desirable.

Another important issue in embedded systems are *non-functional properties (NFPs)* of a generated DBMS instance. Often these are of interest to the stakeholder but cannot be configured directly. For example, a developer wants to tailor the functionality of a DBMS product line for his or her application, but also has to stay within the resource constraints of a given embedded device with fixed RAM and ROM size. Other NFPs like performance or response time are also very

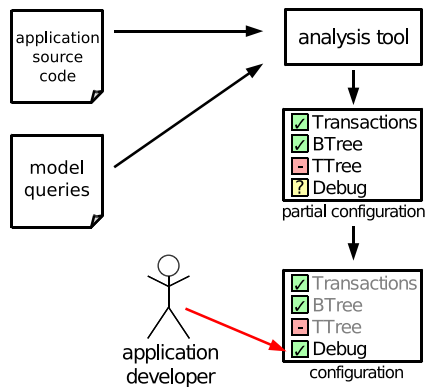


Figure 3: Automated detection of needed features with the analysis tool.

important in the embedded domain. Thus NFPs should also be considered in the product derivation.

In the FAME project we aim at improving tool support for product derivation. We address the configuration complexity by an approach that partially automates the configuration process using the functional requirements of a client application on a DBMS and furthermore integrate non-functional constraints.

3.1 Functional Requirements

When developing client applications that use a DBMS SPL the inherent *uses* relationship between application (e.g., a personal calendar application) and DBMS suggests to derive the need for DBMS features from the application itself.

We developed an analysis tool (see Figure 3) which automatically detects an application’s need for infrastructure features by analyzing the C++ sources of the application [19]. For example, it would be beneficial to detect the applications’s need for the feature JOIN of a DBMS to remove this functionality if it is not used. First, we statically analyze the application’s sources which results in an application model (a control flow graph with additional data flow and type information), abstracting from syntactic details in the source code. Infrastructure features that are suitable for automatic detection can be associated with queries on the application model (*model queries* in Figure 3). These queries answer the question whether the application needs a particular feature. For example, in an application that uses Berkeley DB as a database infrastructure, a certain flag combination used to open a database environment indicates the need for the feature TRANSACTION, which can be formulated as one of the abovementioned queries.

The result of this process is a list of features that need to be included in the DBMS that the application is using. This list can be further refined by analyzing constraints between features of an SPL that are part of the feature model of that application. Ideally, large parts of a feature diagram can be configured automatically. The developer has to manually select only features that cannot be derived from the applica-

tion’s sources.

An evaluation of the approach with the refactored Berkeley DB (cf. Section 2.2) confirmed the assumption that the need for infrastructure features can be derived from the application sources in most cases. Our experiments with a benchmark application (that uses Berkeley DB) showed that 15 of 18 examined Berkeley DB features can be derived automatically from the application’s source code; only 3 of 18 features were generally not derivable, because they are not involved in any infrastructure API usage within any application.

3.2 Non-functional Properties

As already stated also NFPs of an SPL are important for embedded systems. For example, binary size and memory consumption are critical for resource constrained environments and should also be considered in the product derivation process. To allow control over those properties our objective is to further automate product derivation. Using *non-functional constraints* we can exclude variants of an SPL that do not fulfill these constraints.

We support this by (1) measuring information about NFPs of concrete DBMS and (2) assisting or automating the selection of features based on measured NFPs and user defined constraints. To achieve these goals, our idea is to store as much information as possible about generated products in the model describing the SPL. This data is used to assist the derivation of further products.

The ideas are part of the *Feedback Approach* [21], which enables the application engineer to perform analysis (both static and dynamic) on generated products so that knowledge about specific NFPs can be obtained. This information can be assigned to a complete product specification (*configuration properties*), to a specific product feature (*feature properties*), or to implementation units, e.g., classes, aspects, or components (*component properties*).

The result of the analysis is stored to be used during product derivation to obtain the NFPs of a new product that is to be derived. This can be based on a calculation of an optimal configuration using the properties assigned to features or by estimating the properties based on heuristics (e.g., by using similarities between the product to be derived and earlier created instances). Calculating optimal solutions based on constraints is known as the *constraint satisfaction problem (CSP)* that belongs to the complexity class *NP-complete*. Currently we are using a greedy algorithm to calculate optimal solutions to cope with the complexity of the problem. Furthermore, we can give hints to the user what the properties of a configured instance will be by using information about already instantiated products.

We think that calculating an optimal solution has to be based on both: Properties assigned to features and properties of concrete instances. First, a greedy algorithm can be used to derive promising product configurations using feature properties. In the second step, more accurate values

for non-functional properties can be obtained by including heuristics and information about already instantiated products and components. An optimal solution is selected by comparing these corrected values.

Our work on NFPs is at an early stage, however, our preliminary results are promising. We have shown the feasibility of the idea for simple NFPs like *code size* [21] and are developing heuristics and analysis components to address *performance* of SPL instances.

4. RELATED WORK

Development of customizable data management software has been in the focus of research for several years. Batory and Thomas used code generation techniques to customize DBMS [4]. They aimed at creating special language extensions, e.g., to ease the use of cursors. As one of the origins of FOP, Batory et al. focused on customizing DBMS with Genesis [2]. In contrast to FOP as it is known today, it was not based on OOP and its complexity decreased usability. There have been many other developments to support extensibility of DBMS in the last 20 years. These approaches found their way into current DBMS but cannot provide appropriate customizability to support embedded systems (e.g., Kernel Systems). Additionally, detailed knowledge is often needed to implement a concrete DBMS or extend existing ones [11]. Component-based approaches are getting popular for traditional DBMS [11, 7, 17]. These, however, introduce a communication overhead that degrades performance and increases memory consumption. Furthermore, limited customizability because of crosscutting concerns does not allow for fine-grained customization. To overcome this limitation Nyström et al. developed a component-based approach named COMET that uses AOP to tailor the components [17]. In another approach, Tešanović et al. examined AOP for DBMS customization [24]. They evaluated their approach using Berkeley DB and showed customizability for small parts of the system. Both approaches (and there is no other approach that we are aware of) could not show concrete implementations of a complete customizable DBMS nor detailed evaluations. Other approaches like PicoDBMS [6] or DELite [20] concentrate on special requirements on data management for resource constrained environments and not on customizable solutions.

There is less research on automated tailoring of DBMS, infrastructure SPLs, with their special relationship to applications built on top of them. Fröhlich takes this relationship into account and aims at automatic configuration [12]. In this approach the set of infrastructure symbols referenced by the application determines which product line variant is needed. Apart from the comparably simple static analysis the main difference to our approach is the lack of logical isolation between analysis and configuration, established by a feature model.

NFPs of SPLs are getting more into the focus of current research. Cysneiros et al. propose the modeling of NFPs

during application engineering [10]. This approach considers required non-functional behavior and adds it to design documentation in order to make the non-functional properties traceable. Bass et al. also address NFPs during software architecture design [1] to relate the NFPs to the system's architecture. Since these techniques tackle the same problem in a different stage of development, we see our work as a complementary approach and believe in the synergy between them.

5. CONCLUSION

We illustrated that FOP can be used to develop tailor-made data management solutions also for embedded systems. We applied an extractive approach to an existing DBMS and thereby could show that FOP has no negative impact on performance. We also presented our current work on the FAME-DBMS product line implemented using FOP and a mixed granularity for decomposition. The resulting high customizability is needed for embedded devices but increases the configuration space, rendering manual configuration complex and error-prone. Addressing this increased complexity, we presented two approaches that help simplifying variant selection by partially automating the configuration process and by providing non-functional properties for product derivation. Although we need to extend the approach and need further evaluation, first results are very promising.

In future work, we plan to create complete tool support that allows to develop SPLs optimized for embedded systems. As already outlined, we will continue working on tailor-made DBMS and plan to extend SPL composition and optimization to cover multiple SPLs (e.g., including the operating system and client applications) to optimize the software of an embedded system as a whole. Furthermore, we think that knowledge about the application domain has to be included in the product derivation process to automatically tailor the DBMS with respect to a concrete application scenario. For example, the data that is to be stored could be considered to statically select the optimal index.

Acknowledgments

Marko Rosenmüller and Norbert Siegmund are funded by German Research Foundation (DFG), Project SA 465/32-1, Horst Schirmeier and Julio Sincero by Project SP 968/2-1. The presented work is part of the FAME-DBMS project⁵, a cooperation of Universities of Dortmund, Erlangen-Nuremberg, Magdeburg, and Passau, funded by DFG.

6. REFERENCES

- [1] L. J. Bass, M. Klein, and F. Bachmann. Quality Attribute Design Primitives and the Attribute Driven Design Method. In *Revised Papers from the*

⁵<http://www.fame-dbms.org/>

- International Workshop on Software Product-Family Engineering*, pages 169–186. Springer-Verlag, 2002.
- [2] D. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise. GENESIS: An Extensible Database Management System. *IEEE Transactions on Software Engineering*, 14(11):1711–1730, 1988.
- [3] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.
- [4] D. Batory and J. Thomas. P2: A Lightweight DBMS Generator. *Journal of Intelligent Information Systems*, 9(2):107–123, 1997.
- [5] I. D. Baxter and M. Mehlich. Preprocessor Conditional Removal by Simple Partial Evaluation. In *Proceedings of the Working Conference on Reverse Engineering*, pages 281–290. IEEE Computer Society Press, 2001.
- [6] C. Bobineau, L. Bouganim, P. Pucheral, and P. Valduriez. PicoDMBS: Scaling Down Database Techniques for the Smartcard. In *Proceedings of the International Conference on Very Large Data Bases*, pages 11–20. Morgan Kaufmann, 2000.
- [7] S. Chaudhuri and G. Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In *Proceedings of the International Conference on Very Large Data Bases*, pages 1–10. Morgan Kaufmann, 2000.
- [8] P. Clements and C. Krueger. Point/Counterpoint: Being Proactive Pays Off/Eliminating the Adoption Barrier. *IEEE Software*, 19(4):28–31, 2002.
- [9] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.
- [10] L. M. Cysneiros and J. C. S. do Prado Leite. Nonfunctional Requirements: From Elicitation to Conceptual Models. *IEEE Transactions on Software Engineering*, 30(5):328–350, 2004.
- [11] K. R. Dittrich and A. Geppert. Component Database Systems: Introduction, Foundations, and Overview. In *Component Database Systems*, pages 1–28. dpunkt.Verlag, 2001.
- [12] A. Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, 2001.
- [13] T. Härder. DBMS Architecture – Still an Open Problem. In *Datenbanksysteme in Business, Technologie und Web*, pages 2–28. Gesellschaft für Informatik (GI), 2005.
- [14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 220–242. Springer-Verlag, 1997.
- [15] C. W. Krueger. New Methods in Software Product Line Practice. *Communications of the ACM*, 49(12):37–40, 2006.
- [16] T. Leich, S. Apel, and G. Saake. Using Step-Wise Refinement to Build a Flexible Lightweight Storage Manager. In *Proceedings of the East-European Conference on Advances in Databases and Information Systems*, pages 324–337. Springer-Verlag, 2005.
- [17] D. Nyström, A. Tešanović, M. Nolin, C. Norström, and J. Hansson. COMET: A Component-Based Real-Time Database for Automotive Systems. In *Proceedings of the Workshop on Software Engineering for Automotive Systems*, pages 1–8. IEEE Computer Society Press, 2004.
- [18] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer-Verlag, 1997.
- [19] H. Schirmeier and O. Spinczyk. Tailoring Infrastructure Software Product Lines by Static Application Analysis. In *Proceedings of the International Software Product Line Conference*, pages 255–260. IEEE Computer Society Press, 2007.
- [20] R. Sen and K. Ramamritham. Efficient Data Management on Lightweight Computing Devices. In *Proceedings of the International Conference on Data Engineering*, pages 419–420. IEEE Computer Society Press, 2005.
- [21] J. Sincero, O. Spinczyk, and W. Schröder-Preikschat. On the Configuration of Non-Functional Properties in Software Product Lines. In *Proceedings of the Software Product Line Conference, Doctoral Symposium*. Kindai Kagaku Sha Co. Ltd., 2007.
- [22] H. Spencer and G. Collyer. Ifdef Considered Harmful, or Portability Experience With C News. In *Proceedings of the USENIX Summer 1992 Technical Conference*, pages 185–197, 1992.
- [23] M. Stonebraker and U. Cetintemel. One Size Fits All: An Idea Whose Time Has Come and Gone. In *Proceedings of the International Conference on Data Engineering*, pages 2–11. IEEE Computer Society Press, 2005.
- [24] A. Tešanović, K. Sheng, and J. Hansson. Application-Tailored Database Systems: A Case of Aspects in an Embedded Database. In *Proceedings of International Database Engineering and Applications Symposium*, pages 291–301. IEEE Computer Society Press, 2004.
- [25] B. Warneke, M. Last, B. Liebowitz, and K. S. J. Pister. Smart Dust: Communicating with a Cubic-Millimeter Computer. *Computer*, 34(1):44–51, 2001.
- [26] M. Weiser. Some Computer Science Issues in Ubiquitous Computing. *Communications of the ACM*, 36(7):75–84, 1993.

A relational file system as an example for tailor-made DMS

Konstantin Koll
University of Dortmund, Germany
Computer Science I
D-44227 Dortmund
+49-231-7948786
koll@ls1.cs.uni-dortmund.de

ABSTRACT

This paper motivates the development of tailor-made data management software by presenting a relational file system that has been written from scratch to contain only the database functionality absolutely required for operation. In this paper, the background of the file system is briefly described, and it is stated why an off-the-shelf database system was not suitable for its major components. This application serves as a model for what highly configurable data management software should be able to deliver. If such a system was available at the time of development, a large amount of work would have been saved.

1. INTRODUCTION

Today, computers are widely used to store multimedia files. Among others, digital video, music and images are prominent examples [10]. Unfortunately, file systems have not evolved to accommodate the need for navigation through thousands of files: users are still forced to create a directory hierarchy for their files. This hierarchical scheme implicitly sorts files to certain criteria, e.g. events, artists or projects. Users have to store their files in the proper location then. This is very inflexible, as a file can reside in only one folder, and thus appear in only one single context (unless links or copies are manually created by the user). Even worse, modern file formats also contain metadata attributes besides their actual "payload data": many MP3 files save the artist's name inside an ID3 tag [13], and almost every digital still image camera creates JPEG files with an Exif tag [9]. Since these attributes are not stored in the data structures of the file system, but reside inside the file body (see figure 1), they can only be accessed by software that is aware of them, thus waiving potential benefits for file access on the file system level. The LCARS file system [2] presented in this paper has been created to tackle this problem.

1.1 Previous work

To improve the situation described above, various approaches exist. Because file metadata resembles structured data that can be managed by the means of relational algebra, many solutions incorporate light-weight or even fully-featured relational database systems to store file attributes. A very prominent example for this is the now discontinued Microsoft WinFS [12] that is built on top of the Microsoft SQL server and was supposed to ship with Windows Vista. The academic community relies on similar techniques (e.g. [15]), among them a user-serviced keyword database [4]. Other products, mainly so called "desktop search engines" created by major search engine vendors like Google Desktop [3] or Apple Spotlight [1], use a full-text index.

1.2 Contribution

This paper presents the architecture of the LCARS file system [2], which strives to combine both file systems and relational database technology. Its key components are well-defined and registered file formats, different applications that handle these file formats, and domains that deliver physical storage and provide an indexing/retrieval system for file attributes. Each of these components comes with their own requirements in terms of interoperability, reliability or performance.

Due to this special application of database technology and its deployment in a file system context, the requirements are hard to meet for current off-the-shelf systems. Pointing out these difficulties will hopefully lead to database systems that are more flexible and thus more suitable for similar applications that fall "outside the box".

2. ARCHITECTURE OVERVIEW

Figure 2 shows the architecture of the LCARS file system along with its key components. LCARS is the acronym for "library with computer assisted retrieval system". The file system is closely tied to an operating system [2] and is implemented as an augmented semantic file system [14]. The term "augmented" means that the functionality of a traditional ("physical") file system is extended by another layer of software (i.e. the semantic file system) that is placed between user applications and the "real" file system — very much like many existing solutions [1,3,4,12,15].

The augmented approach to developing a new file system comes with obvious benefits. One of them is the reduced workload during development as there is no need to create the basic file system operations like block access and directory structures from scratch. Another benefit is the interoperability: the augmented file system can work on top of almost any known file system (FAT, NTFS, ext2, ...) and is not tied to a specific platform, given that no special features of a certain physical file system are used.

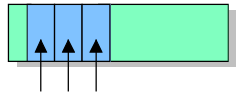
Name	Last access	Size	File body
X.JPG	01/24/2008	89237	 Additional metadata

Figure 1. File, with additional attributes inside the file body.

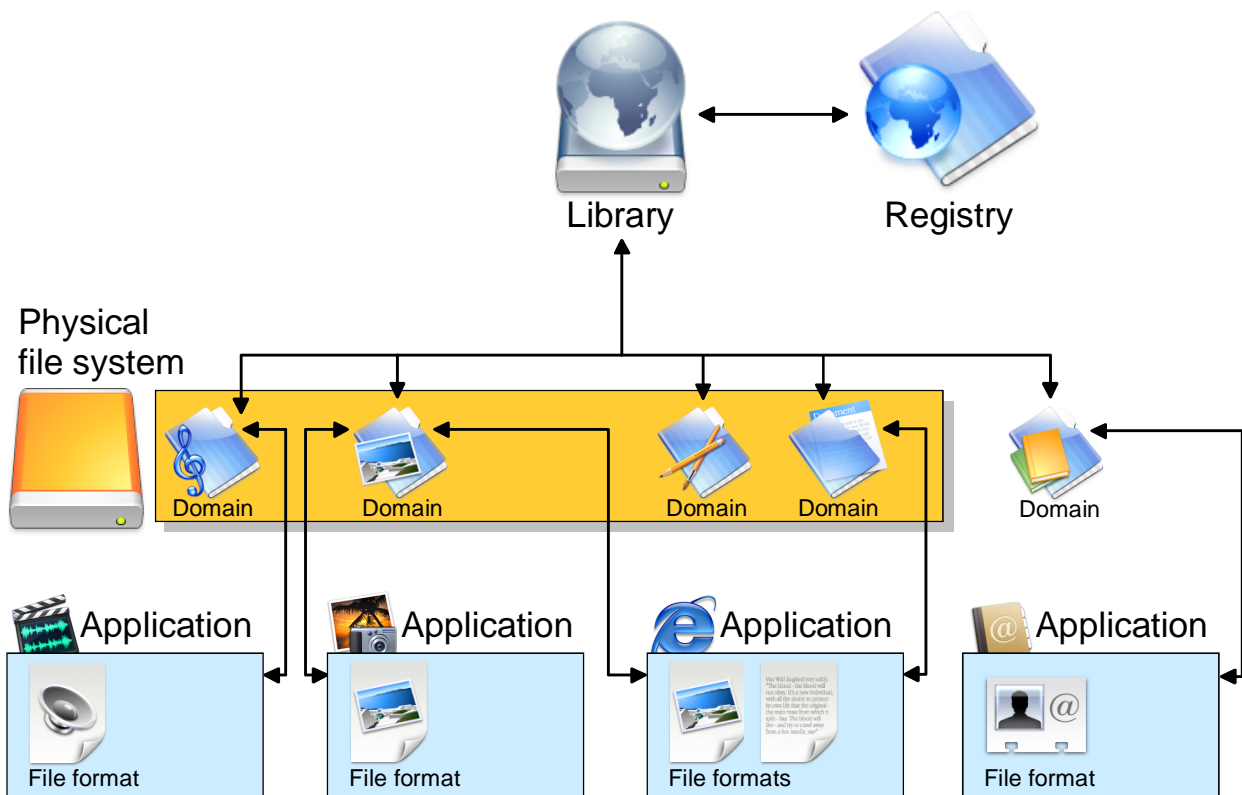


Figure 2. LCARS file system architecture overview.

2.1 File formats

Each file that can be accessed by the semantic file system layer must adhere to a format specification, hence all files are strictly typed. File formats introduce certain attributes to the overall data space. These attributes may be derived from both the physical file system (e.g. file creation date) and the file body (metadata attributes). The LCARS file system is completely oblivious to the attributes' origin.

The type of a file may be identified by anything from a number, a three letter "extension" to the file name like JPG or PDF, a FourCC, a GUID [5], or a MIME-Type string. Their properties (e.g. the attributes they contain) are stored in an entity called "registry". Unregistered file types (e.g. system files for internal use) can only be processed through the traditional API of the physical file system, but not by the means of the semantic file system.

2.2 Applications

Obtaining file attributes is a common problem for any software that relies on metadata. In the LCARS scheme of things, it becomes the applications' responsibility to deliver all public attributes of the file formats under their control to the file system through a dedicated API. This is achieved by starting an application's executable file with a special parameter that causes it to silently load the files in question, extract their attributes, and return them to the indexer.

Using the respective applications to gather file attributes is only possible because the LCARS file system is closely tied to the

operating system, which in turn provides the API functions needed.

Existing solutions rely on implementing file formats that are considered important from scratch, or on third-party plug-ins to integrate a new file type. However, it is clearly more elegant to use the application itself that has to implement the file format anyway for loading and saving.

Extracting file metadata is beyond the usual scope of a database system. Today, using an off-the-shelf system for this task requires some kind of frontend application or middleware that feeds tuples to the database. Basically, many existing desktop search engines and comparable products are such frontend applications. For the specific tasks that come with a semantic file system, a highly configurable data management software should provide at least the same kind of extensibility than the dedicated API described above.

2.3 Domains

The LCARS semantic file system is comprised of so-called "domains". These entities link the file system to applications and their file formats, and also provide physical storage for files.

2.3.1 Physical storage

The LCARS file system completely virtualizes physical storage to the operating system. Files and their attributes are presented to the semantic file system layer regardless of their origin. This allows to mount various storage locations directly into the file system, very much like ODBC works for different databases systems. The most

common occurrence are domains that reflect directories of the physical file system (e.g. the user's home directory).

Some applications manage fully structured data, e.g. an address book or a calendar application that keeps track of appointments. There are standard file formats for these tasks, like the visit card file (.VCF) [6] or the iCalendar format (.ICS) [7]. The size of such files is usually very small (in the range on 1 KB or even less), so the entirety of all appointments or the entire address book is distributed among a large number of tiny files. This decreases the retrieval performance on these files (e.g. printing all addresses), and also wastes disc space due to the minimum allocation size of file systems (up to 64 KB in extreme cases). In conclusion, physical file systems are poorly outfitted for small files of fully structured data.

Virtualizing the physical storage of files through domains allows to tackle this issue: special domains for fully structured data easily store data in a relational database instead of solitary files. Tables containing structured data are subsequently stored as one file per table in the physical file system. In this case, domains become a middleware between applications and database systems.

Applications typically perform file I/O through a set of API functions provided by the operating system. When using the LCARS file system, this API is provided by the different domains instead, making the actual physical storage completely transparent not only to the semantic file system layer, but also to applications.

So-called "virtual domains" are a special case, as they do not use the underlying physical file system at all (rightmost domain in figure 2). They are used to present other information as files, just like the Unix /proc file system [11].

2.3.2 File retrieval

Physical data storage in the LCARS file system is a task for domains as described in the section above. Since no other modules have got access to the physical storage, file operations also have to be carried out by domains — including file retrieval according to attribute values (i.e. searching for files with certain properties).

During file retrieval, query optimization ensures that only those domains are affected that manage files potentially fulfilling the query. For example, if all files with a width larger than 1024 pixels are desired, a domain serving address files only does not need to perform a search since addresses have neither a width nor a resolution.

3. INDEXING METADATA

Since file retrieval is a task for domains in the LCARS file system, indexing of file attributes is also performed by them. Domains face additional problems that have been tackled with a custom indexing method.

3.1 File system environment

A major difference between professional database systems and the tailor-made database functionality in the LCARS file system is the storage of data: in a professional environment (i.e. on a dedicated server), the data management system has got immediate "raw" access to all sectors of a disc, without any file system. This is, among many other advantages, very beneficial for storing tree structures: the nodes of a tree can be linked by simply storing sector numbers inside the linking nodes. A sector containing a node can be read in $O(1)$.

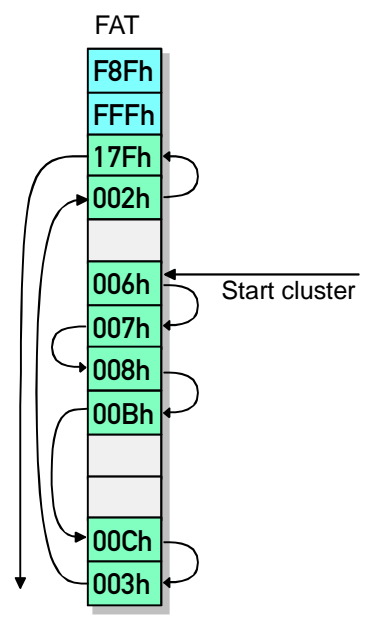


Figure 3. The file allocation table in the FAT file system resembles a linked list that has to be processed in $O(n)$ to access a certain offset inside a file.

When a physical file system is present, the index has to be stored in a file since raw disc access bypassing the file system is prohibited. Because the index has no notion of sector numbers any more, tree nodes are usually linked by storing the offset inside the index file instead of a sector number.

In addition to physically access a certain disc sector, the number of that sector has to be determined from the offset inside the file by the means of the file system. The performance of this operation obviously depends on the actual file system used and its data structures.

A simple example is the FAT file system which seems outdated, but is still widely used today by digital cameras [8], by flash drives and by MP3 players for compatibility reasons. The FAT file system manages a data structure called "file allocation table", which is basically a list of numbers; each block is assigned one entry in this list (see figure 3). An entry contains either the number of the subsequent block, or a special code to mark, among other conditions, the end of the file (i.e. no successor). Seeking to a certain file offset requires the operating system to process this list, starting with the first block of the file, and then continuing until the offset is reached.

Obviously, processing the FAT requires linear time. This factor has to be applied whenever a seek occurs, which has a devastating effect on the performance of tree structures. Assume a B-tree with a height of $O(\log n)$: when stored inside a file in the FAT file system, traversing the B-tree from its root to a leaf node will suddenly perform in $O(n \log n)$, not in $O(\log n)$.

More sophisticated file systems like NTFS or ext2 use more elaborate data structures which perform better. NTFS uses a tree to keep track of blocks allocated by a file, while ext2 relies on a structure called "I-Node". They impose an additional factor of $O(\log n)$ or $O(1)$ respectively to any seek that occurs.

However, a penalty for any seek operation remains, so it is apparent why database systems preferably run on raw drives without a file system, and why many operating systems require a special swap partition outside the file system for virtual memory. The augmented approach of the LCARS file system has the drawback that it is forced to run on top of an existing physical file system, so raw disc access or a special partition are no viable options in this case.

3.2 Partially populated data space

Another property of the file system environment, and more specifically file metadata, is the distribution of file objects in the overall data space. Each attribute to be indexed can be considered a dimension of the data space, so d attributes would form a d -dimensional hypercube: $\Omega = A_1 \times \dots \times A_d$ if all attributes are mapped and normalized to the range of $[0 \dots 1]$.

However, the data space formed by file metadata is usually not a d -dimensional hypercube; in fact, it is just a subset. The hypercubic data space allows that each point inside the hypercube can be populated, i.e. get files assigned. This assumes that all attributes are present in all files and are independent of each other. But this is not true for files, because certain attributes are present in some file formats, but not in others. An example for this is the width and height: both are present in images and video files, but not in audio files. From a geometric perspective, this renders certain regions in the hypercube void, i.e. they cannot be populated by any item.

Figure 4 shows an example of such a sparse data space. Let there be two file types, one containing the attributes A_1 and A_3 , and the second one containing A_2 and A_3 . The resulting hypercube would be $\Omega = A_1 \times A_2 \times A_3$. Since no file contains both A_1 and A_2 , the real metadata space consists only of the two surfaces $A_1 \times A_3$ and $A_2 \times A_3$.

The resulting data space can be described for arbitrary file metadata. Let there be k different file formats. In addition to the n attributes A_1 to A_n that shall be common for all file formats, each file type j shall have m additional attributes named $B_{j,1}$ to $B_{j,m}$. Then the partially populated metadata space can be defined as:

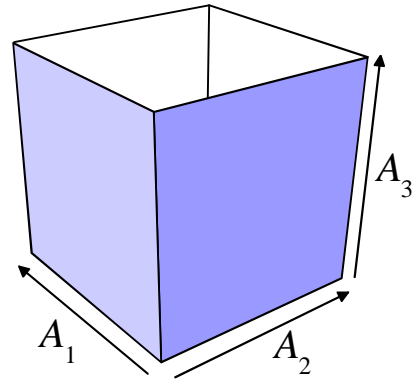


Figure 4. A hypercubic data space, with data objects populating the filled surfaces only.

$$\Omega = A_1 \times \dots \times A_n \times \bigcup_{j=1}^k B_{j,1} \times \dots \times B_{j,m_j} \quad (1)$$

The index structure developed for the LCARS file system takes advantage of this sparsity by using a "master index" for the attributes $A_1 \dots A_n$, and one additional "slave index" for each of the k file formats to hold their attributes $B_{j,1} \dots B_{j,m}$. This saves memory in comparison to one large index table that has to store NULL for attributes not present in a given file, and also reduces the amount of data that needs to be retrieved from the index if any slave indexes are irrelevant to a given query.

3.3 The master/slave index

The indexing method employed by various domains in the LCARS file system benefits from the partially populated data space and is immune to any impact from the file system environment. The so-called "master/slave index" is comprised of a single master index that contains all attributes common to all file formats in that domain (e.g. file name and type). For each file format, an additional slave index with all remaining attributes of that type is stored. Splitting the attributes of a given file makes explicit use of the partially populated data space, and avoids storing NULL for attributes that are not defined by the file format.

Master index			Slave index for images			Slave index for MP3s		
Filename	Type	...	Filename	Width	Height	Filename	Genre	Artist
A.JPG	Image	...	A.JPG	1024	768	B.MP3	Hip Hop	...
B.MP3	MP3	...	C.JPG	640	480	D.MP3	Soul	...
C.JPG	Image	...	E.JPG	2048	1536			
D.MP3	MP3	...						
E.JPG	Image	...						

Figure 5. Sample master/slave index for image and MP3 files.

A simple heap file is used as internal organisation for each index table. The index is built by appending new tuples at the end of the master heap file and the appropriate slave heap file for each new file. This ensures that the order of tuples is maintained across tables. From the perspective of relational algebra, the master index and its associated slave indexes each represent a relation. Query processing in a master/slave index is thus reduced to a join operation of these relations, e.g. **MASTER** \bowtie **IMAGES** or **MASTER** \bowtie **MP3**. Since simultaneous appending of tuples maintains the order across tables, the merge join algorithm incorporating simultaneous sequential scanning of all index tables can be used to compute these joins in $O(n)$. After joining a tuple from the master index with the appropriate slave index, it has to be decided whether the file is part of the search result or not.

The master/slave index has been implemented from scratch as most off-the-shelf systems are not able to recognize the order of tuples across different index tables. The only exception known to the author is the Berkley DB that can be equipped with custom sorting functions and cursors, and is thus suitable to store a master/slave index. This kind of custom expandability can be considered crucial for special database applications like indexing file metadata.

4. SUMMARY

This paper presents a file system that offers database functionality, and also employs relational technology itself to index file metadata. This is a very unusual setting for database systems. Required features, such as application connectivity and indexing data structures that perform well inside a file system, had to be implemented from scratch as they were missing in most standard data management software.

5. REFERENCES

- [1] Apple Corporation. Spotlight. <http://www.apple.com/macosx/features/spotlight/>
- [2] DESKWORK Operating System. <http://www.deskwork.de/>
- [3] Google Inc. Google Desktop Features. <http://desktop.google.com/features.html>
- [4] Gorter, O. 2004. Database File System - An Alternative to Hierarchy Based File Systems. Master thesis, University of Twente, 2004
- [5] Internet Engineering Task Force 2005. A Universally Unique Identifier (UUID) URN Namespace. In Request For Comments 4122. <http://www.ietf.org/rfc/rfc4122.txt>
- [6] Internet Engineering Task Force 1998. A MIME Content-Type for Directory Information. In Request For Comments 2425. <http://www.ietf.org/rfc/rfc2425.txt>
- [7] Internet Engineering Task Force 1998. Internet Calendaring and Scheduling Core Object Specification (iCalendar). In Request For Comments 2445. <http://www.ietf.org/rfc/rfc2445.txt>
- [8] Japan Electronics and Information Technology Industries Association 1998. Design rule for Camera File system. In JEITA-49-2 1998. <http://www.exif.org/dcf.PDF>
- [9] Japan Electronics and Information Technology Industries Association. 2002. Exchangable image file format for digital still cameras: Exif. Version 2.2, JEITA CP-3451, April 2002
- [10] Kersten, M. et al. 2003. A Database Striptease or How to Manage Your Personal Databases. In Proceedings of the 29th VLDB Conference, Berlin 2003. <http://www.vldb.org/conf/2003/papers/S34P01.pdf>
- [11] Killian, T.J. 1984. Processes as files. In USENIX Association 1984 Summer Conference Proceedings, 1984
- [12] Microsoft Corporation. Windows Vista Developer Center: Chapter 4, Storage. <http://msdn2.microsoft.com/en-us/library/aa479870.aspx>
- [13] Nilsson, M. ID3v2 - The Audience is informed. <http://www.id3.org/>
- [14] Object Services and Consulting Inc. Semantic File Systems. <http://www.objs.com/survey/OFSExt.htm>
- [15] Shoens, K. et al. 1993. The Rufus System: Information Organization for Semi-Structured Data. In Proceedings of the 19th VLDB Conference, Dublin 1993. <http://www.vldb.org/conf/1993/P097.PDF>

Flexible Transaction Processing in the Argos Middleware

Anna-Brith Arntsen
Computer Science
Department
University of Tromsøe
9037 Tromsøe, Norway
annab@cs.uit.no

Mats Mortensen
Computer Science
Department
University of Tromsøe
9037 Tromsøe, Norway
mats@stud.cs.uit.no

Randi Karlsen
Computer Science
Department
University of Tromsøe
9037 Tromsøe, Norway
randi@cs.uit.no

Anders Andersen
Computer Science
Department
University of Tromsøe
9037 Tromsøe, Norway
aa@cs.uit.no

Arne Munch-Ellingsen
Computer Science
Department
University of Tromsøe
9037 Tromsøe, Norway
arneme@cs.uit.no

ABSTRACT

Transactional requirements, from new application domains and execution environments, are varying and may exceed traditional ACID properties. We believe that transactional middleware platforms must be flexible in order to adapt to varying transactional requirements. This is to some extent demonstrated within Web service environments where support for both ACID and long-running business transactions are provided. This paper presents an extension along the path to flexible transaction processing in the form of the **Argos Transaction Layer**. As opposed to present systems, the **Argos Transaction Layer** offers the potentiality to hot-deploy an extensible number of concurrently running transaction services, each providing different transactional guarantees. Currently, the **Transaction Layer** offers two services, one serving the ACID properties of distributed transactions, and one supporting long-running business transactions based on the use of compensation.

Categories and Subject Descriptors

H.4 [Information Systems Applications]: Miscellaneous;
D.2.8 [Software Engineering]: Metrics

General Terms

Architecture, configuration, management

Keywords

Flexible transactional middleware, Adaptability, Component-based system

1. INTRODUCTION

New application domains, including workflows, cooperative work, medical information systems, e-commerce, and web services environments, have transactional requirements that may be varying and evolving, exceeding the traditional ACID properties. An example is a travel arrangement application issuing a number of sub-transactions: booking flights, hotel rooms, theater tickets, and car rentals. This is a so-called long-running transaction, which, if structured as an

ACID transaction, will impede both performance and concurrency of a system. This transaction performs better if structured as a non-ACID transaction revealing intermediate results upon commit, and by using compensating transactions in case of failure (i.e. as Sagas [9]). A medical information system is another example, in which patient journals, radiographs and spoken reports may be stored over a number of heterogeneous sites. Medical personnel handling patient information may have either ACID or beyond-ACID requirements related to for instance response time, real-time guarantees or mobility issues.

The characteristics of extended transaction models [6] support our conviction that the "one-size fits all" paradigm is not sufficient and that a single approach to extended transactions will not suit all applications. To some extent, the "one-size fits all" contradiction is applied within Web services domains supporting two transaction services with different transactional guarantees.

However, it is our belief that transaction processing environments with two services fail to provide sufficient support for wide areas of applications. We designed **Reflects** [2] to fill the gap between required and provided support for flexible transaction processing, which offers an *extensible* number of concurrently active transaction services.

To demonstrate the flexibility enabled by **Reflects**, this paper presents a realization of a flexible transactional middleware platform based on the architecture of **Reflects**. The result is the **Argos Transaction Layer**, implemented as part of the **Argos** middleware container [17, 16]¹. The **Argos Transaction Layer** possesses the ability to embed an extensible number of concurrently running transaction services, each supporting different transactional guarantees. Currently the **Transaction Layer** provides two transaction services, one assuring the ACID properties of distributed transactions, and one supporting long-running business transactions based on the use of compensation. This layer adds flexible transaction execution to the **Argos** middleware in that an application can choose a suitable transaction service from the services available in the **Transaction Layer**, and that concurrently active transaction services are supported. The transactional support in **Argos** is configurable

¹<http://argos.cs.uit.no>

in that new services can be dynamically deployed during run time.

This paper is organized as follows. Section 2 presents related work and section 3 background on the **ReflecTS** platform. Section 4 follows with a presentation of the basic functionalities of the **Argos** middleware. Section 5 presents **Argos Transaction Layer** and an evaluation of the work. Finally, section 6 draws a conclusion and give directions for future work.

2. RELATED WORK

Traditional transactional middleware like Microsoft Transaction Server (MTS) [5], Sun's Java Transaction Server (JTS) [19] and CORBA's Object Transaction Service (OTS) [10] provide merely one transaction service supporting the ACID properties of distributed transactions.

The CORBA Activity Service Framework [13] extends the traditional approach by presenting an abstraction supporting various extended transaction models by a general-purpose event signalling mechanism. These ideas are also adopted by the J2EE Activity Service ² framework, which is the J2EE programming model of CORBA's Activity service.

Other approaches found within Web services domains provide support for more than one transaction service. The Web services specifications, WS-AtomicTransactions and WS-BusinessTransactions [12], and the OASIS BTP [14] specification defines two services, one serving ACID transactions and the other serving long-running activities. The Arjuna Technologies [15] has defined the Arjuna Transaction Service and the Arjuna Web Services Transaction implementation as part of the JBoss Transactions ³. The JBoss Transactions solution implements a transaction layer supporting atomic and long-running business transactions as part of its specifications. The GoTM ⁴ initiative by ObjectWEB aims to be the next generation transaction framework supporting a wide set of core services to build various personalities compliant to advanced transaction models.

By extending the traditional approaches, different extended transaction models [7] will be supported. To represent different transaction models the ACTA language [4] provides a general formalisms for describing advanced transaction models. Another formalism framework is the aspect-oriented language KALA [8], which extends the ACTA approach by adding a higher level of abstraction to achieve precise specification of transaction models.

The work presented in this paper contrasts related work in several matters. First, by supporting an extensible number of *concurrently* running transaction services, each exhibiting different transactional guarantees, and next, by providing run-time support for adding and/or removing services.

3. REFLECTS

The flexible transaction processing framework **ReflecTS** [2][3] has been designed to bridge the gap between required and provided support for flexible and adaptable transaction processing. **ReflecTS** provides flexibility by embedding an extensible number of transaction managers (*TM*s), each one offering different transactional guarantees.

²<http://sdic-esd.sun.com/ESD24/JSCDL/>

³<http://labs.jboss.com/jbosstm/>

⁴http://jotm.objectweb.org/TP_related.html

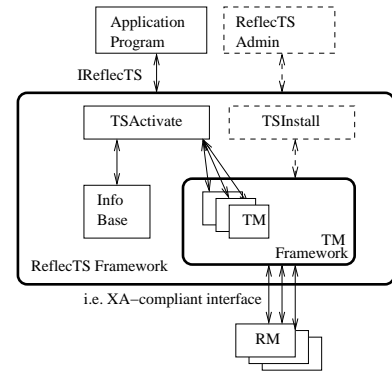


Figure 1: ReflecTS overview

The **ReflecTS** architecture is presented in figure 1 showing the **TM Framework** embedding *TMs*. The **TSInstall** module handles requests for *TM* configurations and resource manager (*RM*) registrations. These activities involves a consecutive update of the **InfoBase** keeping appurtenant descriptors related to the *TMs* and *RMs* (i.e. about transactional mechanisms and communication abilities).

The **IReflecTS** interface defines the interaction between the application program (*AP*) and **ReflecTS**, and is called to demarcate global transactions and to control the direction of their completion [2]. Via this interface, applications defines a transaction's transactional requirements and information about its requested *RMs*.

Based on the transactional requirements from the applications and the descriptors of available *TMs*, **TSActivate** selects a suitable *TM* for each particular transaction execution. Subsequently, and before the transaction is eventually started, a *TS* responsible for the coordination of the execution of the transaction is **composed** out of the selected *TM* and the requested *RMs*. A *TS* is responsible for coordinating the execution of a distributed transaction according to its provided properties. This is done in collaboration between the *TM* and the set of *RMs*, where the *TM* manages the global transactions by coordinating commit and recovery and the *RMs* perform local transaction control and participates in global commit and recovery protocols.

Tightly related to the transaction service **composition** and **activation** procedures are evaluation of *compatibility*. In order to assure the requested transactional requirements, committing the composition phase involves successful evaluation of **Vertical Compatibility** [3] between each pair of *TM* and *RM*. **Vertical Compatibility** involves defining matching transactional mechanisms (commit/recovery and concurrency control mechanisms) and communication abilities in each *TM - RM* pair.

Transaction service activation must be synchronized to assure that the properties of every transaction and each active *TS* sustain, involving evaluation of **Horizontal Compatibility**. This evaluation assures that any two services competing for the same resources can be concurrently active only if they are compatible. Compatible services does not violate the results and the consistency of the transactions executed by the individual services.

4. BACKGROUND ON ARGOS

Argos is a component-based middleware container for the

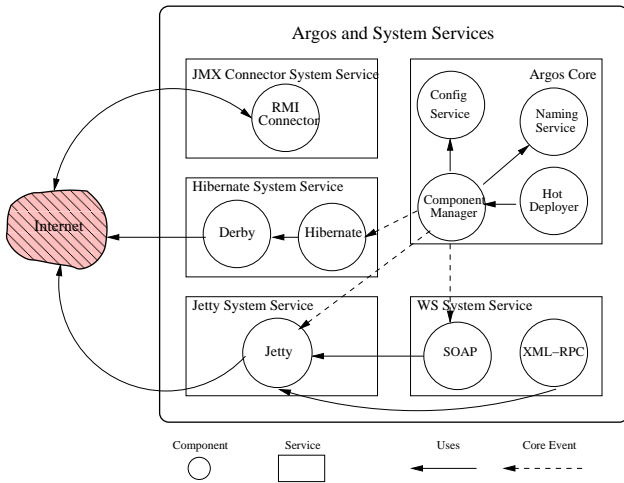


Figure 2: Argos Core and System Services

development of personal middleware systems [17, 16]⁵. Its functionalities are realized as a set of JMX (Java Management Extensions) DynamicMBeans⁶ components. This component model defines an architecture for management of distributed resources (local or remote), which is extended in Argos with elements related to component and service meta-models, component lifecycle and component dependency handling.

Figure 2 pictures the minimum configuration defining the Argos Core, which includes the extended JMX component model, a set of supported annotations, and core system services such as bindings (bindings to information sources), naming service and persistency. Persistency services includes accessing DBMS systems and Web services. The Web services functionalities are represented by the Jetty system service and the WS System Service providing necessary facilities for an Argos application to implement Web services. The DBMS functionalities are performed by Hibernate⁷ and Derby, through where applications interact with the databases without relying on any sort of SQL.

Argos applies annotations as provided by Java⁸. Annotations are included in the service code as metadata tags preceded by a @ and are used in combination with dynamic generation of components increasing the ease of extending component functionality and code maintenance. Argos component model builds on Plain Old Java Objects (POJO), which together with annotations are instantiated as JMX Dynamic MBeans by the container. A deployment descriptor defines dependencies to system and user services, and Argos uses *reflection* to inspect these descriptors for known annotations. Argos assures that all dependencies are met before a service is started, and the reflective abilities "glue" the services together.

The Argos core supports "Hot deployment", which gives a flexible structure for configurations of services without demanded recompilations. The HotDeployer component accomplishes this by deploying services and applications dur-

⁵<http://sourceforge.net/projects/jargos/>

⁶<http://java.sun.com/javase/technologies/>

⁷<http://www.hibernate.org/>

⁸<http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html>

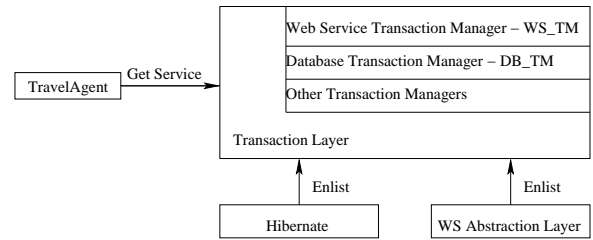


Figure 3: High-level model of the Argos Transaction Layer

ing run-time without restarting the container. Only a minimal modification of existing files is required. When a Argos service or application is loaded into the system as a POJO component, the HotDeployer component is initiated, producing MBeans representing the service.

Present Argos services [16] access local and external resources (both DBMS systems and Web services) by using the Hibernate System Service and the WS (Web Service) System Service. The WS System Service makes it easy for an application programmer to create a new Web service by using the @Webmethod annotation.

An Argos Container is an extension of the Argos Core, embedding both *system* and *user* services. In the container, deployed system and user services are developed to take advantage of the constitutional capabilities of the Argos core. For example, to access and store objects in a database, user applications apply the functionalities of the Hibernate system service. The Argos Transaction Layer presented in the following, is realized in a Argos Container.

5. ARGOS TRANSACTION LAYER

5.1 Introduction

The Argos Transaction Layer implements some of the core functionalities described in the ReflectS architecture. The basic functionalities of the Argos Transaction Layer are as follows: 1) Initial and run-time configuration of TMs, 2) An extensible number of concurrently running TMs, each exhibiting different transactional guarantees, and 3) TM selection.

Argos Transaction Layer is implemented as a system service, currently embedding two different TMs with different transactional guarantees: 1) Database Transaction Manager (DB_TM), supporting atomic transactions, and 2) Web Services Transaction Manager (WS_TM) supporting long-running Web services transactions. A TravelAgent application is implemented and deployed as a user service within the Argos container. The components of Argos Transaction Layer and the TravelAgent application are depicted in Figure 3 and described in the following.

5.2 Travel Agent Application

The TravelAgent application supports the arrangement of a journey involving a number of subtasks: *flight reservations*, *booking of hotel rooms*, *car reservations*, and *booking of theater tickets*. These services are exposed by the different companies either as Web services or DBMS application systems, where the latter supports the TX-interface [11]. A high-level overview of the TravelAgent application is pictured in figure 4.

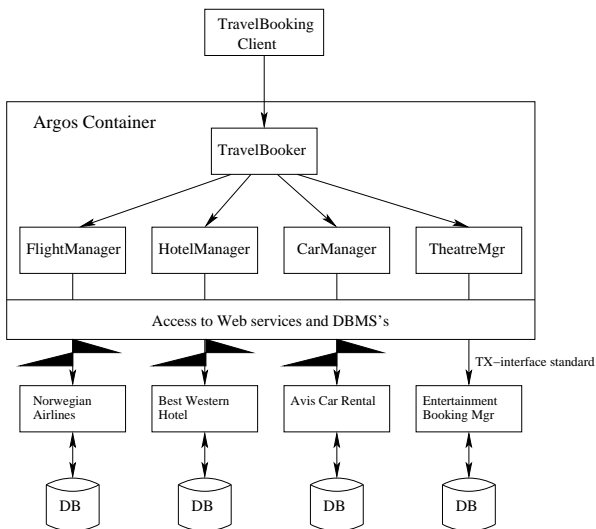


Figure 4: Model of the TravelAgent application

The TravelAgent application includes a TravelBooker component, which is responsible for demarcating transaction boundaries, specifying compensating transactions, and for interacting with the TravelBooking client and the managers of the underlying resources. The TravelBooker provides an interface to the clients (presented in section 5.3). This interface includes methods for initiating reservations performed by the managers.

The responsibilities of the application managers; Flight-, Hotel-, Car and Theater-manager, are to interact with Web Services and DBMS systems via the *abstraction layer*. The abstraction layer provides transparent access to underlying resources and ensures they are enlisted with the appropriate transaction manager and the correct transaction.

The TravelBooking client perform *TM* selection based on the requirements of the transaction, strict or relaxed atomicity, which in turn activates either the *DB_TM* or the *WS_TM* respectively. The *WS_TM* manages transactions involving both databases and Web Services while assuring relaxed (or semantic) atomicity based on the use of compensation. The *DB_TM* coordinates distributed transactions by the use of two-phase commit (2PC) protocol according to the X/Open DTP model [11]. By so means, *DB_TM* assures atomic commit of transactions running at DBMS systems. Inherent in the 2PC protocol is the ability to perform one-phase commit of transactions, signifying that *DB_TM* may work toward Web services when relaxed atomicity is a requirement.

5.3 Transaction Layer

The system components of the **Argos Transaction Layer** are implemented in the **Argos** middleware. Its components including the transaction managers, *WS_TM* and *DB_TM*, and the components of the TravelAgent application, are illustrated in figure 5.

The **Transaction Layer** offers the generic application programming interface (API) presented in table 1. This API, which follows the Java Transaction API (*JTA*) specification⁹, allows the applications to retrieve, select and use a *TM* of their choice.

⁹<http://java.sun.com/products/jta/>

Table 1: Transaction Layer API

```
public UserTransaction getUserTransaction(String id)
public TransactionManager getTransactionManager()
```

By means of the Hot Deployment facility, the pool of available *TMs* is adaptable to requirements from applications and from the environment. *TM* implementations are constrained to implement the generic interfaces listed in table 2:

Table 2: *TM* Generic Interface

```
javax.transaction.TransactionManager
javax.transaction.UserTransaction
argos.transaction.TransactionCoordinator
argos.transaction.Transaction
```

Multiple applications can use the same *TM* without any concurrent conflicts, which is accomplished by creating a new *TM* object for each thread. If a thread calls *getUserTransaction()* multiple times it will always receive the same object, but multiple threads will have different *TMs* and, thus, different *UserTransaction()* object. Each *TM* and each transaction is uniquely identified.

5.3.1 Resource Enlistment

Database and Web services resources are automatically enlisted and registered with the correct *TM* when such resources are used in a transaction context. The *TM* responsible for coordinating the transaction makes sure that all enlisted resources either abort or commit.

Hibernate and **WS Abstraction Layer** facilitates access to DBMS and Web services respectively. Hibernate is a part of the **Argos** core and the **WS Abstraction Layer** is a part of the **Argos** Transaction Layer implementation. Enlistment via Hibernate is facilitated by adding a custom connection provider suitable for each accessible database resource. The **WS Abstraction Layer** provides an abstraction to access underlying resources, which makes the services easier to use compared to access via **WS System Service** and **SOAP**. The main intention is to make this interaction similar as with regular Java objects. This is done by providing an API to the applications. Through this interface, applications specifies parameter values, which must be valid **SOAP** types, including an endpoint address (specified as an annotation), the operation name, parameter types and a return type. Additionally, **Argos** applications provides a description of each Web service, containing information necessary for the abstraction layer to create a Java object representing the Web service. The benefits of the **WS Abstraction Layer** are as follows: 1) Applications can use Web services in the same way they use any other object. 2) The abstraction layer will hide the type mapping so the application does not have any knowledge of **SOAP** at all.

5.3.2 Database Transaction Manager

The Database Transaction Manager (*DB_TM*) coordinates the execution of distributed transactions over multiple databases according to the X/Open DTP model [11]. The scheme of *DB_TM* assures atomicity where either all or none of the

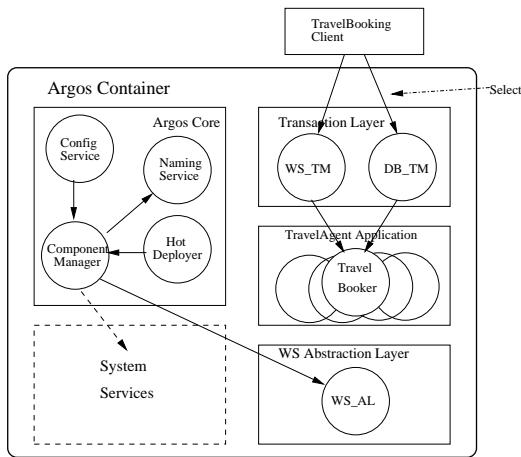


Figure 5: Integrating Transaction Service and TravelAgent Components in the Argos Container

tasks in a transaction commit. Concurrency is controlled by keeping the resources locked during the whole operation.

To support the experiments of this project, the Java Open Transaction Manager (JOTM) [18] providing database transaction support where selected. JOTM implements the three first interfaces from table 2. Initially, JOTM is compatible with the standard JTA interface, making it a perfect match for the rest of the system.

5.3.3 Web Service Transaction Manager

The Web Service Transaction Manager (WS_TM) provides the Argos applications with the abilities to use multiple Web services and databases in the same application. The WS_TM implements a one-phase commit protocol [1] and assures relaxed atomicity based on the use of compensation [9]. Each Web service interface is adapted to conform to identical interfaces, which are exposed to the managers via the WS Abstraction Layer.

Table 3: Register Compensation

<code>registerCompensation(String func, Object... params)</code>
--

Every Web service operation defines a compensating operation as defined in table 3. The implementations of compensating activities are performed based on information given by the applications. Argos applications register compensation operations after having performed a Web service operation. When Argos load the application, it saves this information for use in case of transaction failure and a subsequent rollback. If the original operation fails, Argos simply call the compensating operation registered. For simplicity, we assume that each underlying Web service implements and exhibits the same API. This scheme will work for all Web services providing the ability to register compensating operations.

5.4 Evaluation

Our tests with Argos Transaction Layer and the TravelAgent application show that i) the middleware platform can offer different transaction services to the application, ii) new transaction services can be deployed in the Transaction

Layer during run time, and iii) the platform can support concurrently active transaction services. These are key features in the ReflecTS architecture and important for providing the required flexible transaction execution environment.

The Argos middleware container was chosen as our core middleware platform because of its extensible architecture for easy deployment of system services during runtime. From our tests with Argos Transaction Layer we find Argos to be a suitable platform for implementing a layer for flexible transaction execution.

The Hot Deployment facility supported by Argos makes transaction service deployment during runtime possible. At present, we have only two transaction services available for use in the Transaction Layer. These are the DB Transaction Manager ("DB_TM"), supporting traditional ACID transactions, and WS Transaction Manager ("WS_TM"), supporting long-running transactions based on the use of compensation. However, more transaction services offering other types of transactional properties may be developed and deployed into the Transaction Layer.

When testing Argos Transaction Layer we started the system with one available transaction service. Initially, the transactional properties of this service represented the only transaction execution possibility in Argos. During run time the second transaction service was deployed, and the transactional properties P of both services, i.e. $P(DB_TM)$ and $P(WS_TM)$, were available to the application. The requirements of each issued transaction determined which one was selected to control its execution. By the use of the WS_TM manager, transaction failure and recovery was handled by the initiating of compensating operations.

Testing of the Argos Transaction Layer and the TravelAgent application have demonstrated concurrency among the two deployed transaction managers. Both managers can be concurrently active coordinating transaction executions. This is achieved by instantiating separate objects for each thread. As the two transaction managers may cause conflicting transaction executions when controlling execution on a common data set, we constrained the use of the managers so that DB_TM and WS_TM managed transactions accessing separate data sets. Compatibility between concurrent transaction services is part of our ongoing work, and the results of this will be reported elsewhere.

One of the key features of this work has been enlistment of the transactional resources with the appropriate manager. This standard middleware functionality simplifies the programming model for applications. To implement this feature, Argos had to implement abstractions for accessing transactional resources. For the case of the database resources, this was automatically performed by means of a custom connection provider for Hibernate. For the case of the Web services, this was performed by the WS abstraction Layer.

During this work, the performance of the system were not considered. The main emphasize has been given transaction service deployment, achieving concurrency among the services and automatic enlistment of transactional resources.

6. CONCLUDING REMARKS AND FUTURE WORK

Varying transactional requirements from new application domains demand flexible and adaptable transaction process-

ing assuring properties exceeding the traditional ACID properties. The primary goal for this project was to develop a **Transaction Layer** for the **Argos** middleware container providing flexible transaction processing as proposed in the **ReflectTS** architecture [2]. This project has proven the **Argos Transaction Layer** as a proper host for concurrently running transaction services. Currently, the **Transaction Layer** provides two transaction managers: one assuring the ACID properties of distributed transactions, and one assuring relaxed atomicity based on the use of compensating activities. A **TravelAgent** application has demonstrated transaction manager selection and concurrent usage while assuring the transactional requirements. This application initiates transactions both at regular DBMSs and at Web services. In addition, we have shown that, by means of the Hot Deployment feature, configurations of transaction managers are feasible without system recompilation.

As part of current work, we are working on formalisms related to evaluation of compatibility, both to assure transactional requirements and to assure the properties of each active transaction and transaction service. These are issues solving problems when incompatible transaction services are accessing overlapping data. Further, this version of **Argos Transaction Layer** applies service selection at the application layer. Evidently, this increases the complexity of the applications while offloading the complexity of the middleware. In order to improve and ease the development of applications, we will, during future work add mechanisms for service selection at the middleware layer (as proposed by **ReflectTS**).

7. REFERENCES

- [1] Maha Abdallah, R. Guerraoui, and P. Pucheral. One-phase commit: Does it make sense? In *ICPADS '98: Proceedings of the 1998 International Conference on Parallel and Distributed Systems*, page 182, Washington, DC, USA, 1998. IEEE Computer Society.
- [2] Anna-Brith Arntsen and Randi Karlsen. Reflects: a flexible transaction service framework. In *ARM '05: Proceedings of the 4th workshop on Reflective and adaptive middleware systems*, pages 1–6, Grenoble, France, 2005. ACM Press.
- [3] Anna-Brith Arntsen and Randi Karlsen. Transaction service composition, a study of compatibility related issues. In *Proceeding of the 9th International Conference on Enterprise Information Systems, ICEIS 2007*, Funchal, Madeira - Portugal, June 2007.
- [4] Panos K. Chrysanthis and Krithi Ramaritham. Acta: A framework for specifying and reasoning about transaction structure and behavior. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, May 1990.
- [5] Microsoft Corporation. The .net framework, 2000. <http://www.microsoft.com/net/>.
- [6] Ahmed K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, 1992.
- [7] W. Litwin Elmagarmid A., Y. Leu and M. Rusinkiewicz. A multibase transaction model for interbase. In *Proceedings of the 16th International Conference on VLDB*, pages 507–518, 1990.
- [8] Johan Fabry and Theo D'Hondt. Kala: Kernel aspect language for advanced transactions. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1615–1620, Dijon, France, 2006. ACM Press.
- [9] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 249–259, San Francisco, California, United States, 1987. ACM Press.
- [10] Object Management Group. Corba services, transaction service specification, version 1.4, 2003. http://www.omg.org/technology/documents/formal/transaction_service.htm.
- [11] Open Group. X/open distributed transaction processing: Reference model, version 3, 1996.
- [12] W3C Working Group. Web services architecture, working draft, February 2004. <http://www.w3.org/TR/ws-arch/>.
- [13] I. Houston, M. C. Little, I. Robinson, S. K. Shrivastava, and S. M. Wheeler. The corba activity service framework for supporting extended transactions. *Lecture Notes in Computer Science*, 2218, 2001.
- [14] Mark Little. Transactions and web services. *Commun. ACM*, 46(10):49–54, 2003.
- [15] Arjuna Technologies Ltd. Web services transaction management (ws-txm) ver1.0. 2003. http://www.arjuna.com/library/specs/ws_caf_1-0/WS-TXM.pdf.
- [16] Arne Munch-Ellingsen, Anders Andersen, and Dan Peder Eriksen. Argos, an extensible personal application server. In *Proceedings of the Middlware 2007*, Newport Beach, Orange County, California, USA, November 2007. ACM Press.
- [17] Arne Munch-Ellingsen, B. Thorstensen, D.P. Eriksen, and Anders Andersen. Building pervasive services using flock sensor network and flock container middleware. In *Proceedings of the IEEE 21st International Conference on Advanced Information Networking and Applications, AINA 2007*, Niagara Falls, Ontario, Canada, 2007.
- [18] Open Source Middleware Object Web. Java open transaction manager, 2005. <http://jotm.objectweb.org>.
- [19] Allarmaraju Subrahmanyam. Java transaction service, 1999. <http://www.subrahmanyam.com/articles/jts/JTS.html>.

Tailor-made Lock Protocols and their DBMS Integration

Sebastian Bächle

University of Kaiserslautern
67663 Kaiserslautern, Germany
baechle@informatik.uni-kl.de

Theo Härder

University of Kaiserslautern
67663 Kaiserslautern, Germany
haerder@informatik.uni-kl.de

ABSTRACT

We outline the use of fine-grained lock protocols as a concurrency control mechanism for the collaboration on XML documents and show that their tailor-made optimization towards the access model used (e.g., DOM operations) pays off. We discuss how hard-wired lock services can be avoided in an XML engine and how we can, based on loosely coupled services, exchange lock protocols even at runtime without affecting other engine services. The flexible use of these lock protocols is further enhanced by enabling automatic runtime adjustments and specialized optimizations based on knowledge about the application. These techniques are implemented in our native XML database management system (XDBMS) called XTC [5] and are currently further refined.¹

1. MOTIVATION

The hierarchical structure of XML documents is preserved in native XDBMSs. The operations applied to such tree structures are quite different from those of tabular (relational) data structures. Therefore, solutions for concurrency control optimized for relational DBMSs will not meet high performance expectations. However, efficient and effective transaction-protected collaboration on XML documents [1] becomes a pressing issue because of their number, size, and growing use. Tailor-made lock protocols that take into account the tree characteristics of the documents and the operations of the workload are considered a viable solution. But, because of structure variations and workload changes, these protocols must exhibit a high degree of flexibility as well as automatic means of runtime adjustments.

Because a number of language models are available and standardized for XML [9,10], a general solution has to support fine-grained locking – besides for separating declarative XQuery requests – for concurrently evaluating stream-, navigation-, and path-based queries. With these requirements, we necessarily have to map all declarative operations to a navigational access model, e.g., using the DOM operations, to provide for fine-granular concurrency control. We have already developed a family consisting of four DOM-based lock protocols [6]. Here, our focus is on the engineering aspects how such protocols can be efficiently integrated, but sufficiently encapsulated in an XDBMS such that they can be automatically exchanged or adapted to new processing situations at runtime.

In Section 2, we explain the need for lock protocols tailored to the specific characteristics of XML processing, before we outline the mechanism underlying the runtime exchange of lock protocols in Section 3. Using the concept of meta-locking, we sketch in Section 4 how we achieved cross-comparison of 12 lock protocols

in an identical XDBMS setting without any system modification. Here, XTC (XML Transaction Coordinator, [5]) served as a testbed for all implementations and comparative experiments. Various forms of runtime adjustments on lock protocols are discussed in Section 5, before we introduce further ways of protocol specializations in Section 6. Finally, Section 7 concludes the paper.

2. FINE-GRAINED DOM-BASED LOCKING

Because our XML documents are stored in a B*-tree structure [5], the question whether or not specific tree-based lock protocols can be used immediately arises. So-called B-tree lock protocols provide for structural consistency while concurrent database operations are querying or modifying database contents and its representation in B-tree indexes [2]. Such locks also called latches isolate concurrent operations on B-trees, e.g., while traversing a B-tree, latch coupling acquires a latch for each B-tree page before the traversal operation is accessing it and immediately releases this latch when the latch for the successor page is granted or at end of operation at the latest [1]. In contrast, locks isolate concurrent transactions on user data and – to guarantee serializability [4] – have to be kept until transaction commit. Therefore, such latches only serve for consistent processing of (logically redundant) B-tree structures.

Hierarchical lock protocols [3] – also denoted as multi-granularity locking (MGL) – are used “everywhere” in the relational world. For performance reasons in XDBMSs, fine-granular isolation at the node level is needed when accessing individual nodes or traversing a path, whereas coarser granularity is appropriate when traversing or scanning entire trees. Therefore, lock protocols, which enable the isolation of multiple granules each with a single lock, are also beneficial in XDBMSs. Regarding the tree structure of documents, objects at each level can be isolated acquiring the usual locks with modes R (read), X (exclusive), and U (update with conversion option), which implicitly lock all objects in the entire subtree addressed. To avoid lock conflicts when objects at different levels are locked, so-called intention locks with modes IR (intention read) or IX (intention exclusive) have to be acquired along the path from the root to the object to be isolated and vice versa when the locks are released [3]. Hence, we could map the relational IRIX protocol to XML trees and use it as a generic solution where the properties of the DOM access model are neglected.

Using the IRIX protocol, a transaction reading nodes at any tree level had to use R locks on the nodes accessed thereby locking these nodes together with their entire subtrees. This isolation is too strict, because the lock protocol unnecessarily prevents writers to access nodes somewhere in the subtrees. Giving a solution for this problem, we want to sketch the idea of lock granularity adjustment to DOM-specific navigational operations. To develop true DOM-

¹ This work has been partially supported by the German Research Foundation (DFG).

	-	IR	NR	LR	SR	IX	CX	SU	SX
IR	+	+	+	+	+	+	+	-	-
NR	+	+	+	+	+	+	+	-	-
LR	+	+	+	+	+	+	-	-	-
SR	+	+	+	+	+	-	-	-	-
IX	+	+	+	+	-	+	+	-	-
CX	+	+	+	-	-	+	+	-	-
SU	+	+	+	+	+	-	-	-	-
SX	+	-	-	-	-	-	-	-	-

Figure 1. Lock compatibilities for taDOM2

based XML lock protocols, we introduce a far richer set of locking concepts. While MGL essentially rests on intention locks and, in our terms, subtree locks, we additionally define locking concepts for nodes, edges, and levels. Edge locks having three modes [6] mainly serve for phantom protection and, due to space restrictions, we will not further discuss them.

We differentiate read and write operations thereby renaming the well-known (IR, R) and (IX, X) lock modes with (IR, SR) and (IX, SX) modes, respectively. As in the MGL scheme, the U mode (SU in our protocol) plays a special role, because it permits lock conversion. Novel lock modes are NR (node read) and LR (level read) in a tree which, in contrast to MGL, read-lock only a node or all nodes at a level, but not the corresponding subtrees. Together with the CX mode (child exclusive), these locks enable *serializable* transaction schedules with read operations on inner tree nodes, while concurrent updates may occur in their subtrees. Hence, these XML-specific lock modes behave as follows:

- An NR mode is requested for reading context node c . To isolate such a read access, an IR lock has to be acquired for each node in the ancestor path. Note, the NR mode takes over the role of IR combined with a specialized R, because it only locks the specified node, but not any descendant nodes.
- An LR mode locks context node c together with its direct-child nodes for shared access. For example, evaluation of the child axis only requires an LR lock on context node c and not individual NR locks for all child nodes.
- A CX mode on context node c indicates the existence of an SX lock on some direct-child nodes and prohibits inconsistent locking states by preventing LR and SR locks. It does not prohibit other CX locks on c , because separate child nodes of c may be exclusively locked by other transactions (the compatibility is then decided on the child nodes themselves).

Figure 1 contains the compatibility matrix for our basic lock protocol called taDOM2. To illustrate its use, let us assume that the node manager has to handle for transaction T_1 an incoming request *Get-ChildNodes()* for context node *book* in Figure 2. This requires appropriate locks to isolate T_1 from modifications of other transactions. Here, the lock manager can use the level-read optimization

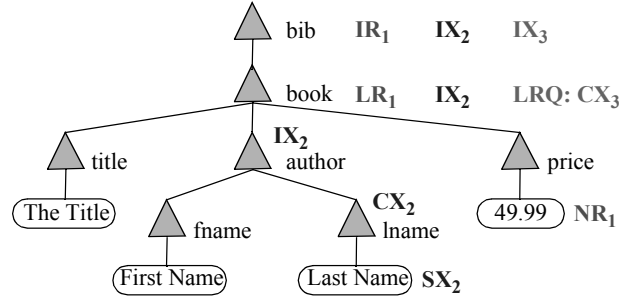


Figure 2. taDOM2 locking example

and set the perfectly fitting mode LR on *book* and, in turn, protect the entire path from the document root by appropriate intention locks of mode IR. Our prefix-based node labeling scheme called SPLIDs (stable path labeling identifiers are similar to OrdPaths [8]) greatly supports lock placement in trees, because SPLIDs carry the node labels of all ancestors. Hence, access to the document is not needed to determine the path to a context node. After having traversed all children, T_1 navigates to the content of the *price* element after the lock manager has set an NR lock for it. Then, transaction T_2 starts modifying the value *lname* and, therefore, acquires an SX lock for the corresponding text node. The lock manager complements this action by acquiring a CX lock for the parent node and IX locks for all further ancestors. Simultaneously, transaction T_3 wants to delete the *author* node and its entire subtree, for which, on behalf of T_3 , the lock manager must acquire an IX lock on the *bib* node, a CX lock on the *book* node, and an SX lock on the *author* node. The lock request on the *book* node cannot immediately be granted because of the existing LR lock of T_1 . Hence, T_3 – placing its request in the lock request queue (LRQ: CX₃) – must synchronously wait for the release of the LR lock of T_1 on the *book* node.

Hence, by tailoring the lock granularity to the LR operation, the lock protocol enhances concurrency by allowing modifications of other transactions in subtrees whose roots are read-locked.

3. USE OF A PROTOCOL FAMILY

Experimental analysis of these protocols led to some severe performance problems in specific situations which were solved by the follow-up protocol taDOM2+. Conversion of LR was particularly expensive. Assume T_1 wants modify *price* and has to acquire an SX lock on it in the scenario sketched in Figure 2. For this purpose, the taDOM2 protocol requires a CX lock on its parent *book*. Hence, the existing LR has to be converted into a CX lock and, in turn, a successful conversion requires NR locks on all children (potentially read by T_1) of *book*. As opposed to ancestor determination by the SPLID of a context node, the lock manager cannot calculate the SPLIDs of its children and, hence, has to access the document to explicitly locate all affected nodes – a very expensive operation. By introducing suitable intention modes, we obtained the more complex protocol taDOM2+ having 12 lock modes. The DOM3 standard introduced a richer set of operations which led to several new tailored lock modes for taDOM3 and – to optimize specific conversions – we added even more intention modes resulting in the truly complex protocol taDOM3+ specifying compatibilities and conversion rules for 20 lock modes (see [6] for details).

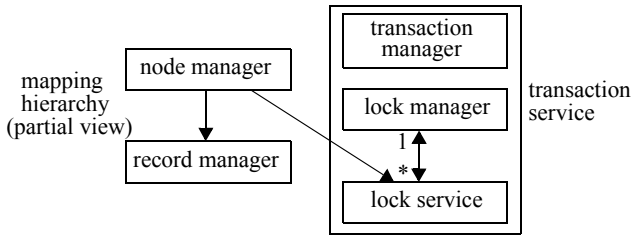


Figure 3. Interaction of node manager and locking facilities

While in our initial implementation the taDOM2 protocol was hard-wired, we looked for an elegant integration mechanism to transparently enable protocol changes (e.g., another compatibility matrix) or use of alternative protocols. Therefore, we decoupled the logic for navigating and manipulating the documents from all protocol-specific aspects by encapsulating them in so-called lock services, which are provided by the lock manager. The node manager uses them to enforce its isolation needs, instead of directly requesting specific locks from the lock manager. For this purpose, the lock service interface offers a collection of methods, which sufficiently cover all relevant cases, e.g., for locking a single node or an entire subtree in either shared or exclusive mode. Figure 3 sketches the interaction of node manager and the locking facilities involved in protocol use, lock mode selection, and application of conversion rules.

This small restructuring reduced the responsibility of the node manager for coping with *how the resources have to be locked* to simply saying *what resources have to be locked*. Based on such “declarative” lock requests, the lock service infers the appropriate locks and lock modes according to the respective protocol and acquires them from the lock manager. The granted locks and the waiting lock requests are maintained in a conventional lock table and a wait-for graph as it is known from relational systems. For protocol-specific details like compatibility matrix and lock conversions, however, the lock manager depends on information provided by the lock service. Thus, internal structures of the lock manager like the lock table and the deadlock detector could be completely decoupled from the used protocols, too. Finally, we are now able to change the lock protocol by simply exchanging the lock service used by the node manager. Furthermore, it is now even possible to use multiple protocols, e.g., taDOM2+ and taDOM3+, simultaneously for different kinds of and workloads for documents inside the same server instance.

4. META-LOCKING

As described in the previous section, the key observation for transparent lock protocol exchange is an information exchange between lock manager and a lock service about the type of locks and compatibilities present. The lock services controlled by the lock manager can then be called by specific methods and each individual lock service can act as a kind of abstract data type. As a consequence, the node manager can plan and submit the lock requests in a more abstract form only addressing general tree properties. Using this mechanism, we could exchange all “closely related” protocols of the taDOM family and run them without additional effort in an identical setting. By observing their behavior under the same benchmark, we gained insight into their specific blocking behavior and lock administration overhead and, in turn, could react with some fine-tuning.

Even more important is a cross-comparison between different lock protocol families to identify strengths and weaknesses in a broader context. On the other hand, when unrelated lock protocols having a different focus can be smoothly exchanged, we would get a more powerful repertoire for concurrency control optimization under widely varying workloads.

We found quite different approaches to fine-grained tree locking in the literature and identified three families with 12 protocols in total: Besides our taDOM* group with 4 members, we adjusted the relational MGL approach [3] to the XML locking requirements and included 5 variants of it (i.e., IRX, IRX+, IRIX, IRIX+, and URIX) in the so-called MGL* group. Furthermore, three protocol variants described in [7] were developed as DOM-based lock protocols in the Natix context (i.e., Node2PL, NO2PL, and OO2PL), but not implemented so far. They are denoted as the *2PL group.

To run all of them in an identical system setting – by just exchanging the service responsible for driving the specific lock protocol – is more challenging than that of the taDOM family. The protocol abilities among the identified families differ to a much larger extent, because the MGL* group does not know the concept of level locks and the mismatch of the *2PL group with missing subtree and level locks is even larger.

For this reason, we developed the concept of meta-locking to bridge this gap and to automatically adjust the kinds of lock requests depending on the current service available. Important properties of a lock protocol influencing the kind of service request are the support of shared level locking, shared tree locking, and exclusive tree locking. To enable an inquiry of such properties by the node manager, the lock service provides three methods.

- *supportsSharedLevelLocking*: If a protocol supports the level concept, a request for all children or a scan traversing the child set can be isolated by a single level lock (i.e., LR). Otherwise, all nodes (and navigation edges) must be locked separately.
- *supportsSharedTreeLocking*: Analogously to level locks, subtrees can be read-locked by a single request, if the protocol has this option. Otherwise, all nodes (and navigation edges) of the subtree must be locked separately.
- *supportsExclusiveTreeLocking*: This protocol property enables exclusive locking of a subtree by setting a lock on its root node. If this option is not available, then subtree deletion requires traversal and separate locking of all nodes, before the deletion can take place in a second step.

For a lock request on a context node, the node manager can select a specification of the lock mode (*Read*, *Update*, or *Exclusive*) for the context node itself, the context node and the level of all its children or the context node and its related subtree. For navigational accesses, a lock mode for one of the edges *prevSibling*, *nextSibling*, *firstChild*, or *lastChild* can be specified, in addition. Then, the lock manager translates the lock request to a specific lock mode dependent on the chosen protocol.

Although the MGL* group is only distantly related and the *2PL group is not related at all to our protocol family, this meta-locking concept enabled without further “manual” interaction a true and precise cross-comparison of all 12 protocols, because they were run under the same benchmark in XTC using the same system configu-

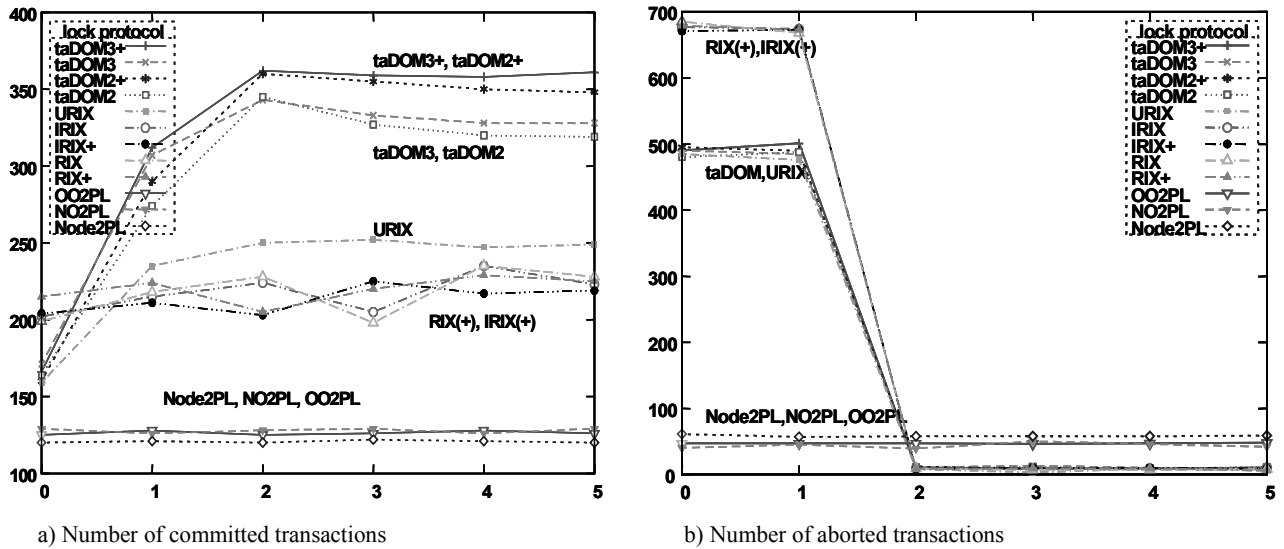


Figure 4. Overall results of a transaction benchmark (variation of lock depth)

ration parameters. All benchmark operations and the node-manager-induced lock protocols were applied to the taDOM storage model [5] of XTC and took advantage of its refined node structure and the salient SPLID properties concerning lock management support.

As it turned out by empirical experiments, *lock depth* is an important and performance-critical parameter of an XML lock protocol. Lock depth n specifies that individual locks isolating a navigating transaction are only acquired for nodes down to level n . Operations accessing nodes at deeper levels are isolated by subtree locks at level n . Note, choosing lock depth 0 corresponds to the case where only document locks are available. In the average, the higher the lock depth parameter is chosen, the finer are the lock granules, but the higher is the lock administration overhead, because the number of locks to be managed increases. On the other hand, lock conflicts typically occur at levels closer to the document root (lower lock depth) such that fine-grained locks (and their increased management) at levels deeper in the tree do not pay off. Obviously, the taDOM and the MGL protocols can easily be adjusted to the lock-depth parameter, whereas the *2PL group cannot benefit from it.

In our lock protocol competition, we used a document of about 580,000 tree nodes (~8MB) and executed a constant system load of 66 transactions taken from a mix of 5 transaction types. For our discussion, neither the underlying XML documents nor the mix of benchmark operations are important. Here, we only want to show the overall results in terms of successfully executed transactions (throughput) and, as a complementary measure, the number of transactions to be aborted due to deadlocks.

Figure 4a clearly indicates the value of tailor-made lock protocols. With the missing support for subtree and level locks, protocols of the *2PL group need a ponderous conversion delivering badly adjusted granules. On the other hand, the MGL protocols (only level locks missing) roughly doubled the transaction throughput as compared to the *2PL group. Finally, the taDOM* group almost doubled the throughput compared to the MGL* group. A reasonable application to achieve fine-grained protocols requires at least

a lock depth of 2, which also confirms the superiority of MGL and taDOM in terms of deadlock avoidance (see Figure 4b).

Hence, the impressive performance behavior of the taDOM* group reveals that a careful adaptation of lock granules to specific operations clearly pays off (see again the discussion in Section 2).

5. RUNTIME PROTOCOL ADJUSTMENT

At runtime, the main challenge for concurrency control is the preservation of a reasonable balance of concurrency achieved and locking overhead needed. The most effective and widely used solution for this objective is called *lock escalation*: The fine-grained resolution of a lock protocol is – preferably in a step-wise manner – reduced by acquiring coarser granules. For example in relational systems, the page containing a specific record is locked instead of the record itself. If too many pages are to be locked in the course of processing, the lock manager may try to acquire a single lock for the entire table. In B-tree indexes, separator keys of non-leaf pages can be exploited as intermediate lock granularities to improve scalability [2]. Although native XDBMSs often store the document nodes in page-oriented, tree-based index structures, too, an escalation from node locks to page locks is not suitable anymore. Because the nodes are stored in document order and, as a consequence, fragments of multiple subtrees can reside in the same physical page, page level locks would jeopardize the concurrency benefits of hierarchical locks on XML document trees. Hence, lock escalation in our case means the reduction of the lock depth: we lock subtrees at a higher level using a single lock instead of separately locking each descendant node visited. Consequently, the number of escalation levels is limited by the structure of a document and not by its physical DBMS representation. This is also a notable aspect of our encapsulation design.

The acquisition of a coarser lock granule than actually needed is typically triggered by the requestor of the locks itself, e.g., a subtree scan operator, or by the lock manager, when the number of maintained locks for a transaction reaches a critical level or the requested lock is at a deeper level than the pre-defined maximum lock

depth. Admittedly, especially the heuristics of the lock manager is rather a mechanism enabling a system to handle large documents than an optimization for higher transaction throughput. The lock escalation is performed independently of the processing context, because it is *necessary* and not because it is *wise*. Therefore, we present in the following some concepts that allow us to go further than a simple reduction of the global lock depth and to increase the performance of the system by doing smart lock escalation.

We aim to preserve fine-grained resolution of our lock protocols in hot spot regions of a document to maintain high concurrency, while we gracefully give it up in low-traffic regions to save system resources. In doing so, we have to face the challenge to decide whether or not it is good idea to perform a lock escalation on a subtree. Again, the solution lies in the design of our locking facilities. By making the lock manager “tree-aware”, we can very easily exploit its implicit knowledge about the traffic on a document. Lock requests for a specific document node trigger the lock manager to transparently acquire the required intention locks on the ancestor path. The ancestor nodes and the appropriate lock modes are, of course, provided by the lock service. Thus, the lock table knows not only the current lock mode of a node, the number of requests and the transactions that issued these requests, but also its level in the document and the level of the target node when it requests the intention locks on the path from root to leaf. This information can be used as input for a heuristics to decide about local lock escalations in specific subtrees.

The example depicted in Figure 5 illustrates how this cheaply gathered information can be used effectively: Transaction T_1 requests a shared lock for the highlighted element node at level 6, but before this request can be granted, appropriate intention locks on the ancestor path have to be acquired. At level 4, the lock table recognizes that T_1 already requested 50 shared intention locks on this node. This indicates that T_1 has already done some navigation steps in the subtree rooted at the current node, and that it will probably perform some more. Therefore, the lock table *asks* the lock service if the initial lock request at level 6 should be overridden by a stronger lock at level 4 to limit the number of locks in case that T_1 continues its navigation in the subtree. Because the target level is more than one level deeper and locks of other transactions are not present on this node, the lock service decides in favor of a shared subtree lock on the ancestor node to save resources. If the node at level 4, however, would also be locked by another transaction T_2 with an intention exclusive lock, it would probably apply the escalation to the parent node at level 5 to avoid a blocking situation.

The great advantage of this approach is that collection of additional information is avoided and that all relevant information is always immediately available when needed. In our example, the levels and the distance of the current and the target level served as weights in this decision process. Nevertheless, it is also possible to incorporate high-level information like statistics about the current transaction workload or the document itself. Besides the maximum depth and the number of document nodes, especially the fan-out characteristics of the document can be helpful. Depending on the complexity of the heuristics used, however, it is reasonable to reduce the additional computational overhead to a minimum. For that reason, the lock modes and the number of requests for a specific resource might

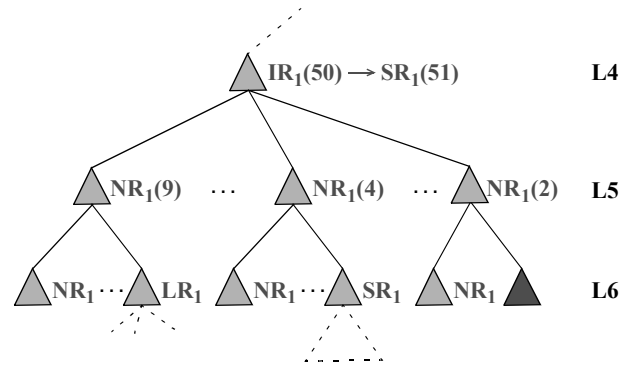


Figure 5. Lock escalation example

only be consulted, if the number of locks held by the current transaction reaches a certain threshold, and only if they indicate an optimization potential, further information has to be evaluated.

Although navigation on the XML document tree or evaluation of declarative queries formulated in XPath or XQuery allow many degrees of freedom and cause varying access patterns, some basic tasks like the evaluation of positional predicates are repeated very often. Hence, another promising way to optimize the application of our protocols is the adaptation to typical access patterns. To give a simple example, assume that a transaction wants to navigate to the first child node of an element that satisfies a certain search criterion. The transaction can either fetch all child nodes by a single call of *GetChildNodes()* and pick the searched one, or descend to the first child node with *GetFirstChild()* and scan for the child node with *GetNextSibling()*. In the first case, only a single LR lock is required to protect the operation, but many child nodes are unnecessarily locked possibly causing lower concurrency. In the second case, concurrency may be much higher, since only those nodes are locked that are necessary to keep the navigation path stable. However, the lock overhead is higher, too, because each call of *GetNextSibling()* requires the acquisition of additional locks.

Obviously, it depends on the position of the searched child node and the fan-out of the specific element whether the first or the second alternative is better, and, ideally, the caller respectively the query optimizer should choose the appropriate alternative. In most cases, however, it is not possible to reliably predict the best one. Then, it is reasonable to start with the second strategy to preserve high concurrency. If the search does not stop until the k -th child is reached, the lock service can be advised to try lock escalation and to lock all siblings with an LR lock on the parent, because further calls of *GetNextSibling()* will follow with high probability. Instead of waiting for an explicit hint, the node manager itself could also keep track of the n most recent navigation steps and predict the best lock mode for the next steps.

The goal of this kind of pre-claiming is to request locks that are not only sufficient for the current operation but also optimal for future operations. In contrast to the on-demand escalation capabilities we described before, such context-sensitive heuristics can be applied much earlier and lead to better overall performance, because not only locking overhead but also danger of deadlocks are reduced.

Further heuristics are primarily designed to speed up the evaluation of declarative queries. The context nodes for a query result, for example, are usually identified before the actual output is computed. Because the output often embodies nodes in the subtree rooted at the context nodes, the respective nodes or subtrees can already be locked when a context node is identified. This avoids late blocking of a transaction during result construction. Contrarily, the lock depth can be locally increased in the identification phase to reduce the risk of blocking other transactions on examined nodes that did not qualify for the query result.

6. PROTOCOL SPECIALIZATION

So far, we described general runtime optimizations for a universal XDBMS. Further improvements are possible if we take also aspects of the applications themselves into account and adjust our lock protocols accordingly. Since the adaptations are by nature very special to a specific application area, we do only sketch a few scenarios at this point to give an idea of how such specializations may look like.

Applications that change the documents in a uniform manner appear to be a promising field. A special case are “append-only” scenarios like digital libraries where the document order does not play an important role, and new nodes or subtrees are only appended. Existing structures remain untouched and, at the most, the content of existing nodes is updated, e.g., to correct typing errors. This allows us to omit the edge locks most of the time because transactions must not protect themselves from phantom insertions between two existing nodes. In addition to that, insertions can be flagged with an additional exclusive insert lock to signal other transactions that they can skip the new last child if they do not rely on the most recent version of the document.

Many applications do also rely on a specific schema to define tree-structured compounds that reflect logical entities like customer data, addresses, or products, and are typically accessed as a whole. Hence, such entities in a document may be identified with the help of a schema and directly locked with a single subtree lock, whereas other parts are still synchronized with fine-grained node locks. In an XML-based content management system, for example, the metadata is accessed and modified in very small granules, whereas the content parts usually consist of large subtrees with some sort of XML markup that are always accessed as a logical unit. In some applications it might even be possible to change the basic lock granule from single document nodes to larger XML entities like complex XML-Schema types.

7. CONCLUSION

In this paper, we proposed the use of techniques adaptable to various application scenarios and configurations supporting high concurrency in native XDBMS. We started with an introduction into the basics of our tailor-made lock protocols, which are perfectly eligible for a fine-grained transaction isolation on XML document trees, and showed how they can be encapsulated and integrated in

an XDBMS environment. By introducing the concept of meta-locking, we discussed the software-engineering principles for the exchange of the lock service responsible for driving a specific lock protocol. Furthermore, we demonstrated how we could extend our approach initially designed for taDOM protocols to also support other locking approaches in our prototype XTC to cross-compare foreign protocols and to prove the superiority of our protocols with empirical tests in an identical system configuration.

Moreover, we presented further refinements of our encapsulation design, which allows us to easily control and optimize the runtime behavior of our lock protocols without making concessions to the encapsulation and exchangeability properties. Finally, we outlined some possibilities to customize the protocols for special application scenarios. In our future work, we will focus on improvements concerning the efficient evaluation of declarative queries based on the XQuery language model as well as the self-optimizing capabilities of our XDBMS prototype.

8. REFERENCES

- [1] R. Bayer and M. Schkolnick: Concurrency of Operations on B-Trees. *Acta Informatica* 9:1–21 (1977)
- [2] G. Graefe: Hierarchical locking in B-tree indexes. *Proc. National German Database Conference (BTW 2007)*, LNI P-65, Springer, pp. 18–42 (2007)
- [3] J. Gray: Notes on Database Operating Systems. In *Operating Systems: An Advanced Course*, Springer-Verlag, LNCS 60: 393–481 (1978).
- [4] J. Gray and A. Reuter: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann (1993)
- [5] M. Haustein and T. Härder: An Efficient Infrastructure for Native Transactional XML Processing. *Data & Knowledge Engineering* 61:3, pp. 500–523 (2007)
- [6] M. Haustein and T. Härder: Optimizing lock protocols for native XML processing. To appear in *Data & Knowledge Engineering* 2008.
- [7] S. Helmer, C.-C. Kanne, and G. Moerkotte: Evaluating Lock-Based Protocols for Cooperation on XML Documents. *SIGMOD Record* 33:1, pp. 58–63 (2004)
- [8] P. O’Neil, E. O’Neil, S. Pal, I. Cseri, G. Schaller, N. Westbury. *ORDPATHS: Insert-Friendly XML Node Labels*. *Proc. SIGMOD Conf.*: 903–908 (2004)
- [9] Document Object Model (DOM) Level 2 / Level 3 Core Specific., W3C Recommendation
- [10] XQuery 1.0: An XML Query Language. <http://www.w3.org/XML/XQuery>
- [11] XQuery Update Facility. <http://www.w3.org/TR/xqupdate>

Database servers tailored to improve energy efficiency

Goetz Graefe

Hewlett-Packard Laboratories

ABSTRACT

Database software can be tailored for specific application domains and their required functionality, for specific hardware and its characteristics, or for other purposes. This brief paper collects issues and techniques required or desirable for making a server-class database management system energy-efficient. The opportunities go far beyond cost functions and general performance improvements. Topics include additional layers in the memory hierarchy, I/O optimizations, data format, scheduling, adaptive query plan execution, and self management in novel ways.

This paper represents a challenge rather than a solution. Promising approaches and techniques are indicated throughout, but the challenge is to provide tools for building and tailoring experimental database server software that enables research into energy-efficient database management.

1 INTRODUCTION

Electricity is expensive in terms of both direct costs and its impact on the environment. For example, in the US the direct cost is about \$1 per Watt per year ($\$1/W/y \div 365d/y \div 24h/d = 11.4¢/kWh$). Rates for large industrial users are substantially lower (in a few places as low as $1¢/kWh$ [Q 07]), but if customer costs for power transformation, air conditioning, uninterruptible power supply, and their maintenance are also considered, \$1 per Watt per year is again a reasonable first approximation. A SATA disk drive draws about 12W during operation and therefore costs about \$12 per year. Over the lifetime of a disk drive (3-6 years), its energy costs are similar to its initial purchasing cost and should be given similar consideration. Other components also have roughly similar costs for initial purchasing and lifetime power, including processors, memory, main boards, and power supplies.

Sites of new data centers have been chosen by proximity to electricity generation as well as to water for cooling, for example Google in The Dalles, Oregon and Yahoo and Microsoft in Quincy, Washington – all near the Columbia River and hydroelectric dams. At the same time, hardware designers consider power and cooling among their greatest current challenges [RRT 08]. Some current hardware consumes about 10kW per 19" rack. Power has become a significant component of data center costs, not to mention its impact on an organization's reputation and on the global environment.

While software for mobile (battery-powered) devices may include techniques targeted at energy efficiency, current server software typically does not. For database serv-

ers, some techniques aimed at general performance and scalability may also have incidental benefits in power efficiency, e.g., data compression, but designing database servers for energy efficiency seems to hold untapped potential.

One can argue whether an improvement of 25% is worth the required effort. For energy efficiency as for traditional performance metrics, 25% performance improvement due to software changes does not seem an impressive research break-through. In sales situations, however, 25% is an important improvement not only in performance and scalability but also in energy savings and costs. 25% certainly make a difference for a notebook user about to run out of battery power or for a data center operator about to run out of transformation (power supply) or cooling capacity. Given the early stage of research into software tailored to energy efficiency, it seems advisable to explore multiple directions, pursue any improvements readily obtained, and look for areas worthy of future focus.

Performance improvements, e.g., a new generation of CPUs or a faster sort algorithm, usually imply less energy required for a given amount of work. On the other hand, the goal here is to focus on opportunities for energy efficiency beyond those achieved by new hardware and its higher performance.

The purpose of this paper is to challenge software and tools developers to recognize this opportunity and to unlock its potential. Section 2 lists various hardware features that enable energy savings; many of these features require appropriate software tailored to control or exploit the hardware. Section 3 identifies opportunities for tailoring data management software for optimal energy efficiency. Section 4 briefly points out some opportunities for self-management tailored to energy efficiency. Section 5 offers some preliminary conclusions on tailoring data management software for energy efficiency.

2 HARDWARE

Hardware efforts towards energy efficiency are well known, largely due to their importance for mobile devices and portable computers. These include dynamic voltage and frequency adjustments of processors, temperature-controlled fans, power-saving modes for disk drives (which used to be marketed as noise reduction), and multi-core processors instead of ever-higher clock rates.

The most appropriate hardware differs substantially whether raw performance or energy efficiency is the primary goal [RSR 07]. The definition of a first benchmark is a welcome first step. Other benchmarks for entire work-

loads in online transaction processing and business intelligence remain open opportunities. The most immediate approach might be definition of energy efficiency metrics for established benchmarks such as those defined by the TPC (www.tpc.org).

Proposals for server hardware include enabling and disabling memory banks, substituting flash memory devices for rotating disks, and virtualization. Virtualization of processing and storage may be useful for power savings if implemented and controlled appropriately.

The obvious question is how data management software can cooperate with and exploit such techniques, and what other hardware-oriented techniques data management software can contribute.

For example, assuming a homogeneous multi-core processor, what software techniques are required to split even small tasks into concurrent actions with minimal coordination overhead? In a heterogeneous multi-core processor (with some cores optimized for performance and some for energy efficiency), what scheduling techniques should assign actions to cores?

Flash memory can serve as extended (slow) RAM or as extended (fast) disk storage [G 07a]. The former usage, perhaps as virtual memory controlled by the operating system, may be possible without software changes. In a database server, the latter usage seems to more appropriate because flash memory can serve as persistent storage. It requires adoption of techniques from log-structured file systems [RO 92], possibly in the form of write-optimized B-trees [G 04]. Log-structured file systems and write-optimized B-trees can subsume the “wear leveling” usually implemented in flash devices with “flash translation layer” and can also enable more energy-efficient RAID levels and usage. Erasing large blocks of flash storage is very similar to space reclamation in log-structured file systems over RAID storage.

Flash memory may also benefit from other changes in the database server software. For example, is asynchronous I/O still useful in terms of performance or energy? What is the policy for page movement between flash storage and disk storage, and how is the policy implemented? The data structures for the policy, e.g., a linked list for an LRU-like policy, should be maintained in RAM and could be re-initialized rather than recovered after a system crash.

Another approach attempts to reduce RAM and its high energy consumption by using flash memory. With very short access times, flash memory may serve as backing store for virtual memory, but in database query processing, it might be even more efficient (and thus energy-efficient) to employ index nested loops join rather than to rely on virtual memory for a large temporary data structure such as a hash table in hash join or hash aggregation.

Perhaps the most immediate opportunity for energy savings through carefully crafted database server software

is highly reliable storage from lower-reliability and low-power components. Due to the “small write penalty” of RAID-4, -5, and -6 (dual redundancy surviving any dual failure), many users and vendors prefer RAID-1 (mirroring). Mirroring doubles power consumption for writes but not for reads. It also doubles power and cooling needs during idle times. In contrast, RAID-5 merely increments power and cooling during writes and during idle; reads are as efficient as with raw devices. RAID-6 can achieve the same cost RAID-5 by using arrays twice as large, with no loss in reliability. RAID-6 using energy-efficient 2.5” drives may match traditional 3.25” enterprise drives in performance and reliability with substantial energy savings. A hot spare can improve mean time to repair in any RAID level; an active hot spare can even improve the performance during normal operation in RAID-5 and -6.

The challenge for tailoring data management software is to enable the appropriate software techniques that enable and complement these array arrangements.

3 SOFTWARE

Although perhaps not obvious, database server software can contribute a fair amount to energy-efficiency of database servers. The common theme here is that tailoring data management software to energy efficiency is a required next step in its evolution as much as, for example, support for semi-structured and unstructured data and support for novel hardware such using flash memory as intermediate layer in the memory hierarchy. For the time being, perhaps the most appropriate goal of tailoring database server software for energy efficiency is to enable experimentation. Factoring the software architecture and its individual components, careful design using long-lived abstract interfaces and alternative concrete implementations, and separation of policies and mechanisms are traditional implementation techniques that require even more attention and perhaps tool support than in the past.

3.1 Query optimization

In query optimization, the cost calculations can compare plans based on energy consumption rather than performance or processing bandwidth. The result may be different break-even points for alternative query execution plans. For example, index nested loops join may be chosen over hash join in spite of higher response times, because in a hash join, scan (I/O) and hashing (CPU) typically overlap and thus draw power simultaneously, whereas index nested loops join usually switches back and forth between CPU processing and I/O. The core issue, of course, is that with respect to power, concurrency and asynchronous operations cost the same as sequential or synchronous operation [ZEL 02], whereas traditional performance-oriented query optimization careful hid concurrency and parallelism in its delay-oriented cost models.

Effective heuristic guidance quickly establishes tight cost bounds that enable safe pruning of alternative plans guaranteed to be non-optimal. Heuristic pruning when optimizing for energy efficiency may be different from heuristic pruning for performance or scalability. As for traditional cost goals, bounded-loss pruning continues to apply to both alternative plans and complementary plans. For example, ignoring cost differences of 5% permits pruning when an alternative plan costs at least 95% of the best known plan and when the second input of a join when its cost is less than 5% of the cost of the first input.

Perhaps most importantly, query optimization must consider at which layer in the memory hierarchy an index is available [RD 05]. In traditional query optimization, if the I/O bandwidth matches or exceeds the bandwidth of predicate evaluation, scanning an index on disk costs the same as scanning the same index in memory. In query optimization for energy efficiency, this is clearly not the case. This example also suggests the need for novel query optimization techniques [RD 05] and for novel dynamic query execution plans that choose based on memory residency at run-time or start-up-time. In other words, software design methodology and tools ought to permit preserving functionality from (query) compile-time to (plan) run-time at will, e.g., cardinality estimation, cost calculation, and plan choices.

In order to make optimal use of indexes available in memory, B-trees must be exploited in all ways both for search and for sort order. For example, an index with keys $\langle a, b, c, d, e \rangle$ can support predicates on only $\langle b \rangle$ and $\langle d \rangle$ (no restriction on $\langle a \rangle$ and $\langle c \rangle$) by efficiently enumerating the distinct values of $\langle a \rangle$ and of $\langle a, b, c \rangle$ [LJB 95]. Moreover, the same index enables efficient scans ordered on $\langle a, e \rangle$ by sorting $\langle e \rangle$ values for each distinct value of $\langle a \rangle$, on $\langle b \rangle$ by merging runs defined by distinct values of $\langle a \rangle$, on $\langle a, c \rangle$ by combining these techniques, and on $\langle a, c, e \rangle$ by applying them multiple times. Of course, traditional set operations on indexes (intersection, union, difference), single-table index join (two indexes of the same table together cover a query), and multi-table index join (indexes on different tables reduce fetch operations from one or more tables) should also be supported and exploited for minimal energy use. Adding a large variety of possible query execution plans even for simple queries or simple components of complex queries adds substantial complexity to query optimization, in particular the plan search and its heuristics for search guidance and for pruning. Development, testing, and verification of search and of heuristics seem underdeveloped fields in software engineering, at least in the practice of database system development.

3.2 Scheduling

As hardware designers employ multi-core processors for energy efficiency, designers and implementers of database software must react, not only to make database server

software perform well on such processors but also to contribute to energy efficiency. In particular, the granularity of parallelism should become finer. For example, one thread might prefetch needed data into the cache while another thread performs the actual work, one thread might decompress data while another thread performs predicate evaluation, or scanning and predicate evaluation within a single page may be divided into multiple tasks assigned to multiple threads.

Another scheduling consideration is CPU performance. If predicate evaluation for a page can be completed faster than the next page can be read from disk, i.e., if CPU bandwidth is higher than I/O bandwidth, energy can be saved by reducing CPU clock frequency, voltage, power consumption, and cooling requirement. Complementary tailoring of database management system and operating system, e.g., providing mechanisms in the operating system to be governed by policies in the database management system, might out-perform generic solutions OS-centric [MB 06] in terms of both system throughput and energy savings. One such design, albeit clearly not taking complementary tailoring to its limits, is cooperative I/O proposed by Weissel et al. [WBB 02].

Parallel query execution has two contradicting effects on energy efficiency and thus requires new investigations and new heuristics for query optimization. On one hand, parallelism introduces additional effort compared to serial query execution, for example, data transfer, thread creation, flow control, and other coordination tasks. Thus, it seems that parallel query execution always requires more energy than serial execution. On the other hand, parallel query execution permits each processor to run at much lower frequency and voltage without overall performance loss. Given that energy and cooling needs are not linear with voltage and frequency, parallel query execution may actually save energy despite adding actual work. This is very fortunate given the recent hardware trends towards highly parallel multi-core processors. For optimal energy efficiency, very dynamic control of the degree of parallelism is probably required.

3.3 Physical database design

Compression has been considered with respect to performance, e.g., in the “ten byte rule” [GP 97], but can also be guided by energy-efficiency by comparing the additional processor energy for compression and de-compression with the energy saved in the I/O subsystem due to less volume. Tailoring for optimal energy efficiency requires a repertoire of compression methods, from simple value truncation (e.g., trailing spaces), de-duplication (applied per field, per record, per page, per index, or per table), and order-preserving compression (Huffman, arithmetic).

One specific compression technique for non-unique non-clustered indexes employs bitmap instead of traditional lists of row identifiers. Interestingly, if run-length encoding

is employed as bitmap compression, the counters are quite similar to compression of lists of row identifiers using the arithmetic difference between row identifiers [G 07].

Designed to reduce the amount of I/O in large range scans, column stores have recently received renewed attention [SAB 05]. For exact-match queries, both in OLTP applications and in BI applications using materialized and indexed views, merged indexes [G 07] can provide master-detail clustering of related records and thus save I/O and space in the buffer pool. Multi-dimensional index such as UB-trees [RMF 00] deserve renewed research with respect to energy efficiency in BI applications as they may avoid both large single-dimensional range scans and index intersection operations.

3.4 I/O scheduling

I/O scheduling and placement offer further opportunities for energy savings. For example, write-only disk caches promise to reduce disk accesses at the expense of introducing write delays. Log-structured file systems can reduce disk accesses while introducing only a minimal write delay. Write-optimized B-trees permit adaptively combining read-optimized and write-optimized operation, and they do not depend on cleaning entire large disk segments before writing. Tailoring database indexing software requires appropriate options during software assembly, data definition, and adaptively during database operation. Moreover, a specifically tailored database management system ought to be combined with an operating system tailored in an appropriate and complementary way.

If multiple concurrent query operations scan the same or similar data, sharing the scan can improve performance but also save energy. There are multiple policies for sharing scans [ZHN 07], but sharing can be pushed even further. An obvious opportunity is caching results of the inner plan in nested iteration [G 03]. Stop-and-go operation such as sorting and hash join could present another opportunity. As they materialize an intermediate result in memory or on disk, such an intermediate result could be shared. In a sort operation, for example, each consumer might have to repeat the final merge step, but it would save computation of the input, run generation, and any intermediate merge steps.

There are also savings opportunities in writing. For example, write off-loading [NDR 08] directs write operations to differential files [SL 76] on alternative disks rather than spin up powered-down disks. This technique can be further improved in a RAID-1 environment (mirroring) where one mirror may be powered down, the other mirror can serve read requests, and a differential file elsewhere satisfies write requests. More techniques of this kind will undoubtedly be invented and require evaluation in experimental and production environments.

In general, buffer pool management can contribute to energy management by means of sharing, asynchronous prefetch and write operations, and sharing. Dynamic plans

that choose the alternative plan based on current buffer pool contents could save substantial energy for I/O. The tradeoffs and break-even points between sharing and not sharing probably differ depending on the devices involved, the query execution plans executing concurrently, and the relative importance of energy savings versus performance. Again, until these techniques and tradeoffs are better understood, tailor-made data management software should offer multiple techniques together or one at-a-time.

3.5 Update techniques

Update operations offer an unusual savings opportunity. Updates need to be applied to all indexes, materialized views, summary synopses such as histograms, etc., but not necessarily immediately. In other words, deferred index maintenance can be employed to alleviate temporary load spikes and thus to reduce peak power consumption. In parallel systems, deferred index maintenance employed in only some of the nodes can reduce not only peak power demands but also temporary load imbalance.

A special form of deferred index maintenance captures all updates in the target index but not necessarily at their final, search-optimized locations: a partitioned B-tree [G 03a] may merely append incoming data changes and thus employ partitions in the same way differential files employ master and delta files [SL 76]. Additional performance and energy savings may be possible if a single step applies the updates of many user transactions to the master. This technique has been explicitly proposed for column stores based deferred changes captured in traditional row format [SAB 05] but it seems independent of the storage formats involved.

Other techniques for load spikes are mechanisms to “pause and resume” all utilities such as index creation, defragmentation, histogram refresh, database consistency checks, etc. Loading new data (“roll-in”) and erasing out-of-date data (“roll-out”) may permit “pause and resume” based on subsets of the change set, based on subsets of maintenance operations such as index update and verification of integrity constraints, or both.

4 SELF-MANAGEMENT

Designing database software for energy efficiency also affects database utilities, for example defragmentation, reorganization, and tuning the physical database design with indexes, partitioning, materialized views, and placement of data in the memory hierarchy.

These utilities must consider both costs and benefits with respect to energy efficiency as well as timing of the operation relative to both peak loads and idle time. Utilities should not force operating processors in high-power modes and should not prevent powering down a device.

5 SUMMARY AND CONCLUSIONS

In summary, while power has traditionally been a topic mostly for mobile devices, power and cooling contribute substantially to the cost of data centers. Improvements in energy-efficiency can have substantial positive effects on operational costs, on an organization's reputation, and of course on the global environmental.

Hardware designers have risen to the challenge of creating more energy-efficient components, including multi-core processors running at moderate frequencies, flash memory with low access times and practically zero idle power, and sensor-controlled fans and cooling. Designers and implementers of database software, on the whole, have not yet realized their potential contributions, neither those that exploit hardware improvements designed for energy efficiency nor those that save energy independent of the hardware employed and its energy-saving features.

The present paper outlines challenges and opportunities for specific software techniques that may reduce energy requirements of database servers. Techniques for energy efficiency, not only for databases on mobile devices but also on servers, is a challenge for database researchers, vendors, and all those offering software tools for designing, implementing, and tailoring data management software.

Tailoring data management software will remain a never-ending challenge in order to support novel applications and their data such as spatio-temporal and text mining, to exploit novel hardware such as multi-core processors and flash memory as additional layer in the memory hierarchy, and to serve novel operational needs such as grid computing, autonomic self-management, virtualization, software-as-a-service, and energy efficiency. Thus, any investment in research and development of tools for tailoring database management systems will retain their value for a very long time.

REFERENCES

- [G 03] Goetz Graefe: Executing Nested Queries. BTW Conf. 2003: 58-77.
- [G 03a] Goetz Graefe: Sorting and Indexing with Partitioned B-Trees. CIDR 2003.
- [G 04] Goetz Graefe: Write-Optimized B-Trees. VLDB 2004: 672-683.
- [G 07] Goetz Graefe: Master-detail clustering using merged indexes. Informatik•Forschung und Entwicklung 21(3-4): 127-145 (2007).
- [G 07a] Goetz Graefe: The five-minute rule twenty years later, and how flash memory changes the rules. DaMoN 2007.
- [GP 97] Jim Gray, Gianfranco R. Putzolu: The 5 Minute Rule for Trading Memory for Disk Accesses and The 10 Byte Rule for Trading Memory for CPU Time. ACM SIGMOD 1987: 395-398.
- [LJB 95] Harry Leslie, Rohit Jain, Dave Birdsall, Hedieh Yaghmai: Efficient Search of Multi-Dimensional B-Trees. VLDB 1995: 710-719.
- [MB 06] Andreas Merkel, Frank Bellosa: Balancing power consumption in multiprocessor systems. EuroSys 2006: 403-414.
- [NDR 08] Dushyanth Narayanan, Austin Donnelly, Antony Rowstron: Write Off-Loading: Practical Power Management for Enterprise Storage. FAST 2008.
- [Q 07] http://quincywashington.us/utilities_rates.html, retrieved December 26, 2007.
- [RD 05] Ravishankar Ramamurthy, David J. DeWitt: Buffer-pool Aware Query Optimization. CIDR Conf. 2005: 250-261.
- [RMF 00] Frank Ramsak, Volker Markl, Robert Fenk, Martin Zirkel, Klaus Elhardt, Rudolf Bayer: Integrating the UB-Tree into a Database System Kernel. VLDB 2000: 263-272.
- [RO 92] Mendel Rosenblum, John K. Ousterhout: The Design and Implementation of a Log-Structured File System. ACM TODS 10(1): 26-52 (1992).
- [RRT 08] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, Xiaoyun Zhu: No "Power" Struggles: Coordinated Multi-level Power Management for the Data Center. ASPLOS 2008.
- [RSR 07] Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan, Christos Kozyrakis: JouleSort: a balanced energy-efficiency benchmark. ACM SIGMOD 2007: 365-376.
- [SAB 05] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, Stanley B. Zdonik: C-Store: A Column-oriented DBMS. VLDB 2005: 553-564.
- [SL 76] Dennis G. Severance, Guy M. Lohman: Differential Files: Their Application to the Maintenance of Large Databases. ACM TODS 1(3): 256-267 (1976).
- [WBB 02] Andreas Weissel, Björn Beutel, Frank Bellosa: Cooperative I/O: A Novel I/O Semantics for Energy-Aware Applications. OSDI 2002.
- [ZEL 02] Heng Zeng, Carla Schlatter Ellis, Alvin R. Lebeck, Amin Vahdat: ECOSystem: managing energy as a first class operating system resource. ASPLOS 2002:123-132.
- [ZHN 07] Marcin Zukowski, Sándor Héman, Niels Nes, Peter A. Boncz: Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. VLDB 2007: 723-734.

Generating Highly Customizable SQL Parsers

Sagar Sunkle, Martin Kuhlemann, Norbert Siegmund,
Marko Rosenmüller, Gunter Saake
School of Computer Science
University of Magdeburg
39106 Magdeburg, Germany
{ssunkle,mkuhlema,nsiegmun,rosenmue,saake}@ovgu.de

ABSTRACT

Database technology and the Structured Query Language (SQL) have grown enormously in recent years. Applications from different domains have different requirements for using database technology and SQL. The major problem of current standards of SQL is complexity and unmanageability. In this paper we present an approach based on software product line engineering which can be used to create customizable SQL parsers and consequently different SQL dialects. We give an overview of how SQL can be decomposed in terms of *features* and how different features can be composed to create tailor-made parsers for SQL.

General Terms

Design, Languages

Keywords

Tailor-made Data Management, Embedded Systems, Feature-oriented Programming

1. INTRODUCTION

Since its modest beginnings in the 70's, database technology has exploded into every area of computing. It is an integral part of any modern application where different kinds of data need to be stored and manipulated. All major database technology vendors have adopted a very general approach, combining functionality from diverse areas in one database product [8]. Likewise, *Structured Query Language (SQL)*, the basis for interaction between database technology and its user, has grown enormously in its size and complexity [6, 13]. Starting with the standard selection-projection-join queries and aggregation, SQL now contains a number of additional constructs pertaining to new areas of computing to which database technology has been introduced [8]. All major database vendors conform to ISO/ANSI SQL standards at differing levels, maintaining the syntax suitable for their products, thus further increasing the complexity of learning and using SQL. Requirements like performance, tuning, configurability, etc., differ from domain to domain, thus needing a different

treatment of database technology in different applications. Characteristics of SQL like data access, query execution speed, and memory requirements differ between application domains like data warehouses and embedded systems. Various researchers have shown the need for configurability in many areas of DBMS [6, 13, 17]. We therefore need the ability to select only the functionality we need in database products in general and SQL in particular.

Software product line engineering (SPLE) is a software engineering approach that considers aforementioned issues of products of similar kind made for a specific market segment and differing in features. A set of such products is called a *software product line (SPL)* [18]. In SPLE, products are identified in terms of features which are user-visible characteristics of a product [12, 9]. Decomposing SQL in terms of features, that can be composed to obtain different SQL dialects, can be beneficial and insightful not only in managing features of SQL itself but also in database technology of embedded and real time systems. A customizable SQL and a customizable database management system is a better option than using conventional database products for embedded systems. Embedded and real time systems have different characteristics and performance requirements than the general computing systems due to restricted hardware and frequent use of certain kinds of queries such as projection and aggregation [6]. Embedded systems like various hardware appliances, peer-to-peer, and stream based architectures for embedded devices use shared information resources and require declarative query processing for resource discovery, caching and archiving, etc. [13]. Query processing for sensor networks requires different semantics of queries as well as additional features than provided in SQL standards [14]. A feature decomposition of SQL can be used to create a 'scaled down' version of SQL appropriate for such applications, by establishing a product line architecture for SQL variants.

In this paper, we propose that SPL research is capable of providing answers to the problems of managing features in database products such as SQL engines.

We present an approach for a customizable SQL parser based on product line concepts and a decomposition of SQL into features. We show how different features of SQL are obtained and how these features can be composed to create different SQL parsers.

2. BACKGROUND

2.1 Structured Query Language

SQL is a database query language used for formulating statements that are processed by a database management system to create and maintain a database [21]. The most commonly used SQL query is the SELECT statement which can retrieve data from one or more tables in a database. This data can be restricted using conditional statements in the WHERE clause. SELECT can group related data using the GROUP BY clause and restrict the grouped data with the HAVING clause. It can order or sort the data based on different columns using the ORDER BY clause [15]. SQL contains many statements to create and manipulate database objects. Since its first standardization in 1986, more and more functionality is being included in SQL in each subsequent standard. The latest edition of the SQL standard, referred to as SQL:2003, supports diverse functionality such as call level interfacing, foreign-data wrappers, embedding SQL in Java, business intelligence and data warehousing functions, support for XML, new data types, etc.

The vast scope of SQL's functionality has led many researchers to advocate the usage of a 'scaled down' version of SQL, especially for embedded systems [6, 13, 8]. Embedded systems have many hardware limitations such as small RAM, small stable storage, and high data read/write ratio. Also the applications where embedded systems are used, e.g., healthcare and bank cash cards, need only a small set of queries like select, project, views, and aggregations. A standard called *Structured Card Query Language (SCQL)* by ISO considers inter-industry commands for use in smart cards [11] with restricted functionality of SQL. Some database systems and SQL engines, distinguished as 'tiny', have been proposed to address this issue, e.g., the TinyDB¹ database management system for sensor networks. TinyDB contains TinySQL language for querying sensor networks. TinySQL has a limited functionality compared to SQL such as single table in FROM clause, no column alias in SELECT clause, and sensor networks specific query constructs such as epoch duration and sample period clause.

While the standardization process shows how SQL has increased in size and complexity in terms of features provided, efforts for 'scaled down' versions indicate a need to control and manipulate features of SQL.

¹<http://telegraph.cs.berkeley.edu/tinydb/>

2.2 Software Product Line Engineering

SPLE aims at developing software applications by identifying and building reusable assets in the domain engineering and implementing mass customization in the application engineering [18]. The feature modeling activity applied to systems in a domain to *capture commonalities and variabilities* in terms of features is known as *feature-oriented decomposition*. Feature-oriented decomposition is carried out in the analysis phase of domain engineering. A feature is any end-user-visible, distinguishable and functional or non-functional characteristic of a concept that is relevant to some stakeholder [9, 10]. In modeling the features of SQL (specifically SQL:2003), we take the view of features as the end-user-visible and distinguishable characteristics of SQL. *Feature diagrams* [12, 9] are used to model features in a hierarchical manner as a tree. The root of the tree represents a concept and the child nodes represent features. The hierarchical structure of a feature diagram indicates that there is a parent child relationship between the feature nodes. A feature diagram contains various types of features such as *mandatory*, *optional*, *alternative* features, *AND* features, and *OR* features. A *feature instance* is a description of different feature combinations obtained by including the concept node of the feature diagram and traversing the diagram from the concept. Depending on the type of the feature node, the node becomes part of the instance description [9].

2.3 Feature-Oriented Programming

Feature-oriented programming (FOP) [19] is the *study of feature modularity* and how to use it in *program synthesis* [2]. It treats features as first-class entities in design and implementation of a product line. A complex program is obtained by adding features as incremental details to a simple program. The basic ideas of FOP were first implemented in GenVoca [4] and *Algebraic Hierarchical Equations for Application Design (AHEAD)* [5]. AHEAD uses the *Jak* language, which is a superset of the Java language, for feature-oriented programming. In AHEAD, a programming language and language extensions are defined in terms of a base grammar and extension grammars. Grammars specific to both the language and the language extensions are described using the *Bali* grammar specification language. Bali can be used to specify sub-grammars that can be shared and reused. As a grammar specification language, Bali allows specifying extra constructs for BNF specification such as labels to name the production rules in a grammar. Grammars written in Bali can be composed to specify language extensions and language combinations². A Bali grammar can import definitions for non-terminals from other grammars. Syntax extension and

²<http://www.cs.utexas.edu/users/schwartz/>

the corresponding semantic actions are implemented separately using the Javacc³ parser generator and the Jak language respectively. Features are treated as collaborations among classes and composed using *Jampack* and *Mixin* tools [5].

3. CUSTOMIZABILITY FOR SQL

Customizability for SQL means that only the needed functionality, such as only some specific SQL statement types, is present in the SQL engine. The concepts of features and product lines can be applied to the SQL language so that composition of different features leads to different parsers for SQL. We base our approach for creating a customizable parser for SQL:2003 on the idea of composing sub-grammars to add extra functionality to a language from the Bali approach as stated above. For a customizable SQL parser, we consider LL(k) grammars which are a type of context free grammars. Terminal symbols are the elementary symbols of the language defined by such a grammar while the nonterminal symbols are set of strings of terminals also known as syntactic variables [1]. Terminals and nonterminals are used in production rules of the grammar to define substitution sequences. From a features and feature diagrams perspective, the process of feature-oriented decomposition and feature implementation is realized in two broad stages [20]. In the first stage, the specification of SQL:2003 is modeled in terms of feature diagrams. Given the feature diagram of a particular construct of SQL:2003 that needs to be customized, we need to create the LL(k) grammar that captures the feature instance description. This ensures that the specific feature which we want to add to the original specification is included as well. In the second stage, having obtained LL(k) grammars for the base and extension features separately, we compose them to obtain the LL(k) grammar which contains the syntax for both the base and extension features. With a parser generator, we obtain a parser which can effectively parse the base as well as extension specification for a given SQL construct.

We add semantic actions to the parser code thus generated using Jak and other feature-oriented programming tools, effectively creating a SQL:2003 preprocessor. In the following sections we explain decomposition of SQL:2003 into features and composition of these features.

3.1 Decomposing SQL:2003

For the feature-oriented decomposition of SQL:2003, we use various SQL:2003 standards ISO/IEC 9075 - (n):2003 which define the SQL language. SQL Framework [16] and SQL Foundation [15] encompass the min-

imum requirements of the SQL. Other parts define extensions.

SQL statements are generally classified by their functionality as data definition language statements, data manipulation language statements, data control language statements, etc. [15]. Therefore, we arranged the top level features for SQL:2003 at different levels of granularity with the basic decomposition guided by the classification of SQL statements by function as found in [15]. The feature diagram for features of SQL:2003 is based on the BNF grammar specification of SQL:2003 and other information given in SQL Foundation.

We use the BNF grammar of SQL for constructing the feature diagrams based on the following assumptions:

- The complete SQL:2003 BNF grammar represents a product line, in which various sub-grammars represent features. Composing these features creates products of this product line, namely different variants of SQL:2003.
- A nonterminal may be considered as a feature only if the nonterminal clearly expresses an SQL construct. Mandatory nonterminals are represented as mandatory features. Optional nonterminals are represented as optional features.
- The choices in the production rule are represented as OR features.
- A terminal symbol is considered as a feature only if it represents a distinguishable characteristic of the feature under consideration apart from the syntax (e.g., DISTINCT and ALL keywords in a SELECT statement signify different features of the SELECT statement).

The grammar given in SQL Foundation is useful in understanding the overall structure of an SQL construct, or what different SQL constructs constitute a larger SQL construct. This approach may also be useful in general to carry out a feature decomposition of any programming language as the grammar establishes the basic building blocks of any programming language. The most coarse-grained decomposition is the decomposition of SQL:2003 into various constituent packages. We have chosen to further decompose SQL Foundation, since it contains the core of SQL:2003. Overall 40 feature diagrams are obtained for SQL Foundation with more than 500 features. Other extension packages of SQL:2003 can be similarly decomposed. Figures 1 and 2 show the features *Query Specification* (representing SELECT statement) and the feature *Table Expression* respectively.

To obtain the sub-grammars corresponding to features, we refer to the feature diagram for the given feature. Based on the feature diagram, we create LL(k) grammars for each feature in the feature instance description using the original SQL:2003 BNF specification for *Query Specification* (cf. Section 7.12 in SQL

³<https://javacc.dev.java.net/>

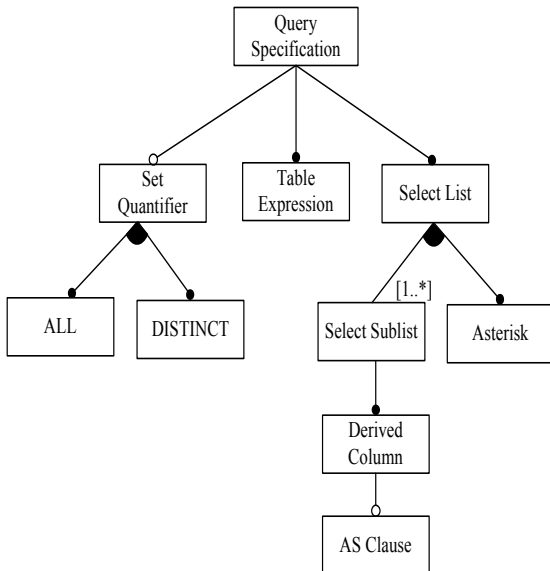


Figure 1: Query Specification Feature Diagram.

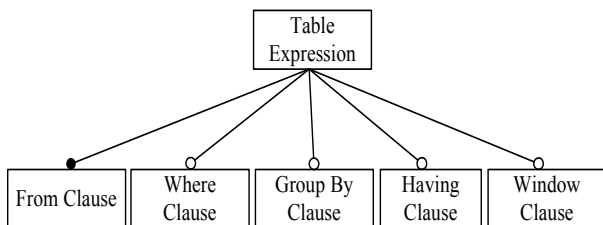


Figure 2: Table Expression Feature Diagram.

Foundation). These sub-grammars are used later during composition to obtain a grammar that can parse various SQL constructs represented by corresponding features. We represent a grammar and the tokens separately. Accordingly, for each sub-grammar we also create a file containing various tokens used in the grammar.

3.2 Composing SQL:2003 Features

During our work on a customizable parser for SQL:2003, we found that the tool *Balicomposer*, which is used for composing the Bali grammar rules, is restrictive in expressing the complex structure of SQL grammar rules. Bali uses its own notation for describing language and extension grammars which are converted to LL(k) grammars as required by the Javacc parser generator. We instead use the LL(k) grammars with additional options used by the ANTLR⁴ parser generator in our prototype. In order to create an SQL engine, we also need the feature-oriented programming capability already present in the form of Jak language and Bali related tools *Mixin* and *Jampack*.

We use the feature diagrams obtained in the decom-

⁴<http://www.antlr.org/>

position to create a feature model from which different features constituting the parser may be selected. Such a parser for SQL:2003 can selectively parse precisely those SQL:2003 statements which are represented as features in feature diagrams under consideration. We explain this with an example.

Suppose that we want to create a parser for the SELECT statement in SQL:2003 represented by the *Query Specification* feature (cf. Figure 1). Specifically we want to implement a feature instance description of $\{Query\ Specification, Select\ List, Select\ Sublist\}$ (with cardinality 1), *Table Expression* (with the *Table Expression* feature instance description, $\{Table\ Expression, From\ Clause, Table\ Reference\}$ (with cardinality 1)). We would proceed as follows:

1. A feature tree of the SELECT statement presents various features of the statement to the user. Selection of different subfeatures of the SELECT statement is equivalent to creating a feature instance description.
2. To create a parser for these features, we make use of the sub-grammars and token files created during the decomposition. We compose these sub-grammars to one LL(k) grammar. Similarly, corresponding token files are composed to a single token file.
3. Using the ANTLR parser generator, we create the parser with the composed grammar. The parser code generated is specific to the features we selected in the first step. That is, it is capable of parsing precisely the features in the feature instance description for which we created LL(k) grammars.

Thus composing the sub-grammars for the *Query Specification* (cf. Figure 1) feature which represents an SQL SELECT statement, the optional *Set Quantifier* feature of Query Specification and the optional *Where Clause* feature of the *Table Expression* (cf. Figure 2) feature which itself is a mandatory feature of *Query Specification*, gives a grammar which can essentially parse a *SELECT statement with a single column from a single table with optional set quantifier (DISTINCT or ALL) and optional where clause*. This procedure can be extended to other statements of SQL:2003, first mapping features to sub-grammars and then composing them to obtain a customizable parser.

In composing LL(k) grammars we must consider the treatment of nonterminals and tokens. The treatment of nonterminals is most involved. We have provided composition mechanisms for various production rules with the same nonterminal, for composition of optional nonterminals, and for composing complex sequences of nonterminals. Production rules in the grammar may or

may not contain choices for the same nonterminal, e.g., $A: B \mid C \mid D$ and $A: B$. The composition of production rules labeled with the same nonterminal is carried as follows:

- If the new production contains the old one, then the old production is replaced with the new production, e.g., in composing $A: BC$ with $A: B$, the production B is replaced with BC .
- If the new production is contained in the old one, then the old production is left unmodified, e.g., in composing $A: B$ with $A: BC$, the production BC is retained.
- If the new and old production rules defer, then they are appended as choices, e.g., in composing $A: B$ with $A: C$, productions B and C are appended to obtain $A : B \mid C$.

We compose any optional specification within a production after the corresponding non optional specification. $A: B$ and $A : B[C]$ or $A : B$ and $A : [C]B$ can be composed in that order only. Some production rules in the grammar may contain complex lists as productions. Complex lists are of the form ' $\langle NT \rangle [\langle comma \rangle \langle NT \rangle \dots]$ '. In the composer for the prototype, if features to be composed contain a sublist and a complex list, e.g., $A: B$ and $A: B [“,” B]$ respectively, then these are composed sequentially with the sublist being composed ahead of the complex list.

A feature may require other features for correct composition. Such features constraints are expressed as 'requires' or 'excludes' conditions on features. We use the notion of composition sequence that indicates how various features are included or excluded.

4. RELATED WORK

Extensibility of grammars is also subject to current research. Batory et al. provide an extensible Java grammar based on an application of FOP to BNF grammars [3]. Our work on decomposing the SQL grammar was inspired by their work. Initially, we tried to use the Bali language and accompanied tools to decompose the SQL grammar. The language extension approach of Bali clearly separates the syntax extension and the implementation of semantic actions. However, given the complexity of SQL:2003 grammar, we cannot achieve the customizability that we need by using Bali [20]. We therefore created a new compositional approach based on ANTLR grammars.

Bravenboer et al. provide with METABORG an approach for extensible grammars [7]. Their aim is to extend an existing language with new syntax from a different language. For that reason they need a parsing approach that can handle various context free grammars simultaneously because multiple languages may

be combined producing various ambiguities. Since we are decomposing only SQL, a single language, we can use the common approach of employing a separate scanner which would be insufficient for their goal. They use *Syntax Definition Formalism (SDF)* to achieve modularity, although the modularity of SDF grammars can be attributed to scanner-less generalized LR (SGLR) parsing mechanism used in METABORG [7]. The notion of inheritable grammar modules in SDF also occurs in Bali. We provide the mechanisms of adding, removing and modifying the production rules in grammar but not inheritable grammars at this point. Disambiguation mechanism of the METABORG approach consists of priority between productions, reject/prefer mechanisms for derivations, enforcing associativity for operators. Similar disambiguation constructs are also present in ANTLR in terms of syntactic and semantic predicates.

5. CONCLUSIONS

Features of SQL:2003 are user-visible and distinguishable characteristics of SQL:2003. The complete grammar of SQL:2003 can be considered as a product line, where sub-grammars denote features that can be composed to obtain customized parsability. In addition to decomposing SQL by statement classes, it is possible to classify SQL constructs in different ways, e.g., by the schema element they operate on. We propose that different classifications of features lead to the same advantages of using the feature concept. We have created 40 feature diagrams for SQL Foundation representing more than 500 features. Other extension packages of SQL can be similarly decomposed into features. We have created different prototype parsers by composing different features. Currently we are creating an implementation model and a user interface presenting various SQL statements and their features. When a user selects different features, the required parser is created by composing these features. Although SPLE and FOP concepts have been applied to programming language extension as in Bali, our approach is unique in that these concepts have been applied to SQL:2003 for the first time to obtain customizable SQL parsers. Modularity and disambiguation constructs for given type of grammars depend largely on the parsers and parser generators used. Similarly implementation of semantics for given language depends on the generated parser. Some parsers represent production rules in the grammar as methods whereas some other parsers represent these as classes. We surmise that the best approach will depend on ease of implementation of semantics. One of our future aims is to find out what kind of modularity of grammars and what kind of parsing mechanism is most suitable for feature-oriented extension of SQL.

Acknowledgments

Norbert Siegmund and Marko Rosenmüller are funded by German Research Foundation (DFG), Project SA 465/32-1. The presented work is part of the FAME-DBMS project⁵, a cooperation of Universities of Dortmund, Erlangen-Nuremberg, Magdeburg, and Passau, funded by DFG.

6. REFERENCES

- [1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.
- [2] D. Batory. A Tutorial on Feature-oriented Programming and Product-lines. In *Proceedings of the 25th International Conference on Software Engineering*, pages 753–754, Washington, DC, USA, 2003. IEEE Computer Society.
- [3] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for Implementing Domain-Specific Languages. In *Proceedings of the International Conference on Software Reuse*, pages 143–153. IEEE Computer Society, 1998.
- [4] D. Batory and S. O’Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, 1992.
- [5] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.
- [6] C. Bobineau, L. Bouganim, P. Pucheral, and P. Valduriez. PicoDMBS: Scaling Down Database Techniques for the Smartcard. In *Proceedings of the International Conference on Very Large Data Bases*, pages 11–20. Morgan Kaufmann, 2000.
- [7] M. Bravenboer and E. Visser. Concrete Syntax for Objects: Domain-specific Language Embedding and Assimilation Without Restrictions. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 365–383. ACM Press, 2004.
- [8] S. Chaudhuri and G. Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, editors, *Proceedings of 26th International Conference on Very Large Data Bases (VLDB’00)*, pages 1–10. Morgan Kaufmann, 2000.
- [9] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
- [10] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.
- [11] International Organization for Standardization (ISO). Part 7: Interindustry Commands for Structured Card Query Language (SCQL). In *Identification Cards – Integrated Circuit(s) Cards with Contacts*, ISO/IEC 7816-7, 1999.
- [12] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.
- [13] M. L. Kersten, G. Weikum, M. J. Franklin, D. A. Keim, A. P. Buchmann, and S. Chaudhuri. A Database Striptease or How to Manage Your Personal Databases. In *Proceedings of the International Conference on Very Large Data Bases*, pages 1043–1044. Morgan Kaufmann, 2003.
- [14] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Transactions on Database Systems*, 30(1):122–173, March 2005.
- [15] J. Melton. Working Draft : SQL Foundation . ISO/IEC 9075-2:2003 (E) 5WD-02-Foundation-2003-09, ISO/ANSI, 2003.
- [16] J. Melton. Working Draft : SQL Framework . ISO/IEC 9075-1:2003 (E) 5WD-01-Framework-2003-09, ISO/ANSI, 2003.
- [17] D. Nyström, A. Tešanović, M. Nolin, C. Norström, and J. Hansson. COMET: A Component-Based Real-Time Database for Automotive Systems. In *Proceedings of the Workshop on Software Engineering for Automotive Systems*, pages 1–8. IEEE Computer Society, 2004.
- [18] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering : Foundations, Principles, and Techniques* . Springer, 1998.
- [19] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer, 1997.
- [20] S. Sunkle. Feature-oriented Decomposition of SQL:2003. Master’s thesis, Department of Computer Science, University of Magdeburg, Germany, 2007.
- [21] R. F. van der Lans. *Introduction to SQL: Mastering the Relational Database Language, Fourth Edition/20th Anniversary Edition*. Addison-Wesley Professional, 2006.

⁵<http://fame-dbms.org>

Architectural Concerns for Flexible Data Management

Ionut Emanuel Subasu, Patrick Ziegler, Klaus R. Dittrich, Harald Gall
Department of Informatics, University of Zurich
{subasu,pziegler,dittrich,gall}@ifi.uzh.ch

ABSTRACT

Evolving database management systems (DBMS) towards more flexibility in functionality, adaptation to changing requirements, and extensions with new or different components, is a challenging task. Although many approaches have tried to come up with a flexible architecture, there is no architectural framework that is generally applicable to provide tailor-made data management and can directly integrate existing application functionality. We discuss an alternative database architecture that enables more lightweight systems by decomposing the functionality into services and have the service granularity drive the functionality. We propose a service-oriented DBMS architecture which provides the necessary flexibility and extensibility for general-purpose usage scenarios. For that we present a generic storage service system to illustrate our approach.

1. INTRODUCTION

Current Database Management Systems (DBMS) are extremely successful software products that have proved their capabilities in practical use all over the place. Also from a research point of view they show a high degree of maturity and exploration. Despite all progress made so far and despite all euphoria associated with it, we begin to realize that there are limits to growth for DBMS, and that flexibility is an essential aspect in DBMS architecture.

Over time DBMS architectures have evolved towards flexibility (see Figure 1). Early DBMS were mainly large and heavy-weight monoliths. Based on such an architecture, extensible systems were developed to satisfy an ever growing need for additional features, such as new data types and data models (e.g., for relational and XML data). Some of these systems allowed extensibility through application front ends at the top level of the architecture [4, 20, 22]. A similar approach is taken by aspect-oriented object database systems such as SADES [2], which support the separation of data management concerns at the top level of DBMS architecture. Based on this, further aspects can be added to

the database as required. By using a monolithic DBMS structure, however, attempts to change DBMS internals will hardly succeed, due to the lack of flexibility.

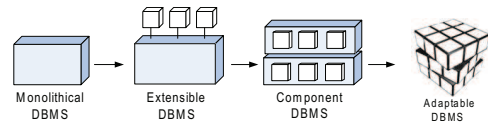


Figure 1: Evolution of DBMS architectures

In response to this, the next step in the evolution of DBMS architecture was the introduction of Component Database Systems (CDBS) [7]. CDBS allow improved DBMS flexibility due to a higher degree of modularity. As adding further functionality to the CDBS leads to an increasing number of highly dependent DBMS components, new problems arise that deal with maintainability, complexity, and predictability of system behavior.

More recent architectural trends apply concepts from interface and component design in RISC approaches to DBMS architectures [6]. RISC style database components, offering narrow functionality through well-defined interfaces, try to make the complexity in DBMS architecture more manageable. By narrowing component functionality, the behavior of the whole system can be more easily predicted. However, RISC approaches do not address the issue of integrating existing application functionality. Furthermore, coordinating large amounts of fine-grained components can create serious orchestration problems and large execution workflows.

Other approaches [13] try to make DBMS fit for new application areas, such as bio-informatics, document/content management, world wide web, grid, or data streams, by extending them accordingly. These extensions are customized, fully-fledged applications that can map between complex, application-specific data and simpler database-level representations. In this way, off-the-shelf DBMS can be used to build specialized database applications. However, adapting DBMS through a growing number of domain specific applications causes increasing costs, as well as compatibility and maintenance problems [21]. Such extensions can lead to hard wired architectures and unreliable systems [14].

The DBMS architecture evolution shows that, despite many approaches try to adapt the architecture, there is no architectural framework that is generally applicable to pro-

vide tailor made data management and directly integrate existing application functionality. In addition, none of the existing approaches can provide sufficient predictability of component behavior. Finally the ability to integrate existing application functionality cannot be achieved without requiring knowledge about component internals. These aspects, however, are essential for a DBMS architecture to support a broad range of user requirements, ranging from fully-fledged extended DBMS to small footprint DBMS running in embedded system environments. In consequence of that, the question arises whether we should still aim at an ever increasing number of system extensions, or whether it is time to rethink the DBMS approach architecturally.

In [23] we introduced a service based DBMS architecture which is based on the concepts of Service-Oriented Architecture (SOA) [15]. We argue that database services in the spirit of SOA are a promising approach to bring flexibility into DBMS architecture, as they are an adequate way to reduce or extend DBMS functionality as necessary to meet specific requirements. In this sense, we make DBMS architecture capable of adapting to specific needs, and, consequently, increase its "fitness for use".

In this paper, we focus on the notion of flexibility in DBMS architecture. We take a broad view on database architecture and present major requirements for a flexible DBMS architecture. Based on this, we propose a service oriented DBMS architecture which provides the necessary flexibility and extensibility for general-purpose usage scenarios. Instead of taking a bottom up approach by extending the architecture when needed, we use a top down approach and specialize the architecture when required. The main feature of our architecture is its flexibility and we do not primarily focus on achieving very high processing performance.

The remainder of the paper is structured as follows. In the next section we present aspects of flexibility that are relevant for extensible and flexible DBMS architectures. Section 3 introduces our Service Based Data Management System (SBDMS) architecture designed to meet these requirements. We illustrate and discuss our approach using examples in Section 4 and finally conclude in Section 5.

2. ASPECTS OF FLEXIBILITY

From a general view of the architecture we can see three main aspects that have to be addressed to achieve our goals: (1) extend the architecture with specialized functionality, (2) handle missing or erroneous parts, and (3) optimize the architecture functionality by allowing it to do the same task in different ways, according to user requirements or current system and architecture configurations. Note that while there may be other concerns, such as security, ease of maintainability, reliability, or system performance our focus in this paper is on the three aspects mentioned above, as flexibility defines the general character of an architecture.

Following the dictionary definition, "flexibility" can be interpreted as (1) admitting of being turned, bowed, or twisted without breaking or as (2) capable of being adapted [19]. This shows that there is not an exact way or metric to measure or increase the "flexibility" of an architecture. The general character of the term can also be seen in the IEEE

definition [1] which sees flexibility as the ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed.

From a different perspective, system flexibility can be seen as the system's capability of being extensible with new components and specializations. Here, "flexibility" refers to the ease with which a system or component can be modified to increase its storage or functional capacity [1]. Extensibility itself, however, does not suffice to create a flexible architecture because it neglects the case of downsizing the architecture. Despite of this, extensibility can be considered as a subcase of flexibility. If the architecture is not able to enable the appropriate changes and adapt to the environment we say that it has limited flexibility.

To provide a systematic view of the architecture flexibility, we have to consider different aspects of flexibility [16]. From the above definitions and the general character of the architecture, the following main aspects of flexibility can be considered:

- *Flexibility by selection* refers to the situation in which the architecture has different ways of performing a desired task. This is the case when different services provide the same functionality using the same type of interfaces. In this scenario no major changes at the architectural level are required.
- *Flexibility by adaptation* handles the case of absent or faulty components that cannot be replaced. Here, existing internal instances of the architecture that provide the same functionality can be adapted to cope with the new situation.
- *Flexibility by extension* allows the system to be adapted to new requirements and optimizations that were not fully foreseen in the initial design.

System quality can be defined as the degree to which the system meets the customer needs or expectations [1]. As extensions can increase the applicability of a system, flexibility and extensibility are important aspects of the quality of an architecture, which can be described as its "fitness for use". To support tailored "fitness for use", future architectures should put more emphasis on improving their flexibility and extensibility according to user needs. In this way the architecture will have the possibility to be adapted to a variety of factors, such as the environment in which the architecture is deployed (e.g., embedded systems or mobile devices), other installed components, and further available services. Some users may require less functionality and services; therefore the architecture should be able to adapt to downsized requirements as well.

3. THE SBDMS ARCHITECTURE

Today, applications can extend the functionality of DBMS through specific tasks that have to be provided by the data management systems; these tasks are called *services* and allow interoperability between DBMS and other applications

[11]. This is a common approach for web-based applications. Implementing existing DBMS architectures from a service approach can introduce more flexibility and extensibility. Services that are accessible through a well defined and precisely described interface enable any application to extend and reuse existing services without affecting other services. In general, Service Oriented Architecture (SOA) refers to a software architecture that is built from loosely coupled services to support business processes and software users. Typically, each resource is made available through independent services that can be distributed over computers that are connected through a network, or run on a single local machine. Services in the SOA approach are accessed only by means of a well defined interface, without requiring detailed knowledge on their implementation. SOAs can be implemented through a wide range of technologies, such as RPC, RMI, CORBA, COM, or web services, not making any restrictions on the implementation protocols [10]. In general, services can communicate using an arbitrary protocol; for example, a file system can be used to send data between their interfaces. Due to loose coupling, services are not aware of which services they are called from; furthermore, calling services does not require any knowledge on how the invoked services complete their tasks.

In [23] we have introduced a Service Based Data Management System (SBDMS) architecture that can support tailored extensions according to user requirements. Founding the architecture on the principles of SOA provides the architecture with a higher degree of flexibility and brings new methods for adding new database features or data types. In the following we present the architecture using *components*, *connectors*, and *configurations*, as customarily done in the field of software architectures [3].

3.1 Architectural Components

Our SBDMS architecture is organized into general available functional layers (see Figure 2), where each layer contains specialized services for specific tasks:

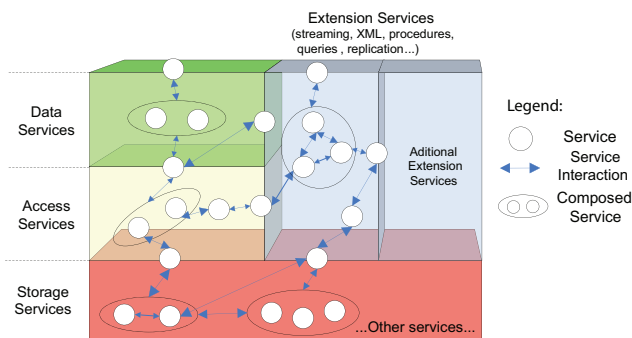


Figure 2: The SBDMS Architecture [23]

- *Storage Services* work at byte level and handle the physical specification of non-volatile devices. This includes services for updating and finding data.
- *Access Services* manage physical data representations of data records and access path structure, such as B-

trees. This layer is also responsible for higher level operations, such as joins, selections, and sorting of record sets.

- *Data Services* present the data in logical structures like tables or views.
- *Extension Services* allow users to design tailored extensions to manage different data types, such as XML files or streaming data, or integrate their own application specific services.

As main components at a low-level, *functional services* provide the basic functions in a DBMS, such as storage services or query services. These services are managed by *coordinator services* that have the task to monitor the service activity and handle service reconfigurations as required. These services are handled by resource management processes which support information about service working states, process notifications, and manage service configurations. To ensure a high degree of interoperability between services, *adaptor services* mediate the interaction between services that have different interfaces and protocols. A predefined set of adapters can be provided to support standard communication protocol mediation or standard data types, while specialized adaptors can be automatically generated or manually created by the developer [17]. Service repositories handle service schemas and transformational schemas, while service registries enable service discovery.

3.2 Architectural Connectors

Connectors have the role to define the type of communication that takes place between software components [8]. Services present their purpose and capabilities through a *service contract* that is comprised of one or more service documents that describe the service [10]. To ensure increased interoperability, services are described through a *service description* document that provides descriptive information, such as used data types and semantic description of services and interfaces. A *service policy* includes service conditions of interaction, dependencies, and assertions that have to be fulfilled before a service is invoked. A *service quality* description enables service coordinators to take actions based on functional service properties. To ensure a high degree of interoperability, service contract documents should be described using open formats, such as WSDL or WS Policy. *Service communication* is done through well-defined communication protocols, such as SOAP or RMI. Communication protocols can be defined according to user requirements and the type of data exchanged between services. An important requirement is to use open protocols, rather than implementation specific technology. This allows one to achieve a high degree of abstraction and reduces implementation details in service contracts, which can reduce service interoperability.

3.3 Architectural Configurations

Configurations of the SBDMS depend on the specific environment requirements and on the available services in the system. To be adaptable, the system must be aware of the environment in which it is running and the available resources. Services are composed dynamically at run time according to architectural changes and user requirements.

From a general view we can envision two service phases: the setup phase and the operational phase. The *setup phase* consists of process composition according to architectural properties and service configuration. These properties specify the installed services, available resources, and service specific settings. In the *operational phase* coordinator services monitor architectural changes and service properties. If a change occurs resource management services find alternate workflows to manage the new situation. If a suitable workflow is found, adaptor services are created around the component services of the workflows to provide the original functionality based on alternative services. The architecture then undergoes a configuration and composition process to set the new communication paths, and finally compose newly created services. This is made possible as services are designed for late binding, which allows a high degree of flexibility and architecture reconfigurability.

3.4 Architectural Flexibility by Extension

Instead of hard-wiring static components we break down the DBMS architecture into services, obtaining a loosely coupled architecture that can be distributed. Such a service-based architecture can be complemented with services that are used by applications and other services allowing direct integration and development of additional extensions. In general, services are dynamically composed to accomplish complex tasks for a particular client. They are reusable and highly autonomous since they are accessed only through well-defined interfaces and clearly specified communication protocols. This standardisation helps to reduce complexity, because new components can be built with existing services. Organising services on layers is a solution to manage and compose large numbers of services. Developers can then deploy or update new services by stopping the affected processes, instead of having to deal with the whole system, as in the case of CDBS. System extensibility benefits from the service oriented basis of the architecture, due to a high degree of interoperability and reusability. In this manner future development within our architectural framework implies only low maintenance and development costs.

3.5 Architectural Flexibility by Selection

By being able to support multiple workflows for the same task, our SBDMS architecture can choose and use them according to specific requirements. If a user wants some information from different storage services, the architecture can select the order in which the services are invoked based on available resources or other criteria. This can be realised in an automated way by allowing the architecture to choose required services automatically, either based on a service description or by the user who manually specifies different workflows. Using extra information provided by other service execution plans, the service coordinators can create task plans and supervise them, without taking into consideration an extensive set of variables, because services just provide functionality and do not disclose their internal structure.

3.6 Architectural Flexibility by Adaptation

Compared with flexibility by selection, flexibility by adaptation is harder to achieve. If a service is erroneous or missing, the solution is to find a substitute. If no other service is available to provide the same functionality through the same

interfaces, but if there are other components with different interfaces that can provide the original functionality, the architecture can adapt the service interfaces to meet the new requirements. This adaptation is done by reusing or generating adaptor services in the affected processes, to ensure that the service communication is done according to the service contracts. The main issue here is to make the architecture aware of missing or erroneous services. To achieve this we introduce architecture properties that can be set by users or by monitoring services when existing components are removed or are erroneous. SOA does not provide a general way to make the architecture aware and adaptable to changes in the current state of the system. Standardised solutions to this problems have been proposed by new architectures that have emerged on the foundations provided by SOA. One of these is the Service Component Architecture (SCA) [18]. SCA provides methods and concepts to create components and describe how they can work together. The interactions between components can be modeled as services, separating the implementation technology from the provided functionality. The most atomic structure of the SCA is the *component* (see Figure 3). Components can be combined in larger structures forming *composites* (see Figure 4). Both components and composites can be recursively contained.

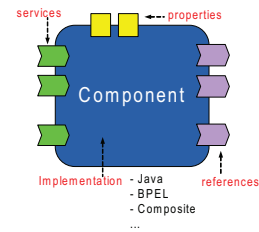


Figure 3: SCA Component [18]

Every component exposes functionality in form of one or more *services*, providing the number of *operations* that can be accessed. Components can rely on other services provided by other components. To describe this dependency, components use *references*. Beside services and references, a component can define one or more *properties*. Properties are read by the component when it is instantiated, allowing to customize its behaviour according to the current state of the architecture. By using services and references, a component can communicate with other software or components through *bindings*. A binding specifies exactly how communication should be done between the parties involved, defining the protocol and means of communication that can be used with the service or reference. Therefore a binding separates the communication from the functionality, making life simpler for developers and designers [5]. Furthermore, SCA or-

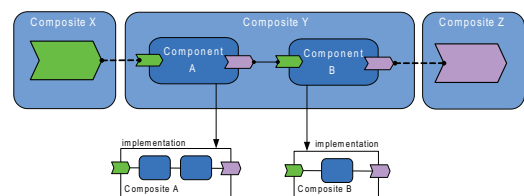


Figure 4: SCA Composites [18]

ganises the architecture in a hierarchically way, from coarse grained to fine grained components. This way of organizing the architecture makes it more manageable and comprehensible [12]. By using component properties, the adaptability character of the architecture can be easier achieved in a standardised way. For all these reasons, we include the principles of SCA into our SBDMS architecture.

3.7 A Storage Service Scenario

To exemplify our approach, we assume a simple storage service scenario and demonstrate how the three main aspects of flexibility from Section 2 can be realised using our proposed architecture. Figure 5 depicts the situation of adding a new

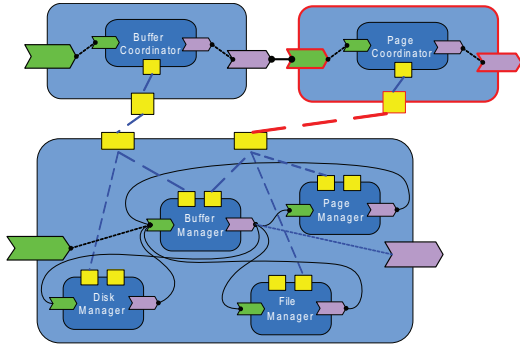


Figure 5: Flexibility by extension

service to the architecture. The user creates the required component (e.g., a Page Coordinator, as shown in Figure 5) and then publishes the desired interfaces as services in the architecture. From this point on, the desired functionality of the component is exposed and available for reuse. The service contract ensures that communication between services is done in a standardised way. In this way we abstract from implementation details and focus on the functionality provided by the system components. This allows ease of extensibility.

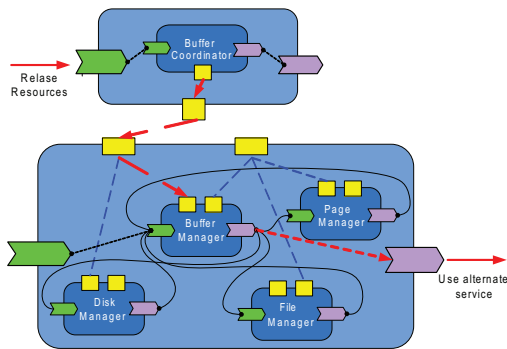


Figure 6: Flexibility by selection

At some point in time, special events may occur. Assume that some service S requires more resources. In this case, S invokes a “Release Resources” method on the coordinator services to free additional resources (see Figure 6). In our architecture component properties can then be set by users or coordinator services to adjust component properties according to the current architecture constraints. In this manner,

other services can be advised to stop using the service due to low resources.

Coordinator services also have the task to verify the availability of new services and other resources. In the example in Figure 6 a service requests more resources. The Buffer Coordinator advises the Buffer Manager to adapt to the new situation, by setting the appropriate service properties. In this case the Buffer Manager can use an alternate available workflow by using other available services that provide the same functionality. Every component behaves as defined by the alternate workflows which are managed by service coordinators. If services are erroneous or no longer available, and

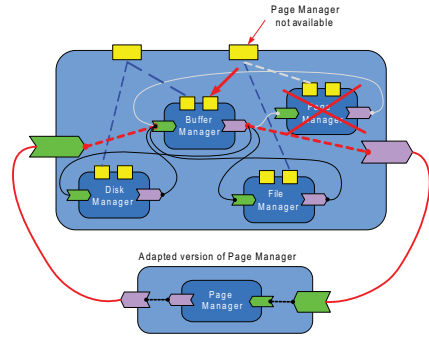


Figure 7: Flexibility by adaptation

other services can provide the same functionality, these can be used instead to complete the original tasks (see Figure 7). Even if performance may degrade to to increased work load, the system can continue to operate. If the service interfaces are compatible, coordinator services will create alternate processes that will compose the equivalent services to complete the requested task, in this way the architecture recomposes the services. Otherwise adaptor services have to be created to mediate service interaction.

4. DISCUSSION

To discuss and further illustrate our architectural concepts, we present two contrasting examples: a fully-fledged DBMS bundled with extensions and a small footprint DBMS capable of running in an embedded system environment.

Users with extensive functional requirements benefit from the ability of our architecture to integrate existing services that were developed by the user to satisfy his needs. Application developers can reuse services by integrating specialized services from any architectural layer into their application. For example developers may require additional information to monitor the state of a storage service (e.g., work load, buffer size, page size, and data fragmentation). Here, developers invoke existing coordinator services, or create customised monitoring services that read the properties from the storage service and retrieve data. In large scale architectures multiple services often provide the same functionality in a more or less specialised way. Since services are monitored by coordinators that supervise resource management and because service adaptors ensure correct communication between services, changes or errors in the system can be detected and alternate workflows and process compositions can be generated to handle the new situation.

If a storage service exhibits reduced performance that no longer meets the quality expected by the user, our architecture can use or adapt an alternative storage service to prevent system failures. Furthermore, storage services can be dynamically composed in a distributed environment, according to the current location of the client to reduce latency times. To enable service discovery, service repositories are required. For highly distributed and dynamic settings, P2P style service information updates can be used to transmit information between service repositories [9]. An open issue remains which service qualities are generally important in a DBMS and what methods or metrics should be used to quantify them.

In resource restricted environments, our architecture allows to disable unwanted services and to deploy small collections of services to mobile or embedded devices. The user can publish service functionality as web services to ensure a high degree of compatibility, or can use other communication protocols that suit his specific requirements. Devices can contain services that enable the architecture to monitor service activity and functional parameters. In case of a low resource alert, which can be caused by low battery capacity or high computation load, our SBDMS architecture can direct the workload to other devices to maintain the system operational. Disabling services requires that policies of currently running services are respected and all dependencies are met. To ensure this, service policies must be clearly described by service providers to ensure proper service functioning.

5. CONCLUSIONS

In this paper we proposed a novel architecture for DBMS that achieves flexibility and extensibility by adopting service-orientation. By taking a broad view on database architecture, we discussed aspects of flexibility that are relevant for extensible and flexible DBMS architectures. On this foundation, we designed our SBDMS framework to be generally applicable to provide tailored data management functionality and offer the possibility to directly integrate existing application functionality. Instead of taking a bottom up approach by extending an existing DBMS architecture, we use a top down approach and specialize our architecture to support the required “fitness for use” for specific application scenarios, ranging from fully-fledged DBMS with extensive functionality to small footprint DBMS in embedded systems. We exemplified how essential aspects of flexibility are met in our architecture and illustrated its applicability for tailor-made data management.

In future work we are going to design the proposed architecture in more detail and define a foundation for concrete implementations. We plan to take existing light weight databases, break them into services, and integrate them into our architecture for performance evaluations. Testing with different levels of service granularity will give us insights into the right tradeoff between service granularity and system performance in a SBDMS.

6. REFERENCES

- [1] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, 10 Dec 1990.
- [2] R. Awais. SADES - a Semi-Autonomous Database Evolution System. In *ECOOOP '98: Workshop on Object-Oriented Technology*, pages 24–25. Springer, 1998.
- [3] L. Bass and others. *Software Architecture in Practice*. AWLP, USA, 1998.
- [4] M. Carey et al. The EXODUS Extensible DBMS Project: An Overview. In *Readings in Object-Oriented Database Systems*, pages 474–499. MKP, 1990.
- [5] D. Chappel. Introducing SCA. Technical report, Chappell & Associates, 2007.
- [6] S. Chaudhuri and G. Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. *The VLDB Journal*, pages 1–10, 2000.
- [7] K. Dittrich and A. Geppert. *Component Database Systems*. Morgan Kaufmann Publishers, 2001.
- [8] S. Dustdar and H. Gall. Architectural Concerns in Distributed and Mobile Collaborative Systems. *Journal of Systems Architecture*, 49(10-11):457–473, 2003.
- [9] S. Dustdar and W. Schreiner. A Survey on Web Services Composition. *International Journal of Web and Grid Services*, 1(1):1–30, 2005.
- [10] T. Erl. *SOA Principles of Service Design*. PTR, 2007.
- [11] A. Geppert et al. KIDS: Construction of Database Management Systems Based on Reuse. Technical Report ifi-97.01, University of Zurich, 1997.
- [12] M. Glinz et al. The Adora Approach to Object-Oriented Modeling of Software. In *CAiSE 2001*, pages 76–92, 2001.
- [13] J. Gray. The Revolution in Database System Architecture. In *ADBIS (Local Proceedings)*, 2004.
- [14] T. Härder. DBMS Architecture – New challenges Ahead. *Datenbank-Spektrum (14)*, 14:38–48, 2005.
- [15] S. Hashimi. Service-Oriented Architecture Explained. Technical report, O’Reilly, 2003.
- [16] P. Heintz et al. A Comprehensive Approach to Flexibility in Workflow Management Systems. *WACC '99*, pages 79–88, 1999.
- [17] H. R. Motahari Nezhad et al. Semi-automated adaptation of service interactions. In *WWW '07*, pages 993–1002, 2007.
- [18] OASIS. *SCA Service Component Architecture, Specification*. 2007.
- [19] Oxford Online Dictionary. <http://www.askoxford.com>.
- [20] H. Schek et al. The DASDBS Project: Objectives, Experiences, and Future Prospects. *IEEE TKDE*, 2(1):25–43, 1990.
- [21] M. Stonebraker and U. Cetintemel. “One Size Fits All”: An Idea Whose Time has Come and Gone. In *ICDE '05*, pages 2–11, 2005.
- [22] M. Stonebraker et al. The Implementation of POSTGRES. *IEEE TKDE*, 2(1):125–142, 1990.
- [23] I. E. Subasu et al. Towards Service-Based Database Management Systems. In *BTW Workshops*, pages 296–306, 2007.

A New Approach to Modular Database Systems

Florian Irmert
Friedrich-Alexander University
of Erlangen-Nuremberg
Department of Computer
Science
Computer Science 6
(Data Management)
Martensstrasse 3
91058 Erlangen, Germany
florian.irmert@cs.fau.de

Michael Daum
Friedrich-Alexander University
of Erlangen-Nuremberg
Department of Computer
Science
Computer Science 6
(Data Management)
Martensstrasse 3
91058 Erlangen, Germany
michael.daum@cs.fau.de

Klaus Meyer-Wegener
Friedrich-Alexander University
of Erlangen-Nuremberg
Department of Computer
Science
Computer Science 6
(Data Management)
Martensstrasse 3
91058 Erlangen, Germany
kmw@cs.fau.de

ABSTRACT

In this paper we present our approach towards a modularized database management system (DBMS) whose components can be adapted at runtime and show the modularization of a DBMS beneath the record-oriented interface as a first step. Cross-cutting concerns like transactions pose thereby a challenge that we answer with aspect-oriented programming (AOP). Finally we show the implementation techniques that enable the exchange of database modules dynamically. Particularly with regard to stateful components we define a service adaptation process that preserves and transmits the component's state.

Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement; D.2.11 [Software Engineering]: Software Architectures; D.2.13 [Software Engineering]: Reusable Software; H.2.4 [Database Management]: Systems

General Terms

Design, Management

Keywords

Adaptation, availability, service-oriented architecture, component replacement, modularity, migration

1. INTRODUCTION

For about 30 years no major company is able to manage the large volume of information without a database management system (DBMS). Well known "generic" database systems like Oracle or DB2 extend their functionality with every release. Such commercial database systems are often developed over many years, and to stand out on the highly

competitive market, a lot of developers are needed to manage the extensive development. Although the "componentization" [18] of database management systems (DBMS) is a topic of research since the 90s, today's DBMSs still consist of a monolithic database kernel. Such a monolithic architecture increases maintenance and development costs additionally. In this paper we present our ideas towards a modular generic database. We describe the necessary properties and present work in progress. Current software engineering practices like service orientation and loose coupling are used to structure the design of the DBMS and therewith simplify the development and maintenance. As a result DBMSs for specific purpose can be developed much faster and cheaper compared to a traditional development process.

We aim to design a modular DBMS that can be adapted to different environments by assembling prefabricated modules. Therefore we need a kind of DBMS "construction kit". For each "building block" dependencies to other blocks have to be defined, and basic modules have to be identified that are necessary in every DBMS. Then a DBMS can be assembled by choosing the right modules for the specific task of the system.

To provide high availability, another challenge is the modification of modular DBMSs at runtime. We want to add, exchange, and remove modules while the database system is running. A possible scenario is the installation of a small DBMS with only a few features and its extension at runtime if new features are required. E.g. at the time of installation of an application a B-tree index is sufficient and therefore only the B-tree module is installed to save disk space. After a while a bitmap index would be very helpful. In a runtime adaptable DBMS the module for the bitmap index can be installed without stopping the DBMS. Towards this requirement a framework is needed to manage the individual modules.

2. RELATED WORK

The drawbacks of a monolithic DBMS are presented in [7]:

- DBMS become too complex. It is not possible to maintain them with reasonable costs.
- Applications have to pay performance and cost penalty

for using unneeded functionality.

- System evolution is more complex, because a DBMS vendor might not have the resources or expertise to perform such extensions in a reasonable period of time.

To solve such problems Dittrich and Geppert [7] propose componentization of DBMSs and identify four different categories of CDBMSs (component DBMS):

- **Plug-in Components.** The DBMS implements all standard functionality, and non-standard features can be plugged into the system. Examples for this category are Oracle Database [2], Informix [13] and IBM's DB2 [6].
- **Database Middleware.** DBMSs falling in this category integrate existing data stores, leaving data items under the control of their original management system, e.g. Garlic [16], Harmony [15] and OLE DB [4].
- **DBMS Services.** DBMSs of this type provide functionality in standardized form unbundled into services, e.g. CORBAServices [3].
- **Configurable DBMS.** DBMSs falling in this category are similar to DBMS services, but it is possible to adapt service implementations to new requirements or to define new services. An example is the KIDS project [8].

A good overview of these categories can be found in [20].

In recent time service-oriented architecture (SOA) has become a popular buzzword. The principles of SOA are also suitable for building modular DBMSs. [19] present an approach towards service-based DBMS. They borrowed the architectural levels from Härder [9] and want to include the advantages introduced by SOA like loosely coupled services. They have defined four different layers, which are implemented as services. It is argued that DBMS build upon this architecture is easily extendable because service can be invoked when they are needed and in case of failure of services, alternative services can answer the request. Tok and Bresnan [21] also introduce a DBMS architecture based on service orientation, called SODA (Service-Oriented Database Architecture). In their DBNet prototype web services are used as the basic building blocks for a query engine.

These approaches do not address the handling of cross cutting concerns (section 3.2) in a modular DBMS and in contrast to our approach do not provide runtime adaptation.

3. OUR APPROACH

The major goals of our CoBRA-DB project (Component Based Runtime Adaptable DataBase) are:

- Modularization of a relational database management system
- Exchanging modules at runtime

In this section we present work in progress. At first we introduce an educational DBMS called i6db, which is developed at our department. The i6db lays the foundations for our recent work towards a modular database system. Then we present our activities regarding the handling of cross cutting concerns inside a DBMS and finally we show our approach concerning runtime adaptation.

3.1 Modularization of database systems

Modularization and the definition of abstract interfaces are the first steps in order to get modules that can be exchanged at runtime. The challenge is to identify the appropriate modules. Härder [9] proposed a multi layer architecture of DBMSs that is very useful to understand the functionality of DBMS and to structure the important parts. The proposed layers are not fine grained enough to map them exactly to DBMS components; this would limit the dynamic adaptation (section 3.3), because the exchange of a whole layer's realization would lead to overhead if only small changes would be required.

In this section we present the architecture of i6db, a DBMS, which was designed and implemented at the department of computer science 6 (database systems) at the university of Erlangen-Nuremberg over the last years. The i6db is written in C++ and concentrates on layer abstraction, design patterns, and loose coupling.

i6db - a database for educational purposes

The i6db was originally designed for educational purposes, e.g. we set transactions and multi-user handling aside. Regarding to Härder's five level architecture (figure 1) [9] we have implemented the layer L1 to L4. The query engine can execute queries based on relational operators. Higher order query languages like SQL are taught on the basis of existing databases for educational purposes.

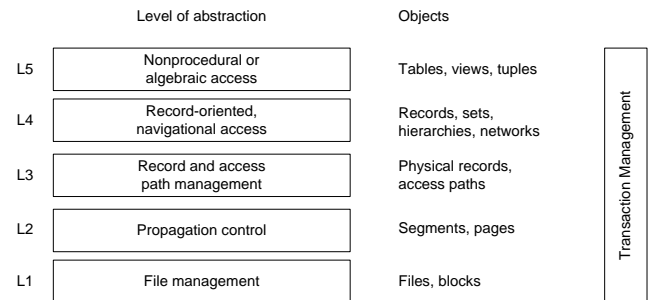


Figure 1: Five layer architecture

The core of i6db consists of seven modules. The module **file** is the L1 abstraction (file management) [9], **segment** and **systembuffer** are the L2 abstraction (propagation control). We assumed page/block-oriented data organization. All modules of L3 use accordingly the system buffer module. At the moment we have four different alternatives for the implementation of the record manager. There are two different algorithms (Tuple Identifier, DataBase key Translation Table) both with the extension of support of records that are larger than database pages. The exchange of those record manager's realizations is quite simple.

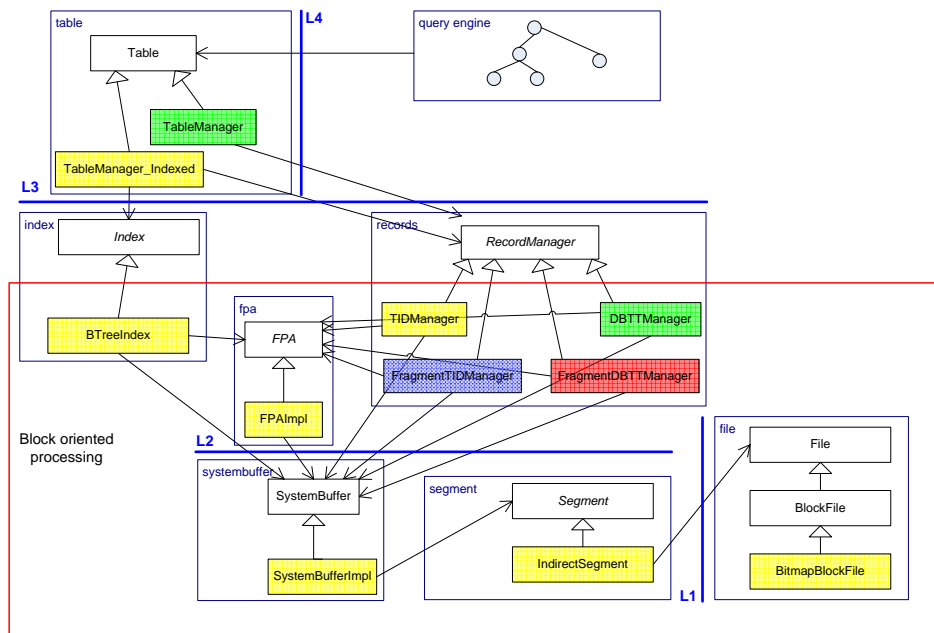


Figure 2: Modules of i6db

In figure 2 the large box enfames all realizations whose algorithms are based upon the use of pages. If i6db should be used as a main memory database, the block organization is obsolete. Then, all block based algorithms and structures must be abolished. As indexes and record manager don't need block orientation necessarily, the interfaces (Index, RecordManager) of the L3 (record and access path management) modules could be used further, but the modules must be exchanged. The *block-tree* index, the *free place administration* and all modules that organize records in blocks must be replaced by structures that are organized in main memory.

The L4 module (record-oriented, navigational access) `table` has two alternatives. The one holds a use-dependency to the `index`-module, the other doesn't. The `table`-module can be accessed by the methods that `table` provides. It depends solely on the table's implementation if indexes can be used. If an index-access method is called and there is no index available a full table scan has to be performed.

Our query engine can be used for the definition of tables and indexes. Records can be stored, removed and altered, too. Queries are defined by a query graph that consists of implemented relational operators. Due to the formal definition of relational operators each operator gets a set of input operators and predicates. Each operator can iterate the result. There are two table iterators implemented that access the tables of the `table`-module either by full-table-scan or by index-table-scan.

The i6db project lays a solid basis for creating a modular and adaptable database at least. In future we will integrate transaction mechanisms. As discussed below transactions are cross-cutting concerns. With the integration by using aspect-oriented programming techniques we can provide an interface that support transaction handling. Then we can

abolish the selfmade query engine and use MySQL 5.1 and integrate our i6db with its sound architecture as storage engine of MySQL [1].

3.2 Uncoupling Transactions

Existing approaches propose the incorporation of SOA to develop a modular DBMS [19, 21]. Modules like the query engine or layers [9] are realized as services. But cross cutting concerns like logging or transaction management (figure 1) make modularization in a full-fledged DBMS difficult, because e.g. to realize transactions, nearly every module is involved in the transaction management [14]. A "transaction object" is created at the beginning of a transaction and can not be destroyed until the transaction commits. Such an object could not be realized as a stateless service and all procedure calls which belong to the statements that are executed within a transaction must be associated with this object.

We are currently working on an extraction of the transactional aspect. The DBMS modules should not be aware of the fact that they are part of a transaction. For a prototype we have removed the implementation of transaction management in SimpleDB [17] and we are currently "re-integrating" the transaction support with the help of aspect-oriented programming (AOP) [12]. With AOP it is possible to intercept the methods of the modules and gain all necessary information to support transactions without the drawback of direct coupling. While we are presenting work in progress, there are still problems to be solved in our prototype, like the tracing of method invocations during the execution of a SQL statement.

3.3 Adaptation at runtime

Beside modularization the second major goal of the CoBRADB project is runtime adaptation. There are different sce-

narios where runtime adaptation is useful:

- Change a module that contains a bug
- Upgrade the DBMS with new functionality
- Remove unused features to increase performance and save space
- Change of a set of modules by cross cutting concerns or multiple changes of different modules as a result in order to change bigger parts or strategies of a DBMS

An example for a far-reaching upgrade would be adding transactions management to a running DBMS, which has not the need for transactions at time of its installation.

We pick up the idea of Cervantes and Hall, who have introduced a service-oriented component model: "A service-oriented component model introduces concepts from service orientation into a component model. The motivation for such a combination emerges from the need to introduce explicit support for dynamic availability into a component model" [5]. This concept can be used as basis for our work towards runtime adaptation. DBMS "modules" are realized as components that provide their functionality as services. Some of these modules are mandatory and provide the DBMS's core functionality like inserting and querying data. Other services are optional and can be added or removed depending on the application and the environment, e.g., if transaction support is not required, the "transaction module" is removed.

We do not propose the distribution of services. Remote procedure calls are much too slow to be used inside a DBMS. The required components should be able to invoke one another locally. To accomplish loosely binding a service searches in a global registry to locate services which are required to fulfill its role and than these service can be bounded and invoked.

One characteristic of service orientation is the fact that service can arrive and disappear at any point in time. Business applications which rely on specific service are not available if a mandatory service disappears and can not be replaced with another adequate service. This behavior is not acceptable if we are building a DBMS, because if one service of the DBMS disappears, all depending applications would not be able to work correctly. With this requirement in mind it is absolutely necessary for a DBMS which relies on service-oriented components that these components are available and their provided services are accessible. To swap services at runtime, the adaptation has to be done transparently for all consumers of that service. Obviously some time is needed to replace a running component with a new implementation. Therefore the method calls have to be interrupted and redirected to the new component. This "switch over act" has to be done in an atomic operation for all services which rely on the adapting service.

To handle the whole adaptation process we introduce an Adaptation Manager which coordinates the individual steps. The process is divided in 3 phases (figure 3). During the

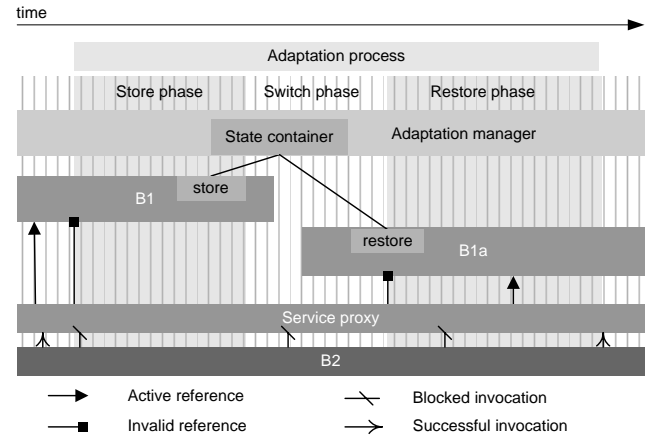


Figure 3: Service adaptation process

store phase the references of all depending service are invalidated and the state of the "old" component is saved in a special data structure. In the switch phase the state is injected in the new component. Last the references are set to the new component in the restore phase, and the references are set to the new component. We describe the adaptation process in [10] in detail.

To enable the adaptation of aspects at runtime we integrate a dynamic AOP (d-AOP) framework in our prototype because d-AOP supports the modification of aspects at runtime. Therefore we can use the techniques we have presented in [11] to integrate dynamic AOP into a service-oriented component model.

4. FUTURE WORK AND CONCLUSION

In the paper we introduced the CoBRA-DB project. The goal of this project is a modularized runtime adaptable DBMS. We argued the problems of "slicing" a database into loosely coupled components and the challenge regarding cross cutting concerns. We are currently implementing a prototype framework to adapt components at runtime. Thereby the state of a component is transferred to the replacing component in an atomic step. With the lessons learned in the i6db project, where we have implemented a DBMS in C++, we are now going to develop a prototype in Java. In parallel we remove the transaction management from a sample DBMS (we use SimpleDB) and reintegrate it with the help of AOP to provide a foundation for further modularization. This is a major difference in contrast to other projects which use SOA to modularize DBMSs. Another distinction is the ability to swap modules at runtime and thereby adapt a DBMS to a changing environment without the need to shutdown the database.

5. REFERENCES

- [1] D. Axmark, M. Widenius, P. DuBois, S. Hinz, M. Hillyer, and J. Stephens. *MySQL 5.1 Referenzhandbuch*. MySQL, 2007.
- [2] S. Banerjee, V. Krishnamurthy, and R. Murthy. All your data: the oracle extensibility architecture. *Component database systems*, pages 71–104, 2001.
- [3] R. Bastide and O. Sy. Formal specification of CORBA

- services: experience and lessons learned. *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 105–117, 2000.
- [4] J. A. Blakeley. Data access for the masses through ole db. *SIGMOD Rec.*, 25(2):161–172, 1996.
- [5] H. Cervantes and R. S. Hall. Autonomous adaptation to dynamic availability using a service-oriented component model. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 614–623, Washington, DC, USA, 2004. IEEE Computer Society.
- [6] J. Cheng, J.; Xu. Xml and db2. *Data Engineering, 2000. Proceedings. 16th International Conference on*, pages 569–573, 2000.
- [7] K. R. Dittrich and A. Geppert, editors. *Component database systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.
- [8] A. Geppert, S. Scherrer, and K. R. Dittrich. KIDS: Construction of Database Management Systems based on Reuse. Technical Report ifi-97.01, University of Zurich, 1997.
- [9] T. Härder. DBMS Architecture - the Layer Model and its Evolution (Part I). *Datenbank-Spektrum*, 5(13):45–56, 2005.
- [10] F. Irmert, T. Fischer, and K. Meyer-Wegener. Improving availability in a service-oriented component model using runtime adaptation. University of Erlangen and Nuremberg, to be published, 2007.
- [11] F. Irmert, M. Meyerhöfer, and M. Weiten. Towards Runtime Adaptation in a SOA Environment. RAM-SE'07 - 4th ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution, co-located at the 21th European Conference on Object-Oriented Programming - ECOOP (Berlin, Germany), July 2007.
- [12] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.
- [13] M. Olson. DataBlade extensions for INFORMIX-Universal Server. *Proceedings IEEE COMPCON*, 97:143–8, 1997.
- [14] A. Rashid. *Aspect-Oriented Database Systems*. Springer, 2004.
- [15] U. Röhm and K. Böhm. Working Together in Harmony - An Implementation of the CORBA Object Query Service and Its Evaluation. In *ICDE'99: Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Australia*, pages 238–247, 1999.
- [16] M. T. Roth and P. M. Schwarz. Don't scrap it, wrap it! a wrapper architecture for legacy data sources. In M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 266–275. Morgan Kaufmann, 1997.
- [17] E. Sciore. SimpleDB: a simple java-based multiuser system for teaching database internals. *ACM SIGCSE Bulletin*, 39(1):561–565, 2007.
- [18] A. Silberschatz and S. Zdonik. Strategic directions in database systems - breaking out of the box. *ACM Comput. Surv.*, 28(4):764–778, 1996.
- [19] I. E. Subasu, P. Ziegler, and K. R. Dittrich. Towards service-based database management systems. In *Datenbanksysteme in Business, Technologie und Web (BTW 2007), Workshop Proceedings, 5.-6. März 2007, Aachen, Germany*, pages 296–306, 2007.
- [20] A. Tesanovic, D. Nystrom, J. Hansson, and C. Norstrom. Embedded databases for embedded real-time systems: A component-based approach. Technical report, Dept. of Computer Science, Linköping University, and Dept. of Computer Engineering, Malardalen University, 2002.
- [21] W. H. Tok and S. Bressan. DBNet: A Service-Oriented Database Architecture. In *DEXA '06: Proceedings of the 17th International Conference on Database and Expert Systems Applications*, pages 727–731, Washington, DC, USA, 2006. IEEE Computer Society.