Software Design Approaches for Mastering Variability in Database Systems

David Broneske, Sebastian Dorok, Veit Köppen, Andreas Meister*
*author names are in lexicographical order
Otto-von-Guericke-University Magdeburg
Institute for Technical and Business Information Systems
Magdeburg, Germany
firstname.lastname@ovgu.de

ABSTRACT

For decades, database vendors have developed traditional database systems for different application domains with highly differing requirements. These systems are extended with additional functionalities to make them applicable for yet another data-driven domain. The database community observed that these "one size fits all" systems provide poor performance for special domains; systems that are tailored for a single domain usually perform better, have smaller memory footprint, and less energy consumption. These advantages do not only originate from different requirements, but also from differences within individual domains, such as using a certain storage device.

However, implementing specialized systems means to re-implement large parts of a database system again and again, which is neither feasible for many customers nor efficient in terms of costs and time. To overcome these limitations, we envision applying techniques known from software product lines to database systems in order to provide tailor-made and robust database systems for nearly every application scenario with reasonable effort in cost and time.

General Terms

Database, Software Engineering

Keywords

Variability, Database System, Software Product Line

1. INTRODUCTION

In recent years, data management has become increasingly important in a variety of application domains, such as automotive engineering, life sciences, and web analytics. Every application domain has its unique, different functional and non-functional requirements leading to a great diversity of *database systems* (DBSs). For example, automotive data management requires DBSs with small storage and memory consumption to deploy them on embedded devices. In contrast, big-data applications, such as in life sciences, require large-scale DBSs, which exploit newest hardware trends, e.g., vectorization and SSD storage, to efficiently process and manage petabytes of data [8]. Exploiting variability to design a tailor-made DBS for applications while making the variability manageable, that is keeping maintenance effort, time, and cost reasonable, is what we call *mastering* variability in DBSs.

Currently, DBSs are designed either as one-size-fits-all DBSs, meaning that all possible use cases or functionalities are integrated at implementation time into a single DBS, or as specialized solutions. The first approach does not scale down, for instance, to embedded devices. The second approach leads to situations, where for each new application scenario data management is reinvented to overcome resource restrictions, new requirements, and rapidly changing hardware. This usually leads to an increased time to market, high development cost, as well as high maintenance cost. Moreover, both approaches provide limited capabilities for managing variability in DBSs. For that reason, *software product line* (SPL) techniques could be applied to the data management domain. In SPLs, variants are concrete programs that satisfy the requirements of a specific application domain [7]. With this, we are able to provide tailor-made and robust DBSs for various use cases. Initial results in the context of embedded systems, expose benefits of applying SPLs to DBSs [17, 22].

The remainder of this paper is structured as follows: In Section 2, we describe variability in a database system regarding hardware and software. We review three approaches to design DBSs in Section 3, namely, the one-size-fits-all, the specialization, and the SPL approach. Moreover, we compare these approaches w.r.t. robustness and maturity of provided DBSs, the effort of managing variability, and the level of tailoring for specific application domains. Because of the superiority of the SPL approach, we argue to apply this approach to the implementation process of a DBS. Hence, we provide research questions in Section 4 that have to be answered to realize the vision of mastering variability in DBSs using SPL techniques.

2. VARIABILITY IN DATABASE SYSTEMS

Variability in a DBS can be found in software as well as hardware. Hardware variability is given due to the use of different devices with specific properties for data processing and storage. Variability in software is reflected by different functionalities that have to be provided by the DBS for a specific application. Additionally, the combination of

In: G. Specht, H. Gamper, F. Klan (eds.): Proceedings of the 26th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 21.10.2014 - 24.10.2014, Bozen, Italy, published at http://ceur-ws.org.

hardware and software functionality for concrete application domains increases variability.

2.1 Hardware

In the past decade, the research community exploited arising hardware features by tailor-made algorithms to achieve optimized performance. These algorithms effectively utilize, e.g., caches [19] or vector registers of *Central Processing Units* (CPUs) using AVX- [27] and SSE-instructions [28]. Furthermore, the usage of co-processors for accelerating data processing opens up another dimension [12]. In the following, we consider processing and storage devices and sketch the variability arising from their different properties.

2.1.1 Processing Devices

To sketch the heterogeneity of current systems, possible (co-)processors are summarized in Figure 1. Current systems do not only include a CPU or an *Accelerated Processing Unit* (APU), but also co-processors, such as *Many Integrated Cores* (MICs), *Graphical Processing Units* (GPUs), and *Field Programmable Gate Arrays* (FPGAs). In the following, we give a short description of varying processor properties. A more extensive overview is presented in our recent work [5].

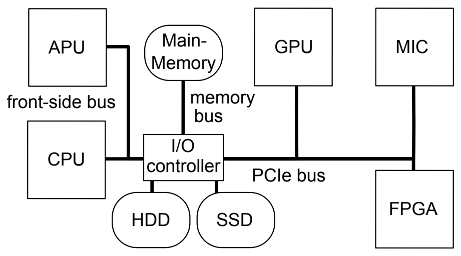


Figure 1: Future system architecture [23]

Central Processing Unit: Nowadays, CPUs consist of several independent cores, enabling parallel execution of different calculations. CPUs use pipelining, *Single Instruction Multiple Data* (SIMD) capabilities, and branch prediction to efficiently process conditional statements (e.g., if statements). Hence, CPUs are well suited for control intensive algorithms.

Graphical Processing Unit: Providing larger SIMD registers and a higher number of cores than CPUs, GPUs offer a higher degree of parallelism compared to CPUs. In order to perform calculations, data has to be transferred from main memory to GPU memory. GPUs offer an own memory hierarchy with different memory types.

Accelerated Processing Unit: APUs are introduced to combine the advantages of CPUs and GPUs by including both on one chip. Since the APU can directly access main memory, the transfer bottleneck of dedicated GPUs is eliminated. However, due to space limitations, fairly less GPU cores fit on the APU die compared to a dedicated GPU, leading to reduced computational power compared to dedicated GPUs.

Many Integrated Core: MICs use several integrated and interconnected CPU cores. With this, MICs offer a high parallelism while still featuring CPU properties. However, similar to the GPU, MICs suffer from the transfer bottleneck.

Field Programmable Gate Array: FPGAs are programmable stream processors, providing only a limited storage capacity. They consist of several independent logic cells consisting of a storage unit and a lookup table. The interconnect between logic cells and the lookup tables can be reprogrammed during run time to perform any possible function (e.g., sorting, selection).

2.1.2 Storage Devices

Similar to the processing device, current systems offer a variety of different storage devices used for data processing. In this section, we discuss different properties of current storage devices.

Hard Disk Drive: The *Hard Disk Drive* (HDD), as a non-volatile storage device, consists of several disks. The disks of an HDD rotate, while a movable head reads or writes information. Hence, sequential access patterns are well supported in contrast to random accesses.

Solid State Drive: Since no mechanical units are used, *Solid State Drives* (SSDs) support random access without high delay. For this, SSDs use flash-memory to persistently store information [20]. Each write wears out the flash cells. Consequently, the write patterns of database systems must be changed compared to HDD-based systems.

Main-Memory: While using main memory as main storage, the access gap between primary and secondary storage is removed, introducing main-memory access as the new bottleneck [19]. However, main-memory systems cannot omit secondary storage types completely, because main memory is volatile. Thus, efficient persistence mechanisms are needed for main-memory systems.

To conclude, current architectures offer several different processor and storage types. Each type has a unique architecture and specific characteristics. Hence, to ensure high performance, the processing characteristics of processors as well as the access characteristics of the underlying storage devices have to be considered. For example, if several processing devices are available within a DBS, the DBS must provide suitable algorithms and functionality to fully utilize all available devices to provide peak performance.

2.2 Software Functionality

Besides hardware, DBS functionality is another source of variability in a DBS. In Figure 2, we show an excerpt of DBS functionalities and their dependencies. For example, for different application domains different query types might be interesting. However, to improve performance or development cost, only required query types should be used within a system. This example can be extended to other functional requirements. Furthermore, a DBS provides database operators, such as aggregation functions or joins. Thereby, database operators perform differently depending on the used storage and processing model [1]. For example, row stores are very efficient when complete tuples should be retrieved, while column stores in combination with operator-at-a-time processing enable fast processing of single columns [18]. Another technique to enable efficient access to data is to use index structures. Thereby, the choice of an appropriate index structure for the specific data and query types is essential to guarantee best performance [15, 24].

Note, we omit comprehensive relationships between functionalities properties in Figure 2 due to complexity. Some

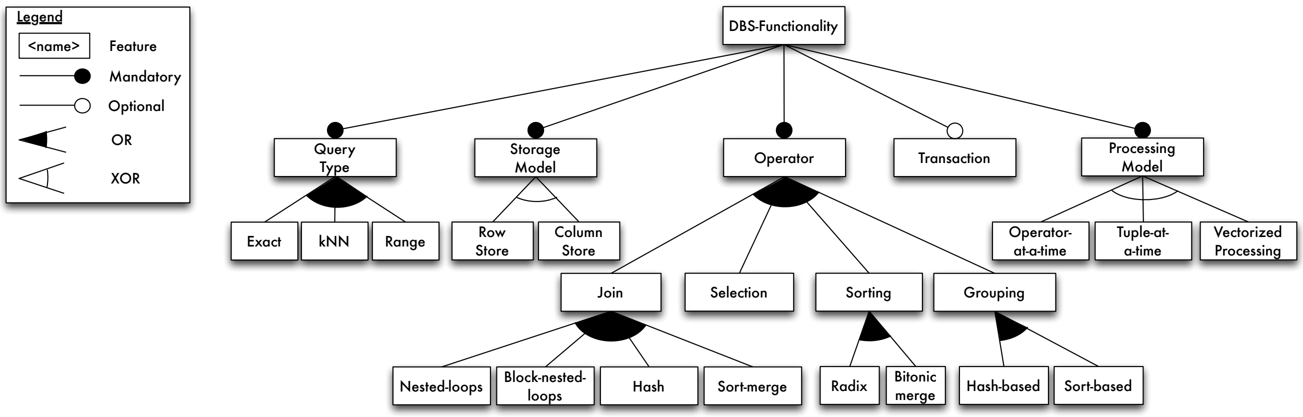


Figure 2: Excerpt of DBMS-Functionality

functionalities are mandatory in a DBS and others are optional, such as support for transactions. Furthermore, it is possible that some alternatives can be implemented together and others only exclusively.

2.3 Putting it all together

So far, we considered variability in hardware and software functionality separately. When using a DBS for a specific application domain, we also have to consider special requirements of this domain as well as the interaction between hardware and software.

Special requirements comprise functional as well as non-functional ones. Examples for functional requirements are user-defined aggregation functions (e.g., to perform genome analysis tasks directly in a DBS [9]). Other applications require support for spatial queries, such as geo-information systems. Thus, special data types as well as index structures are required to support these queries efficiently.

Besides performance, memory footprint and energy efficiency are other examples for non-functional requirements. For example, a DBS for embedded devices must have a small memory footprint due to resource restrictions. For that reason, unnecessary functionality is removed and data processing is implemented as memory efficient as possible. In this scenario, tuple-at-a-time processing is preferred, because intermediate results during data processing are smaller than in operator-at-a-time processing, which leads to less memory consumption [29].

In contrast, in large-scale data processing, operators should perform as fast as possible by exploiting underlying hardware and available indexes. Thereby, exploiting underlying hardware is another source of variability as different processing devices have different characteristics regarding processing model and data access [6]. To illustrate this fact, we depict different storage models for DBS in Figure 2. For example, column-storage is preferred on GPUs, because row-storage leads to an inefficient memory access pattern that deteriorates the possible performance benefits of GPUs [13].

3. APPROACHES TO DESIGN TAILOR-MADE DATABASE SYSTEMS

The variability in hardware and software of DBSs can be exploited to tailor database systems for nearly every database-application scenario. For example, a DBS for high-performance analysis can exploit newest hardware features, such as SIMD, to speed up analysis workloads. Moreover, we can meet limited space requirements in embedded systems by removing unnecessary functionality [22], such as the support for range queries. However, exploiting variability is one part of mastering variability in DBSs. The second part is to manage variability efficiently to reduce development and maintenance effort.

In this section, first, we describe three different approaches to design and implement DBSs. Then, we compare these approaches regarding their applicability to arbitrary database scenarios. Moreover, we assess the effort to manage variability in DBSs. Besides managing and exploiting the variability in database systems, we also consider the robustness and correctness of tailor-made DBSs created by using the discussed approaches.

3.1 One-Size-Fits-All Design Approach

One way to design database systems is to integrate all conceivable data management functionality into one single DBS. We call this approach the *one-size-fits-all* design approach and DBSs designed according to this approach *one-size-fits-all* DBSs. Thereby, support for hardware features as well as DBMS functionality are integrated into one code base. Thus, one-size-fits-all DBSs provide a rich set of functionality. Examples of database systems that follow the one-size-fits-all approach are PostgreSQL, Oracle, and IBM DB2. As one-size-fits-all DBSs are monolithic software systems, implemented functionality is highly interconnected on the code level. Thus, removing functionality is mostly not possible.

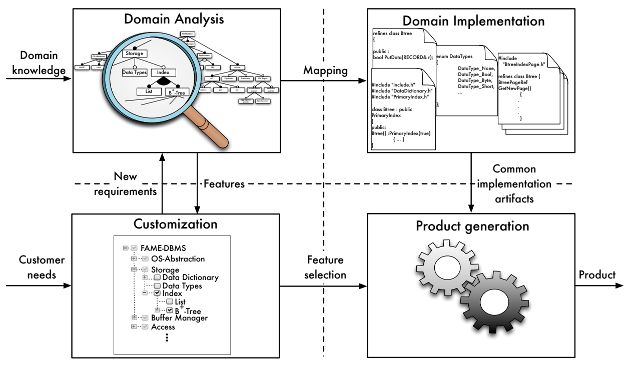
DBSs that follow the one-size-fits-all design approach aim at providing a comprehensive set of DBS functionality to deal with most database application scenarios. The claim for generality often introduces functional overhead that leads to performance losses. Moreover, customers pay for functionality they do not really need.

3.2 Specialization Design Approach

In contrast to one-size-fits-all DBSs, DBSs can also be designed and developed to fit very specific use cases. We call this design approach the *specialization design approach* and DBSs designed accordingly, *specialized* DBSs. Such DBSs are designed to provide only that functionality that is needed for the respective use case, such as text processing, data warehousing, or scientific database applications [25]. Specialized DBSs are often completely redesigned from scratch to meet application requirements and do not follow common design considerations for database systems, such as locking and latching to guarantee multi-user access [25]. Specialized DBSs remove the overhead of unneeded functionality. Thus, developers can highly focus on exploiting hardware and functional variability to provide tailor-made DBSs that meet high-performance criteria or limited storage space requirements. Therefore, huge parts of the DBS (if not all) must be newly developed, implemented, and tested which leads to duplicate implementation efforts, and thus, increased development costs.

3.3 Software Product Line Design Approach

In the specialization design approach, a new DBS must be developed and implemented from scratch for every conceivable database application. To avoid this overhead, the *SPL design approach* reuses already implemented and tested parts of a DBS to create a tailor-made DBS.



```
refines class Btree
{

public :
bool PutData(RECORD& r);

#include "include.h"
#include "DataDictionary.h"
#include "PrimaryIndex.h"

class Btree : public
PrimaryIndex
{
public:
Btree() :PrimaryIndex(true)
        { ... }
}
```

```
enum DataTypes
{
        DataType_None,
        DataType_Bool,
        DataType_Byte,
        DataType_Short,
        ...
};
```

```
#include
  "BtreeIndexPage.h"

refines class Btree {
BtreePageRef
GetNewPage()
        {
        .
        .
        .
        }
```

Figure 3: Managing Variability

To make use of SPL techniques, a special workflow has to be followed which is sketched in Figure 3 [2]. At first, the domain is modeled, e.g., by using a feature model – a tree-like structure representing features and their dependencies. With this, the variability is captured and implementation artifacts can be derived for each feature. The second step, the domain implementation, is to implement each feature using a compositional or annotative approach. The third step of the workflow is to customize the product – in our case, the database system – which will be generated afterwards.

By using the SPL design approach, we are able to implement a database system from a set of features which are mostly already provided. In best case, only non-existing features must be implemented. Thus, the feature pool constantly grows and features can be reused in other database systems. Applying this design approach to DBSs enables to create DBSs tailored for specific use cases while reducing functional overhead as well as development time. Thus, the *SPL design approach* aims at the middle ground of the one-size-fits-all and the specialization design approach.

3.4 Characterization of Design Approaches

In this section, we characterize the three design approaches discussed above regarding:

a) general *applicability* to arbitrary database applications,
b) effort for *managing* variability, and
c) *maturity* of the deployed database system.

Although the one-size-fits-all design approach aims at providing a comprehensive set of DBS functionality to deal with most database application scenarios, a one-size-fits-all database is not applicable to use cases in automotive, embedded, and ubiquitous computing. As soon as tailor-made software is required to meet especially storage limitations, one-size-fits-all database systems cannot be used. Moreover, specialized database systems for one specific use case outperform one-size-fits-all database systems by orders of magnitude [25]. Thus, although one-size-fits-all database systems can be applied, they are often not the best choice regarding performance. For that reason, we consider the applicability of one-size-fits-all database systems to arbitrary use cases as limited. In contrast, specialized database systems have a very good applicability as they are designed for that purpose.

The applicability of the SPL design approach is good as it also creates database systems tailor-made for specific use cases. Moreover, the SPL design approach explicitly considers variability during software design and implementation and provides methods and techniques to manage it [2]. For that reason, we assess the effort of managing variability with the SPL design approach as lower than managing variability using a one-size-fits-all or specialized design approach.

We assess the maturity of one-size-fits-all database systems as very good, as these systems are developed and tested over decades. Specialized database systems are mostly implemented from scratch, so, the possibility of errors in the code is rather high, leading to a moderate maturity and robustness of the software. The SPL design approach also enables the creation of tailor-made database systems, but from approved features that are already implemented and tested. Thus, we assess the maturity of database systems created via the SPL design approach as good.

In Table 1, we summarize our assessment of the three software design approaches regarding the above criteria.

Criteria	Approach		
	One-Size-Fits-All	Specialization	SPL
a) Applicability	−	++	+
b) Management effort	−	−	+
c) Maturity	++	○	+

Table 1: Characteristics of approaches
Legend: ++ = very good, + = good, ○ = moderate, − = limited

The one-size-fits-all and the specialization design approach are each very good in one of the three categories respectively. The one-size-fits-all design approach provides robust and mature DBSs. The specialization design approach provides greatest applicability and can be used for nearly every use case. Whereas the SPL design approach provides a balanced assessment regarding all criteria. Thus, against the backdrop of increasing variability due to increasing variety of use cases and hardware while guaranteeing mature and robust DBSs, SPL design approach should be applied to develop future DBSs. Otherwise, development costs for yet another DBS which has to meet special requirements of the next data-driven domain will limit the use of DBSs in such fields.

4. ARISING RESEARCH QUESTIONS

Our assessment in the previous section shows that the SPL design approach is the best choice for mastering variability in DBSs. To the best of our knowledge, the SPL design approach is applied to DBSs only in academic settings (e.g., in [22]).Hereby, the previous research were based on BerkeleyDB. Although BerkeleyDB offers the essential functionality of DBSs (e.g., a processing engine), several functionality of relational DBSs were missing (e.g., optimizer, SQL-interface). Although these missing functionality were partially researched (e.g., storage manager [16] and the SQL parser [26]), no holistic evaluation of a DBS SPL is available. Especially, the optimizer in a DBS (e.g., query optimizer) with a huge number of crosscutting concerns is currently not considered in research. So, there is still the need for research to fully apply SPL techniques to all parts of a DBS. Specifically, we need methods for modeling variability in DBSs and efficient implementation techniques and methods for implementing variability-aware database operations.

4.1 Modeling

For modeling variability in feature-oriented SPLs, feature models are state of the art [4]. A feature model is a set of features whose dependencies are hierarchically modeled. Since variability in DBSs comprises hardware, software, and their interaction, the following research questions arise:

RQ-M1: What is a good granularity for modeling a variable DBS?
In order to define an SPL for DBSs, we have to model features of a DBS. Such features can be modeled with different levels of granularity [14]. Thus, we have to find an applicable level of granularity for modeling our SPL for DBSs. Moreover, we also have to consider the dependencies between hardware and software. Furthermore, we have to find a way to model the hardware and these dependencies. In this context, another research questions emerges:

RQ-M2: What is the best way to model hardware and its properties in an SPL?
Hardware has become very complex and researchers demand to develop a better understanding of the impact of hardware on the algorithm performance, especially when parallelized [3, 5]. Thus, the question arises what properties of the hardware are worth to be captured in a feature model.

Furthermore, when thinking about numerical properties, such as CPU frequency or amount of memory, we have to find a suitable technique to represent them in feature models. One possibility are *attributes* of extended feature-models [4], which have to be explored for applicability.

4.2 Implementing

In the literature, there are several methods for implementing an SPL. However, most of them are not applicable to our use case. Databases rely on highly tuned operations to achieve peak performance. Thus, variability-enabled implementation techniques must not harm the performance, which leads to the research question:

RQ-I1: What is a good variability-aware implementation technique for an SPL of DBSs?
Many state of the art implementation techniques are based on inheritance or additional function calls, which causes performance penalties. A technique that allows for variability without performance penalties are preprocessor directives. However, maintaining preprocessor-based SPLs is horrible, which accounts this approach the name *#ifdef Hell* [11, 10]. So, there is a trade-off between performance and maintainability [22], but also granularity [14]. It could be beneficial for some parts of DBS to prioritize maintainability and for others performance or maintainability.

RQ-I2: How to combine different implementation techniques for SPLs?
If the answer of RQ-I1 is to use different implementation techniques within the same SPL, we have to find an approach to combine these. For example, database operators and their different hardware optimization must be implemented using annotative approaches for performance reasons, but the query optimizer can be implemented using compositional approaches supporting maintainability; the SPL product generator has to be aware of these different implementation techniques and their interactions.

RQ-I3: How to deal with functionality extensions?
Thinking about changing requirements during the usage of the DBS, we should be able to extend the functionality in the case user requirements change. Therefore, we have to find a solution to deploy updates from an extended SPL in order to integrate the new requested functionality into a running DBS. Some ideas are presented in [21], however, due to the increased complexity of hardware and software requirements an adaption or extension is necessary.

4.3 Customization

In the final customization, features of the product line are selected that apply to the current use case. State of the art approaches just list available features and show which features are still available for further configuration. However, in our scenario, it could be helpful to get further information of the configuration possibilities. Thus, another research question is:

RQ-C1: How to support the user to obtain the best selection?
In fact, it is possible to help the user in identifying suitable configurations for his use case. If he starts to select functionality that has to be provided by the generated system, we can give him advice which hardware yields the best performance for his algorithms. However, to achieve this we have to investigate another research question:

RQ-C2: How to find the optimal algorithms for a given hardware?
To answer this research question, we have to investigate the relation between algorithmic design and the impact of the hardware on the execution. Hence, suitable properties of algorithms have to be identified that influence performance on the given hardware, e.g., access pattern, size of used data structures, or result sizes.

5. CONCLUSIONS

DBSs are used for more and more use cases. However, with an increasing diversity of use cases and increasing heterogeneity of available hardware, it is getting more challenging to design an optimal DBS while guaranteeing low implementation and maintenance effort at the same time. To solve this issue, we review three design approaches, namely the one-size-fits-all, the specialization, and the software product line design approach. By comparing these three design approaches, we conclude that the SPL design approach is a promising way to master variability in DBSs and to provide mature data management solutions with reduced implementation and maintenance effort. However, there is currently no comprehensive software product line in the field of DBSs available. Thus, we present several research questions that have to be answered to fully apply the SPL design approach on DBSs.

6. ACKNOWLEDGMENTS

This work has been partly funded by the German BMBF under Contract No. 13N10818 and Bayer Pharma AG.

7. REFERENCES

[1] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. Row-stores: How Different Are They Really? In *SIGMOD*, pages 967–980. ACM, 2008.

[2] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines*. Springer, 2013.

[3] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *PVLDB*, 7(1):85–96, 2013.

[4] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Inf. Sys.*, 35(6):615–636, 2010.

[5] D. Broneske, S. Breß, M. Heimel, and G. Saake. Toward Hardware-Sensitive Database Operations. In *EDBT*, pages 229–234, 2014.

[6] D. Broneske, S. Breß, and G. Saake. Database Scan Variants on Modern CPUs: A Performance Study. In *IMDM@VLDB*, 2014.

[7] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* ACM Press/Addison-Wesley Publishing Co., 2000.

[8] S. Dorok, S. Breß, H. Läpple, and G. Saake. Toward Efficient and Reliable Genome Analysis Using Main-Memory Database Systems. In *SSDBM*, pages 34:1–34:4. ACM, 2014.

[9] S. Dorok, S. Breß, and G. Saake. Toward Efficient Variant Calling Inside Main-Memory Database Systems. In *BIOKDD-DEXA*. IEEE, 2014.

[10] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake. Do Background Colors Improve Program Comprehension in the #ifdef Hell? *Empir. Softw. Eng.*, 18(4):699–745, 2013.

[11] J. Feigenspan, M. Schulze, M. Papendieck, C. Kästner, R. Dachselt, V. Köppen, M. Frisch, and G. Saake. Supporting Program Comprehension in Large Preprocessor-Based Software Product Lines. *IET Softw.*, 6(6):488–501, 2012.

[12] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational Query Coprocessing on Graphics Processors. *TODS*, 34(4):21:1–21:39, 2009.

[13] B. He and J. X. Yu. High-throughput Transaction Executions on Graphics Processors. *PVLDB*, 4(5):314–325, Feb. 2011.

[14] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *ICSE*, pages 311–320. ACM, 2008.

[15] V. Köppen, M. Schäler, and R. Schröter. Toward Variability Management to Tailor High Dimensional Index Implementations. In *RCIS*, pages 452–457. IEEE, 2014.

[16] T. Leich, S. Apel, and G. Saake. Using Step-wise Refinement to Build a Flexible Lightweight Storage Manager. In *ADBIS*, pages 324–337. Springer-Verlag, 2005.

[17] J. Liebig, S. Apel, C. Lengauer, and T. Leich. RobbyDBMS: A Case Study on Hardware/Software Product Line Engineering. In *FOSD*, pages 63–68. ACM, 2009.

[18] A. Lübcke, V. Köppen, and G. Saake. Heuristics-based Workload Analysis for Relational DBMSs. In *UNISCON*, pages 25–36. Springer, 2012.

[19] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing Database Architecture for the New Bottleneck: Memory Access. *VLDB J.*, 9(3):231–246, 2000.

[20] R. Micheloni, A. Marelli, and K. Eshghi. *Inside Solid State Drives (SSDs)*. Springer, 2012.

[21] M. Rosenmüller. *Towards Flexible Feature Composition: Static and Dynamic Binding in Software Product Lines.* Dissertation, University of Magdeburg, Germany, June 2011.

[22] M. Rosenmüller, N. Siegmund, H. Schirmeier, J. Sincero, S. Apel, T. Leich, O. Spinczyk, and G. Saake. FAME-DBMS: Tailor-made Data Management Solutions for Embedded Systems. In *SETMDM*, pages 1–6. ACM, 2008.

[23] M. Saecker and V. Markl. Big Data Analytics on Modern Hardware Architectures: A Technology Survey. In *eBISS*, pages 125–149. Springer, 2012.

[24] M. Schäler, A. Grebhahn, R. Schröter, S. Schulze, V. Köppen, and G. Saake. QuEval: Beyond High-Dimensional Indexing à la Carte. *PVLDB*, 6(14):1654–1665, 2013.

[25] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *VLDB*, pages 1150–1160, 2007.

[26] S. Sunkle, M. Kuhlemann, N. Siegmund, M. Rosenmüller, and G. Saake. Generating Highly Customizable SQL Parsers. In *SETMDM*, pages 29–33. ACM, 2008.

[27] T. Willhalm, I. Oukid, I. Müller, and F. Faerber. Vectorizing Database Column Scans with Complex Predicates. In *ADMS@VLDB*, pages 1–12, 2013.

[28] J. Zhou and K. A. Ross. Implementing Database Operations Using SIMD Instructions. In *SIGMOD*, pages 145–156. ACM, 2002.

[29] M. Zukowski. *Balancing Vectorized Query Execution with Bandwidth-Optimized Storage.* PhD thesis, CWI Amsterdam, 2009.