Constructing the Next 100 Database Management Systems: Like the Handyman or Like the Engineer?

Andreas Geppert, Klaus R. Dittrich
Institut für Informatik, Universität Zürich

Abstract

While the number of database management systems (DBMSs) increases and the various DBMSs get more and more complex, no uniform method for DBMS construction exists. As a result, developers are forced to start more or less from scratch again for every desired system, resulting in a waste of time, effort, and cost. Hence, the database community is challenged with the development of an appropriate method, i.e. the time-saving application of engineering principles (e.g., reuse). Problems related to a construction method are described, as well as approaches towards solutions.

1 Introduction and Motivation

Traditional (e.g., relational) database management systems satisfy the needs of most conventional applications rather well. This is not true for many so-called non-standard application domains like CAx, OIS, multimedia applications, and so forth. Not only were conventional data models (like the relational model) recognized as inadequate, some new applications also require advanced transaction models (e.g., supporting long-lived, cooperative units of work), and subtle integrity enforcement mechanisms.

Since it turned out that realizing this desired advanced functionality on top of existing (relational) systems can lead to severe inherent performance drawbacks, entire new systems have been developed from scratch.

Object-oriented DBMSs [2] are advocated for many non-standard application domains. However, none of these systems will probably be general enough to support a broad spectrum of different application domains (with diverging requirements) equally well.

Consequently, a variety of DBMSs with different models and functionalities are likely to be developed. Hence, it is necessary to approach DBMS construction itself in a more engineering manner. In other words, *implementation efficiency* and *maintenance efficiency* has to be stressed, while database technology in the past (like other areas in computing!) has focussed almost solely on *runtime efficiency.*

An engineering approach to DBMS construction treats DBMSs as products of a "software community" [12] and tries to make extensive use of pre-fabricated artifacts, methods, and tools. Nevertheless, it is clear that simple "reuse" [15] alone does not help much (recall that the implementation on top of relational DBMSs is also one — though not successful — kind of reuse). We still require ideas and experience about what artifacts to reuse, how to describe them, how to encounter the sometimes broad variety of applicable alternatives, and so forth.

Database technology is challenged to develop innovative DBMS construction methods which support the specification, design, and implementation of DBMSs *for various application domains* that make use of proven reuse techniques [15].

The remainder of this paper is structured as follows: section 2 surveys systems and concepts proposed for support of DBMS construction. Section 3 describes the challenges of an engineering-style DBMS construction method in more detail and shows some rough ideas on how to encounter these challenges. Section 4 contains a conclusion.

2 DBMS Construction Systems

In this section, we shall briefly survey work on *extensible database systems* [6] and elaborate on shortcomings of these systems. Also, we describe principles and problems in DBMS construction methodology.

Kernel Systems

Kernel systems (e.g., WISS [7], the kernel of DASDBS [20]) offer a general, fixed interface that supports common functionality required by all or most DBMSs (e.g., physical object management). Upper layers of a DBMS have to be implemented (i.e., programmed) by the DBI[1].

Obviously, the functionality of the kernel is crucial.

Authors' address: Institut fuer Informatik, Universitaet Zuerich, Winterthurerstr. 190, CH-8057 Zuerich, Switzerland. Email: {geppert, dittrich}@ifi.unizh.ch

If the kernel offers too much functionality, it also restricts the opportunities of the upper layers, while too less functionality increases the implementation efforts of the DBI.

Customizable Systems

This type of systems (e.g., STARBURST [14]) are based on a complete DBMS that can be modified/extended in order to satisfy new requirements. Hence, the basic DBMS is customized to a concrete, new DBMS. In principle, the modifications are performed at code level and therefore require advanced programming skills [14]. Even if some functionality can be realized in a rule-based manner, a high degree of experience (concerning the base system and DBMS technology) is mandatory.

Toolkit Systems

Toolkit systems (e.g., EXODUS [5]) offer a library of modules, which in turn realize alternative techniques for a given task (e.g., physical access paths). However, at least in the plain approach, selection of techniques is not supported and left entirely to the DBI. Likewise, defining an appropriate architecture and linking together the selected modules is also the task of the DBI.

Generator Systems

Generator systems (e.g., GENESIS [4], the EXODUS optimizer generator [13], DMC [16]) support the specification of (parts of) a DBMS functionality and the generation of DBMS components based on those specifications. However, they mostly address only one aspect (e.g., the data model) or even assume a fixed data model (for instance). Hence, new features cannot be easily integrated, nor is the choice among alternative realization techniques supported.

Principles of DBMS Construction Systems

The analysis of the aforementioned DBMS construction systems shows that they rely on some kind of *reuse* (in the broader sense). Essentially, there are two basic principles of reuse: *generation* and *configuration* (or *composition)*.

Generation refers to the automatic creation of code components that realize the mapping from one interface to a lower-level one, or to the automated coding of a specification. Hence, generation is reasonably applicable in cases where a general and unique mapping can be found and an underlying formal model exists. It is not reasonable (at least not without large-scale DBI intervention) if a broad variety of realization alternatives exist.

These cases, on the other hand, are well supported by configuration. Configuration means to select appropriate primitives and to plug them together into an operational system.

Requirements and Features

Based on the short survey given above, we infer that a DBMS construction method should satisfy the following requirements:

1. Architecture
   A generic yet adaptable architecture model is required that is applicable for a broad range of significantly different DBMSs.
2. Knowledge Representation
   Knowledge about database technology (e.g., on alternative realization techniques for a specific task, or on experiences of previous designs) has to be expressed.
3. Design for Reuse
   Design for reuse has to be enforced, i.e., decomposition of techniques and components into easily reusable artifacts.
4. Specification-Based Design
   As far as possible, the construction method should be specification-based (e.g., ACTA [8] for transaction management).
5. Design Completion and Integration
   Based on the architecture design and the requirements specifications, the method has to support completion of the design and the retrieval of adequate artifacts. Based on the architecture framework and the selected techniques, the construction method should support the integration of required classes (be they generated or composed). This assembly results in an operational DBMS (at least, partially).

3 Towards a DBMS Construction Method

In this section, we shall address each of the aforementioned requirements and corresponding possible solutions in more detail.

3.1 Architecture of Configured DBMSs

Although reuse is often understood as reuse of *code fragments*, reuse of *analysis and design information* is probably much more promising [3], [18]. For DBMS construction, the prerequisite and first result is a clear understanding of the architecture of the resulting DBMS. Since the DBMS architecture is also the result

1. DBMS Implementor

of analysis and design, a DBMS construction method should support the *reuse of DBMS architectures*.

The question arises whether always the same architecture can be reused or whether there have to be multiple ones for different DBMSs. In fact, we feel that multiple architectures will be necessary (even in conventional cases there is no uniform architecture that all DBMSs conform to). Note that current extensible DBMSs either leave architectural design to the DBI or use a fixed architecture.

Given a variety of architecture skeletons (amongst which the DBMS designer selects the one that satisfies his/her needs best), we have the problem to *identify* a pre-defined set of architectures, to *represent* them and to support *selection* of appropriate ones. Another problem is how refinement (implementation) of a selected architecture can be supported. In other words, beside the problem of *DBMS* architecture, a construction method also has to address the architecture of the entire *DBMS construction environment*.

> **Challenge 1:** Can a set of architecture frameworks be identified, which is small but also applicable to a broad spectrum of DBMSs? On which level of abstraction can these architectures be described?

To a large extent, these questions are answered by the so-called *domain analysis* [9], [15]. Domain analysis usually yields the area in which the construction method (e.g., generation) has to be applied (trivially, DBMSs in our case). Domain analysis addresses the following questions:

1. Recognizing the domain: where can reusability successfully be applied? What are the common requirements and properties of the systems to be constructed?
2. Identifying domain boundaries: what is the *domain range*, i.e., the range of cases the method should support? Which are the interfaces to other parts (of an enclosing system or other systems)?
3. Identifying variants and invariants: which parts can be fix (e.g., realized hard-wired) and which parts have to be variable?
4. Defining specification input: for the variant part, does a unified (meta) model exist that can be used for a specification of the domain range?
5. Determining layers: given a level of abstraction for specifications (see question 4), what is the appropriate level of primitives for the implementation? In other words, in terms of which interfaces/platforms can the specification be realized? Is there a unified mapping of specifications into the level of primitives?
6. Alternatively, are there "many" possible techniques to realize cases within the domain range?

> **Challenge 2:** Which parts of a DBMS are promising subjects to reuse? Which parts can remain invariant but are nevertheless widely applicable without performance drawbacks?

The answers to some of the questions above seem obvious in the sense that there cannot be one uniform approach to DBMS construction in its entirety (e.g., there is no DBMS meta model with significant domain range). Hence, it is advisable to recursively identify several DBMS *aspects* and to apply domain analysis to them. Such aspects may be object management, transaction management, integrity maintenance, and so on. While a uniform approach cannot be found for the development of an entire DBMS, it should be possible for the various aspects. However, in this case identification of boundaries has to be stressed.

The results of a sample domain analysis might read as follows: the kernel approach is extended in that a fixed, low-level kernel (object server) is supported. This component offers functionality which is likely to be required by all DBMSs, like physical object management and transaction control for low level storage objects. Multi-level transaction management [21] is applied. The object server is regarded as an invariant for the constructed DBMS, i.e. generation/configuration is not regarded as necessary here.

Application domain dependent functionality (e.g., data model and transaction management for data model objects) is partitioned into aspects, which in turn are assigned to (i.e., realized by) *subsystems*. These subsystems are realized on top of the low-level kernel. The upper subsystems are subdivided further into those that realize fixed functionality right away (again, invariants). For instance, a subsystem *user management* realizes the management of a user and group concept. Since for that task a general concept can be offered and can later on be used in parts or in its entirety, no configuration/generation is necessary. For other subsystems like object management or transaction management, no general (efficient) solution can be realized right away. Therefore, these subsystems will have to be configured or generated. For each of these subsystems/aspects, the process described above may be necessary to continue recursively, since it may be the case that one part of a subsystem can be reasonably generated while others have to be configured. Take transaction management as an example. While the transaction manager (the component that e.g. keeps track of inter-transaction dependencies) can be subject to generation, other components like the scheduler will require configuration due to the large number of alternative implementations.

Challenge 3: Which parts of a DBMS can be generated (since they can be described formally and/or a uniform realization exists)?

Summarizing, reuse of analysis and design is supported by a DBMS architecture framework (which reflects results of domain analysis, but also represents previous design experiences). A set of aspects that potentially can be addressed by a DBMS construction and a collection of subsystems will be identified. Each of the subsystems can be regarded as a collection of classes or objects, hence has an interface (in the object-oriented sense) and an internal state invisible for the outside world. Starting with such a framework, DBMS construction can add additional subsystems and will detail the functionality of each of the subsystems.

3.2 Knowledge Representation

In the previous section, we stressed the importance of reusability for DBMS construction. If we aim at the computer-assisted reuse of designs and implementations, how can we represent these reusable artifacts?

To that end, *abstraction* is one crucial issue (i.e., the degree of implementation independence): the more abstract a representation, the easier is reuse of represented artifacts [15]. Since selection (and modification) have to be performed for any reused artifact, *specificity* of artifacts is another problem [3]. If reused artifacts are "large", their reuse payoff is high. Nevertheless, large artifacts are also very specific and therefore are applicable without large-scale modifications only in very few cases. Small and generic artifacts, on the other hand, are easier to find (select), but require composition and thus render lower payoff. Additionally, for small artifacts it is easier to recognize differences and similarities, and selection and ranking of (partially) adequate artifacts is easier as well.

Challenge 4: How can we describe our knowledge on database technology such that
- it is reusable,
- represented facts are not too specific,
- a broad domain range is ensured?

Beside the concept of subsystems (introduced above), the following concepts for the representation of knowledge will be useful for the cases that require composition and configuration. In these cases, there are usually different ways to achieve specific functionality. Different *strategies* are identified where each of them describes a collection of alternative implementations that are similar under one specific point of view. They are realized by *techniques,* where the way how a technique realizes a given task is (partially) predetermined by its strategies. As an example, take concurrency control. There, one criteria will be the *validation time*, i.e. the point in the execution of a transaction when the correctness of its operations is checked. Concretely, we can distinguish strategies for preclaiming, pessimistic, and optimistic validation. One technique for the pessimistic strategy is strict two-phase locking.

Furthermore, the concrete use of techniques will still depend on the properties of other parts (of the DBMS). Hence, they are not useful *atomic units* of reuse, but have in turn to be composed. Atomic units (i.e., artifacts that — beside instantiation — do not require modification/composition) are termed *primitives*.

The benefit of this three-level approach is twofold: first, it stresses the important distinction between strategies and techniques (or policies and mechanisms), and therefore suggests a design from the abstract to the concrete. Second, it can serve as a hook for selecting similar artifacts (where techniques within the same strategy are more similar than those in different ones). Beside this distinction, other relationships (e.g., expressing inconsistent or consistent combinations) will be necessary.

The approach described so far is still static, while DBMS construction (for multiple systems) is dynamic in the sense that occasionally new solutions are invented and in that experience of designers is a very important factor. Actually, many systems have been implemented at least twice; after weaknesses of a prototype have been recognized, the system is redesigned and implemented a second time. Hence, another problem related to knowledge representation is that it is desirable to represent experiences of designers. In the ideal case the construction method can also learn and consider experiences of previous designers.

Challenge 5: Can we classify and structure our knowledge of database technology? Is such a classification (e.g., in terms of strategies and techniques) possible at all? Can we represent our experience (uniformly)?

Although different concepts and models may be adequate to represent the various kinds of knowledge, it is desirable to have one uniform underlying model for knowledge representation (e.g., the Telos model [17]).

3.3 (Component) Design for Reuse

The configuration approach relies heavily on the representation and storage of primitives and techniques in a software information base. Nevertheless, these software artifacts can only be used successfully if they were designed in such a way that reuse is supported well (*design for reuse*).

First, it is not sufficient to represent classes by their interfaces and implementations only, i.e. source code [12]. In order to enable selection of classes and primitives at configuration time, additional information describing the semantics of primitives and classes has to be attached. However, because of the complexity of such component specifications it seems obvious that they cannot be given in a fully formal manner (e.g., algebraically).

Second, design for reuse also means appropriate decomposition of artifacts (recall the aforementioned notion of "specificity"). Techniques, for example, are reasonably decomposed if the resulting (atomic) parts can be plugged together easily to (possibly new) techniques without many modifications. Hence, given a a collection of techniques, it may be necessary to redesign them in order to get them properly decomposed. As an example, take multi-version concurrency control, where a redesign (for reuse) leads to the separation of version control and concurrency control [1].

Challenge 6: By which (DBMS-specific) concepts can we represent knowledge about the semantics of components designed for reuse? Is it realistic at all that DBMS researchers cooperate in decomposition and reuse?

3.4 DBMS Specification

Reasonable DBMS construction cannot start with the design (refinement) of a subsystem right away, since the variety of alternatives will usually be too complex for the "average" DBI. Additionally, whereever possible, a high level of abstraction is desirable [15]. For instance, a DBI will usually prefer to "think" in terms of rules for transaction dependencies [8] instead of technical terms like validation time. Hence, a DBMS construction method also has to support the specification of the semantics of the various subsystems. Nevertheless, the corresponding aspects refer to rather different universes of discourse. In terms of domain analysis, one specification language for entire DBMSs would be too complex. Therefore, one specification language per aspect seems desirable (e.g., one for transaction models, one for data structures, and so forth). However, if multiple specification languages are suggested, the question is once more how these languages interface with each other (domain boundaries).

Challenge 7: Which aspects of a DBMS can be described by specification languages? What are reasonable languages for the various aspects? How can they "exchange" information?

What is the result of a specification (for a specific aspect)? Obviously, the result of such a specification may be twofold: input to a generator (e.g., an optimizer generator [13]), or input for configuration/composition. For composition, the specifications describe the abstract semantics of aspects. Additionally, specifications can be used to derive *requirements* for the subsystems to be configured. In this case an aspect specification is transformed into a requirements specification. This is also the step from the aspect level to the subsystem level, or from the abstract to the more concrete. Naturally, while the specification languages are aspect-specific, a reasonable requirements language should be uniform over the subsystems. Crucial issues to be expressed in such a requirements language are the strategies (and techniques) to be used by a subsystem.

Challenge 8: What are the features of a requirements specification language dedicated to DBMSs?

3.5 Design Completion and Integration

So far, we derived reuse of designs, knowledge representation, support of specification and requirements languages as necessary features of a construction method. However, two further issues are still open: how to complete the design and how to integrate reused components.

For the first, the requirements language supports the specification of necessary features of artifacts to be reused. Obviously, a construction method supports the selection of artifacts matching the specified requirements. In other words, in a given case the requirements specification restricts the potentially large search space of reusable artifacts. However, a requirements specification (as the outcome of aspect specifications) can still be partial and incomplete; hence it has to be possible to let the DBI interactively edit and complete requirements specification.

Simultaneously, after artifacts to be reused have been generated or selected, they have to be *integrated* to an operational DBMS or a part of a DBMS. In object-oriented terminology, *objects* have to be *introduced* and *bound* together in order to cooperatively support some task (*scripting* [19]). In this context, another interesting point is how scripting can support the integration of generated and configured/selected components.

Challenge 9: How well can generators and configuration tools be integrated? Which is a scripting model well-suited for DBMS design?

4 Conclusion

One might ask whether a construction method really is a challenge for future database technology, or whether it is simply an application of proper software engineering.

In fact, DBMS construction can benefit from software engineering approaches and avoid pitfalls that have been experienced there. Nevertheless, many principal problems outlined in this paper are not solved by software engineering in a satisfying manner. Additionally, as often mentioned [9], [18], constructing reusability environments can only succeed if it is performed by people experienced in the domain (thus, DBMS technology).

Consequently, the construction method is a challenging topic for the intersection of software engineering and database technology:

1. Up to now, no adequate, generic architecture model for DBMSs is known[2], whereby "designing a framework is itself research" [23]. Frameworks seem to be a reasonable solution for the architecture problem (for instance, the *A La Carte* approach [11] also relies on frameworks for the integration of heterogeneous DBMSs). Especially, a construction method can prove that such architecture frameworks are applicable with comparable runtime performance.

2. The construction method relies on a model for the description of strategies and techniques for database management, and a concrete taxonomy of those techniques. Obviously, such a widely accepted taxonomy does not exist and can only be developed with a precise knowledge on database technology.

3. With some exceptions (e.g., ACTA [8]) specification languages for various DBMS aspects for the use in configurable DBMSs are unknown.

2. Although such an architecture recently has been proposed for Open OODB [22], it remains still unclear how and by which mechanisms this architecture can be tailored to a concrete DBMS.

On the other hand, we feel that an engineering approach to DBMS construction based on reuse is not an unrealizable *ideal*. It has been shown e.g. for compiler construction that reuse (generation, in the compiler case) can be effective.

While the development of a construction method for DBMSs increases the amount of work on the principles of DBMS development (which — beside scientific insights — has no payoff), it decreases the work necessary for concrete DBMSs on the other hand. Hence, a construction method is of great *practical* use: in the short run, it can serve as an experimental framework for validating new developments. In the long run, it can be used for products and thus decrease construction and maintenance efforts significantly.

5 References

[1] D. Agrawal, S. Sengupta: *Modular Synchronization in Multiversion Databases: Version Control and Concurrency Control*. Proc. ACM SIGMOD Intl. Conf. on Management of Data 1989.

[2] M. Atkinson, F. Bancilhon, D.J. DeWitt, K.R. Dittrich, D. Maier, S.B. Zdonik: *The Object-Oriented Database System Manifesto (a Political Pamphlet)*. Proc. Intl. Conf. on Deductive and Object-Oriented Database Systems 1989.

[3] T.J. Biggerstaff, C. Richter: *Reusability Framework, Assessment, and Directions*. IEEE Software, July 1987.

[4] D.S. Batory, T.Y. Leung, T.E. Wise: *Implementation Concepts for an Extensible Data Model and Data Language*. ACM Trans. on Database Systems 13:3, 1988.

[5] M.J. Carey, D.J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J.E. Richardson: *The Architecture of the EXODUS Extensible DBMS*. In [10].

[6] M. Carey, L. Haas: *Extensible Database Management Systems*. In W. Kim (Ed.): Special Issue on Directions for Future Database Research and Development. ACM-SIGMOD Record 19:4, 1990.

[7] H.-T. Chou, D.J. DeWitt, R.H. Katz, A.C. Klug: *Design and Implementation of the Wisconsin Storage System*. Software - Practice and Experience 15:10, 1985.

[8] P.K. Chrysanthis, K. Ramamritham: *ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior.* Proc. ACM-

SIGMOD Intl. Conf. on Management of Data 1990.

[9] J.C. Cleaveland: *Building Application Generators*. IEEE Software, July 1988.

[10] K.R. Dittrich, U. Dayal (eds.): Proceedings of the 1986 International Workshop on Object-Oriented Database Systems. IEEE Computer Science Press, 1986.

[11] P. Drew, R. King, D. Heimbigner: *A Toolkit for the Incremental Implementation of Heterogeneous Database Management Systems*. VLDB Journal 1:2, 1992.

[12] S. Gibbs, D. Tsichritzis, E. Casais, O. Nierstrasz, X. Pintado: *Class Management for Software Communities*. Special Issue on Object-Oriented Design, CACM 33:9, 1990.

[13] G. Graefe, D.J. DeWitt: *The EXODUS Optimizer Generator*. Proc. ACM SIGMOD Intl. Conf. on Management of Data 1987.

[14] L.M. Haas, W. Chang, G.M. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M.J. Carey, E. Shekita: *Starburst Mid-Flight: As the Dust Clears*. IEEE Trans. on Knowledge and Data Engineering 2:1, 1990.

[15] C.W. Krueger: *Software Reuse*. ACM Computing Surveys 24:2, 1992.

[16] F. Maryanski, J. Bedell, S. Hoelscher, S. Hong, L. McDonald, J. Peckham, D. Stock: *The Data Model Compiler: A Tool for Generating Object-Oriented Database Systems*. In [10].

[17] J. Mylopoulos, A. Borgida, M. Jarke, M. Koubarakis: *Telos: Representing Knowledge About Information Systems*. ACM Trans. on Information Systems 8:4, 1990.

[18] J.M. Neighbors: *Draco: A Method for Engineering Reusable Software Systems*. In T.J. Biggerstaff, A.J. Perlis (eds.): Software Reusability. Volume I: Concepts and Models. ACM Press, 1989.

[19] O. Nierstrasz, D. Tsichritzis, V. de Mey, M. Stadelmann: *Objects + Scripts = Applications*. In D. Tsichritzis (ed.): Object Composition. Centre Universitaire d'Informatique, Universite de Geneve, Geneva 1991.

[20] H.-B. Paul, H.-J. Schek, M.H. Scholl, G. Weikum, U. Deppisch: *Architecture and Implementation of the Darmstadt Database Kernel System*. Proc. ACM SIGMOD Intl. Conf. on Management of Data 1987.

[21] G. Weikum: *Principles and Realization Strategies of Multilevel Transaction Management*. ACM Trans. on Database Systems 16:1, 1991.

[22] D.L. Wells, J.A. Blakeley, C.W. Thompson: *Architecture of an Open Object-Oriented Database Management System*. IEEE Computer 25:10, 1992.

[23] R.J. Wirfs-Brock, R.E. Johnson: *Surveying Current Research in Object-Oriented Design*. Special Issue on Object-Oriented Design. Communications of the ACM 33:9, 1990.