Hannes Voigt, Kai Herrmann, Tim Kiefer, Wolfgang Lehner

**Listen to the customer: model-driven database design**

**Erstveröffentlichung in / First published in:**
DOI: [https://doi.org/10.1145/1866480.1866483](https://doi.org/10.1145/1866480.1866483)

Diese Version ist verfügbar / This version is available on:
[https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-805550](https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-805550)
Listen to the Customer – Model-driven Database Design

Hannes Voigt, Kai Herrmann, Tim Kiefer, Wolfgang Lehner
Dresden University of Technology
Database Technology Group
01062 Dresden, Germany
{firstname.lastname}@tu-dresden.de

ABSTRACT

In modern IT landscapes, databases are subject to a major role change. Especially in Service-Oriented Architectures, databases are more and more frequently dedicated to a single application. Therefore, it is even more important to reflect the application requirements in their design. Software developers and application experts formulate application requirements in software models. Hence, we obviously need to bridge the gap to the software world and directly derive a database design from the software models used in application development and maintenance. We introduce this concept as model-driven database design. In this paper, we present the architecture principles of a model-driven database design tool and details on the enumeration and evaluation of logical database designs.

1. INTRODUCTION

In the last decade, the IT architecture of enterprises has changed dramatically. Following the building plans of the Service-Oriented Architecture (SOA), today’s enterprise IT landscapes comprise a large number of decentralized and specialized application systems, loosely coupled via Web services and orchestrated with business processes. SOAs package functionality into separated interoperable and exchangeable services with decentralized deployment and maintenance. Simple and stateless service interfaces make the functional units accessible over the network, while hiding as many implementation details as possible. Where loose coupling, interoperability, and exchangeability of functional units are the architectural doctrine, every functional unit comes with its own data storage layer. Hence, the role of database systems is fundamentally different from the classical highly centralized, integrated, application-invariant enterprise database. In SOAs, database systems are no longer the fixed stars of the enterprise universe. Rather, a database system is dedicated and specific to a single application, like a satellite orbiting around a planetoid. Despite their new role, databases are as important as before; it is still worthwhile and necessary to tailor them to the applications.

The design and optimization of databases are elaborate and extensive tasks. Out of the need to perform this task for a steadily increasing number of databases, the idea of automatic cost-based design has received a lot of research interest and has also been transferred into productive tools over the last decade. All these tools automatically select physical design structures such as indexes, partitioning schemes, and materialized views based on cost estimates for a given SQL workload. This is appealing because these design tools relieve the DBA from conducting expensive examinations and analyses to find the optimal design. However, the tools address only the needs of the DBA and have been developed in the spirit of the fixed-star database systems. Database applications are seen merely as satellites keeping the fixed star busy by sending SQL statements and waiting for result sets. For a database dedicated to a single application, the application becomes the gold customer. Undeniably, you really have to listen to your gold customer if you want to provide good service. You better speak your customer’s language and you better understand all his needs. Software developers and application experts express application requirements with the means of well-understood software models and not in SQL. We envision a model-driven database design that tightly integrates automatic database design into the software development. A model-driven database design tool is directly usable by software developers and application experts; nonetheless, it still relies on cost estimates for selecting an optimal database design.

Setting up such a model-driven database design tool poses two major challenges: a conceptual and an architectural challenge. First, SQL as a workload description can be used directly in the database design process, because SQL describes the application behavior in operational primitives of the database world. Software models describe the application and its behavior in primitives of the software world. So, the conceptual challenge is the translation of modeled application requirements into database primitives to use them in the database design process.

Second, with software models being the primary methodology of modeling, the former sophisticated task of database modeling declines to a mere model transformation. Developers do not model the world twice; they derive a logical database schema from their software models by applying predefined rule sets or rules of thumb. This neglects that the logical schema has considerable influence on the performance of the database system. A universally optimal transformation from a software model to a logical database schema does not exist. To decide which transformation is optimal for an application, the different resulting designs must be analyzed and compared based on the cost estimated by the database optimizer for the application’s workload. So, the architectural challenge is the integration of logical design into the model-driven database design tool. To lay the foundation for solving the challenges, we developed a principle architecture of an automatic database design tool.
2. MODEL-DRIVEN DATABASE DESIGN

In the conventional model-driven development of database applications, the design of the database is divided into two completely separated steps, depicted in Figure 1(a). Within a first step, a software developer derives a logical database schema from software models. For a software developer, the logical database schema is a platform-specific software model, and its derivation is a task of adding platform specifics. Model transformation tools help with this. They take less specific models, annotate platform specifics and a set of rules, and produce the required more specific models. Comparing different derivations based on a cost analysis is not considered.

In the second step, a database developer considers an application workload and data statistics to create a physical database schema for a given logical database schema, therefore also adding specifics to a model. The specifics of physical storage are added to the logical storage model. Nevertheless, in contrast to the derivation of the logical schema, specifics are added with the objective of reducing the execution cost of the considered workload. A database developer analyses and compares different physical schemas to find the best one for the given setting. Physical design tools are of great assistance by performing most of the task automatically. Given a database and a workload, physical design tools estimate the cost of a large number of different possibly reasonable solutions directly within the database system and pick the best one. However, these tools are limited to database primitives and semantics. Software models as a means of describing data schemas, data statistics and application workloads are out of scope.

Our goal is to provide a database design tool that unites these two steps; a design tool that optimizes the logical and physical design of a database for a given application. With the help of such a database design tool, as depicted in Figure 1(b), software developers can derive a cost-optimized logical and physical database schema directly from the software models. By using software models as the sole input, the database design tool speaks the language of the software developer. It allows the developer to formulate requirements for the data persistence in an application-centric way. The developer annotates the software models with persistence-specific information, such as, which class of data objects has to be persisted, how many data objects need to be managed, or which activities perform which operation on which data objects. Based on these annotated models, the database design tool considers a large number of possibly reasonable logical and physical designs. It derives the application’s database workload from the models and estimates the workload’s execution cost for each of the different designs. By this means, it can analyze the impact of a design on the workload and pick the best for the given application.

Figure 2 shows the principle architecture of our model-driven database design tool. The tool uses structure models, behavior models, and non-functional properties annotated to the models (Section 3). It implements a heuristic to generate candidate designs and the respective database workload for each design. Furthermore, our tool realizes so-called what-if databases to analyze the impact of a candidate design on the respective workload directly within the database system. Based on this analysis, the tool selects the best design found for the given application. We focus this presentation on the enumeration of logical design candidates (Section 4) and their evaluation (Section 5). In the remainder of the paper we present experiments (Section 6) and related work (Section 7) and finally we conclude the paper (Section 8).

3. USE OF SOFTWARE MODELS

Application developers use software models (mostly UML) as the standardized, structured and well-understood way to describe functional and non-functional aspects of software components on different levels of technical abstraction. In modern model-driven software development architectures, software models are an integral part of the development and maintenance and can be seen as high-level source code. The different kinds of software models express different aspects of the application. Structure models such as a class model describe the static nature including the data schema of the application. In comparison, behavior models such as an activity model describe the dynamic nature of the application, i.e., what the application potentially does.

For model-driven database design the standard software models lack elements to express persistence specifics. We classify persistence-specific information according to two orthogonal criteria. First, we distinguish between the static and the dynamic persistence nature of the application. The static nature describes which application data needs persistence. The dynamic nature describes which operations will be performed on the persisted data. Second, we distinguish between functional aspects and non-functional aspects of persistence.

Based on UML Profiles, we developed rudimentary mechanisms to annotate software models with these various persistence specifics.

4. CANDIDATE ENUMERATION

To find the best database design for the modeled application, we generate a large number of different designs. Conceptually, we divide the generation of database design candidates into two levels of enumeration. In the first level, the design tool generates logical schemas. In the second level, it considers different physical
schemas for each individual logical schema. We focus on the logical schema enumeration; for physical schema enumeration, we rely on common algorithms from related work.

The basis for the logical schema enumeration is a class model given as input to the design tool. Each candidate schema can be derived from the class model by a specific model transformation. Figure 3 shows how we internally represent the class model, model transformations and the resulting logical database schema. Class models consist of three types of elements that are relevant for us: classes, attributes, and associations. For every type of elements, there exists different element transformations. Associations may be transformed into a combined table or separated tables. For classes, we have three different element transformations: vertical partitioning, horizontal partitioning and universal tables. (We do not consider transformation alternatives for attributes here, so they are grayed out in the figure.) Logical database schemas consist of tables, columns, foreign key constraints and primary constraints (not shown in the figure). The element transformations define the relational schema elements generated for the class model elements.

Each possible model transformation associates the model elements with one element transformation instance. Figure 4 illustrates this with an example. The shown model transformation associates the class Place and its children with the horizontal partitioning transformation and the association postalCode with combined-table transformation. The classes PopulatedPlace, LunarCrater and PostalCode are associated with vertical partitioning, which is the default for non-parent classes.

Although the number of possible element transformation alternatives is very small, the number of possible model transformation alternatives is high. Every combination of element transformation choices yields another logical database schema; for n elements with k possible transformations, there exist \(k^n\) possible combinations. Hence, enumerating all combinations has an exponential complexity and is not feasible for realistic class models.

We propose a heuristic greedy enumeration algorithm. The basic idea is to start with the most fine-grained logical schema and coarsen it a bit in each iteration as long as the resulting logical schema yields better estimated execution cost. Figure 1 depicts the algorithm. Initially, we associate every class with horizontal partitioning and every association with separated-tables transformation (line 2). We also add to each association and class a list of the remaining possible element transformations. Vertical partitioning and universal tables are only relevant for classes with parent or child classes. The combined table is only reasonable for 1:1 associations and 1:N associations with a low N. After determining the estimated execution cost for the initial schema (line 3–4), we search the single element transformation alternative that reduces execution cost most. We therefore iterate over all elements that have transformation alternatives left, derive a new logical schema by using the alternative transformation of the considered element, and determine the execution cost of this schema (line 6–14). If we find a more beneficial schema, we take this as the current best schema and repeat our search; otherwise we stop (line 15–17). Finally, we return the identified schema alternative.

For n elements with k possible transformations, we have to check \(n \cdot k\) transformations to find the best next logical schema, and there can be at most \(n \cdot k\) best next schemas. Hence, the complexity of our greedy enumeration algorithm is \(O((nk)^2)\).

Our enumeration algorithm associates the model elements with one element transformation instance. To derive the logical schema, we additionally go over all model elements and add the logical schema elements (tables, columns, foreign keys) to our internal representation. Of course, the schema elements we add for a specific model element depend on its associated element transformation. Algorithm 2 illustrates the basic procedure. For clarity of presen-
5. WHAT-IF DATABASES

What-if databases are the prerequisite for the design tool to evaluate candidate designs. Classical what-if interfaces used by physical design tools allow to evaluate the impact of physical design structures on a workload. What-if databases go one step further. They allow to evaluate the impact of the complete database design, i.e., table structures, data statistics, and physical design structures. The impact of a design is evaluated with cost estimates of the optimizer for a given workload. Conventionally, a database has to be present, including tables, constraints, indexes, and the data, to retrieve optimizer cost estimates. Creating all this for each design candidate is extremely expensive and inconvenient. Therefore, what-if databases exist only in the system catalog. If the design tool creates a what-if database, it manipulates the database system catalog. For each database object as part of the design, including tables, columns, constraints, and physical structures such as indexes, it adds the necessary entries to the catalog. Additionally, the tool adds entries to the statistics tables in the system catalog for each data statistic derived from annotations in the software models. Since the optimizer relies only on its catalog information for query optimization, we can now obtain cost estimates for a query by regularly posting a logical query execution. The optimizer translates the query into a query execution plan and returns the estimated cost of this plan without physically executing it. Consequently, by the means of what-if databases, our design tool is able to easily and time-efficiently evaluate the impact of a database design on an application workload based on real optimizer cost estimates. Finally, the evaluation of the design candidates yields the design offering the lowest cost for the modeled application.

6. EXPERIMENTS

To evaluate our model-driven database tool, we conducted a series of experiments. We implemented a prototype of the proposed tool in Java on top of a commercial database management system. All experiments were performed on an Intel Core 2 Duo machine clocked with 2 GHz, equipped with 2 GB of RAM and a 160 GB large hard disk, and running Ubuntu 9.10 64 Bit Linux operating system.

For the experiments, we built a synthetic scenario around data from DBpedia [3]. The DBpedia ontology offers a hierarchy of types, which are additionally cross-linked with associations. Kim et al. describe a transformation from ontologies to UML class models [8]. We used this transformation to generate UML class models as input to the design tool. To get models of varying sizes, we took
sub-trees from the DBpedia type hierarchy. We denote each sub-tree by its most general type, e.g., Person denotes the type Person and all its direct and indirect sub-types including all associations between these types. We also used combinations of sub-trees, indicated by a composite name, e.g., Event-Organization denotes the combined sub-trees Event and Organization. For each ontology sub-trees or sub-tree combination in use, we generated a UML class diagram and annotated each class with the corresponding cardinality in the DBpedia dataset. We set the cardinality of each class to the number of instances of its corresponding type in the DBpedia data set. Additionally, we annotated all classes and their attributes to be persistent.

Aside from the class diagrams, we synthetically generated activity diagrams describing different workloads. For each of the four basic database operations, Update, Delete, Insert, and Select, we generated five workloads of 30 queries each. We created the workload as follows: In an activity model, each activity represents a database operation and is annotated with the class it operates on. Based on this relationship between class and operation, a set of database activities on a class hierarchy can be generally distinguished (1) according to their vertical tendency, i.e., how activities are distributed among the generalization levels, and (2) according to their horizontal tendency, i.e., how activities are distributed among the classes of one level. By varying these two dimensions, we defined the five workloads illustrated in the top of Figure 5. In the icons, the gray rectangle marks the hot spot of database activities in the space of all possible classes. In short, for each basic database operation, the described procedure generates one workload with activities equally distributed over the classes and four workloads with an activity hot spot. Additionally, for the Select activities, we annotated a subset of a class’ associations as join relationships. For that, associations were picked with a probability of 0.4; maximal join depth was 2.

In the first group of experiments, we compared the database designs found by the design tool with the naive design (horizontal partitioning for parent class and separated-tables for associations). With the greedy enumeration, we determined the best and the worst non-naive transformation of the Event-Organization model for each workload. Figure 5 shows the performance on the naive design relative to the best and to the worst design in each setting. The left end of each slider bar represents the performance of the best design; whereas the right each slider represents the performance of the worst design. The slider itself shows the performance on the naive design. As we can see, in most cases, DBs designed using our design tool performed better than when using the naive design. In none of the cases did our tool yield a worse design. This is as expected, because the greedy enumeration starts with the naive design and searches for better designs. Furthermore, we can see that for most of the workloads, there exist considerably worse performing designs. Consequently, conducting a detailed analysis of transformation alternative is important to accomplish a good database design.

In the second group of experiments, we examined our greedy enumeration algorithm. We counted the number of designs considered by the greedy enumeration for class models of varying sizes using the Select-bottom-equal workload. We compared these measured values with the number of absolutely possible designs and the number of designs considered by the greedy enumeration in the worst case. Figure 6 depicts the results. The figure shows the different class models ordered by the total number of possible element transformations, which reflect the complexity of the model for the transformation. As can be seen, the number of absolutely possible designs increases exponentially with the complexity of the class model, whereas our greedy enumeration keeps examining only a small number of designs. For the very small models, we were able to run our tool also with a complete enumeration of all possible designs. On all these models, the greedy enumeration found the same design as the complete enumeration. Although we were able to compare the enumerations only for very small models, because of the runtime complexity of the complete enumeration, we concluded that the greedy enumeration finds sufficiently good designs in affordable time.

In the third group of experiments, we examined the runtime of single what-if database roundtrips for class models of varying sizes. What-if database roundtrips are the most expensive part of the design tool’s analysis procedure, so their runtime is crucial for the runtime of the complete procedure. We measured the runtime of the what-if roundtrip for horizontal partitioning of parent classes and separation of associated classes. According to what-if runtime, this is the worst case transformation because it results in the highest number of database schema elements possible for a given model. Figure 7 depicts the results relative to the smallest model. It is easy to see that the runtime of a single what-if database roundtrip increases linearly with the number of schema elements in the what-if database. This is as expected, because for each element, the what-if interface has to manipulate the system catalog.

7. RELATED WORK

Our model-driven database design relates to a broad range of existing work, mainly in the fields of software technology and automatic database tuning. Model-driven software development has been the main trend in software technology over the last decade. The Object Management Group (OMG) formulated a couple of defactor standards in this field: (1) the Model Driven Architecture (MDA) in 2001 [9], which describes the stacking of models of increasing technical abstraction; (2) the Meta Object Facilities Query/View/Transformations (MOF QVT) standard, which defines languages and facilities for transforming models into other models.

Our model-driven database design tool fits well into the MDA concepts because it realizes a model transformation from a database-independent software model to a database-dependent model. Demuth and Hüllmann presented a work about the transformation of object constraints modeled in UML to database constraints [7]. However, they do not consider the optimization of database design. Nevertheless, their work is an important complement to ours, because constraints have great influence on the database optimizer’s choices and thereby on accurate design decisions. Philippi proposed a tool for the model-driven generation of logical database schemas according to a developer-specified degree of performance optimization [10]. However, Philippi’s tool is based on fixed rules and is not linked to the database optimizer. Our tool uses the database optimizer to analyze transformation alterna-
tives according to estimated cost the data management poses to the application, and it finds the transformation that yields the lowest cost.

Over the last two decades the database community has massively driven the research on automatic database tuning tools, which also includes database systems equipped with means for cost analysis. The elementary peephole into the database optimizer is the so-called what-if interface [4, 6]. It allows optimizer cost estimations for queries under a potential physical design configuration. Based on the what-if concept, many tools for cost-based optimization of physical designs have been proposed; including the optimization of indexes [5], materialized views [1, 11], partitionings [2]. Our work extends the idea of cost-based design optimization to the logical database schema and presents the next consistent step in this line of development.

8. CONCLUSION

We presented the novel concept of model-driven database design. It aims at bridging the gap from database technology to the software world. Model-driven database design ties automatic database design directly to software models. This concept is appealing because it offers three major advantages: First, model-driven database design allows to directly integrate application requirements in the database design and hence to tailor the design to the needs of an application. Second, model-driven database design extends the automatic design to the complete database and therefore gains additional benefits. Third, model-driven database design brings the merits of automatic database design directly to the software developer and consequently makes an optimal database design affordable for a broader range of applications. We discussed the architecture principles of our model-driven database design tool. In more detail, we presented the use of software models, the enumeration of alternative database designs, the generation of corresponding workload and the concept of what-if databases.

For the future, we plan to extend our tool in three major directions. First, we consider the proposed UML profile as a first step version. For real applications, its expressiveness is too small. We plan to extend the profile with means to express a broad range of common functional and non-functional aspects of persistence, for instance, aggregations or data distributions. Second, there exist many more transformation alternatives than have been considered so far. Step by step, we will integrate other transformations in our tool. Third, we plan to investigate a tighter coupling of design tool and what-if databases in analogy to the second generation of what-if interfaces [4].

9. REFERENCES