Evolutionary Database Design

Fredy Oertly
Gerald Schiller
Eidgenössische Technische Hochschule (ETH) Zürich
Institut für Informatik
CH-8092 Zürich, Switzerland

Abstract

Many tools and techniques for database design are available today. These tools and techniques, however, do not take into account the fact that a database schema, its data and its application have to evolve; i.e. go through a number of different versions during their life time. In this paper, we present an approach for evolutionary database design which tries to remedy some of the shortcomings of previous design methods. Our approach distinguishes clearly between a conceptual and a logical database design. A conceptual schema models the relevant aspects of reality. A logical schema describes the structure of the database as generic tables, and it reflects the design decisions taken to map the objects of the conceptual schema into the generic tables. In order to support this strategy by tools, it is necessary to have a version concept and a mechanism for recording design decisions which we call protocolling. The concepts described above are realized in Presto, a development environment for an evolutionary design of database applications which is currently being implemented in our research group.

1 Introduction

Database design has been researched very intensively in the last decade, and there is a well established body of knowledge on the subject available today. [Yao 85, Teory et. al. 86, Zehnder 87]. Database design provides the basis for the design of database applications [Vetter 86]. By database applications we mean applications which are data intensive and which have as their main purpose the storage and retrieval of information about a limited portion of the real world. The traditional, product oriented view of application development aims at constructing an application in a single, phased effort and it assumes that the application will remain stable during its entire life-time. This view is reflected in the traditional life cycle model of software engineering [Boehm 81]. The main drawback of this approach is that it necessitates an unpleasant and costly activity called maintenance in order to accommodate changes and refinements which occur after initial installation of an application. Unfortunately, experience shows that those maintenance activities very often consume a multitude of the resources spent for initial development of the application.

For this reason, alternative approaches are sought which try to solve this problem. One approach [Floyd 88] proposes to adopt an entirely different point of view. In this alternative view, called the process-oriented paradigm, it is not the finished product, but the dynamic process leading to a product which is the center of focus. One of its main characteristics is that computer applications are tools or working environments for people and that they are by definition subject to change; the adequacy of an application is established in processes of controlled use and subsequent revisions. A consequence of this view is that "maintenance" is meaningless in connection with software; the tasks associated with maintenance are mapped onto the development of versions. Similar ideas are proposed by [Gilb 88]; he introduces the notion of evolutionary delivery where application development starts with a nucleus of an application which is extended, refined and delivered to the customer in small increments. Evolutionary delivery is based on early, frequent iterations, and it stresses result orientation in order to achieve a manageable development process.

When trying to transfer these ideas to data engineering, a number of problems need to be solved. If no special precautions are taken, it might be necessary to restructure the database very frequently. This could be very costly or even unfeasible if an application handles large amounts of data. In other words, we are faced with the task of having to go through frequent iterations, but changes to the structure of the database are not always possible. As a way out of this dilemma, we propose a clear distinction between a conceptual schema, which reflects all the changes happening during evolution, and a logical schema which describes the structure of the database and which can be kept as stable as possible. Unlike most of the previous work done in this area, we do not try to automate the mapping from the conceptual schema to the logical one. Instead, the designer is given a selection of possible mappings to choose from. Our approach increases flexibility by allowing the designer to tailor the logical schema to current needs.

At ETH Zürich, we are developing Presto, an environment for evolutionary database and application design. It is a successor to the database design tool Gambit [Brügger et al. 85], and it implements the ideas sketched above. Presto consists of a number of cooperating tools for the design of the conceptual, the logical and the external schemata. Cooperation is facilitated by introducing a semiformal among all tools which defines a "predecessor" relationship between the tools. A central, integrated design dictionary implemented on top of the database system UDAK [Rebsamen et. al. 83] is used to store all of the design objects. Prototypes and production versions of an application can be generated from the stored design data without any manual intervention. In this paper, we discuss the database design and the evolution control aspects of our approach. We use well established database design techniques for both conceptual and logical database design, and we propose a mechanism for evolution control which is based on version concepts similar to those used in engineering applications.
The remaining sections of the paper are organized as follows: Section 2 describes the modelling techniques which are used. We will also define a conceptual and a logical database design approach and introduce an example which is used throughout the paper to demonstrate our concepts. Section 3 from the conceptual to the logical schema. It will be shown that the designer choose among different possibilities. Depending on the circumstances, the design decisions may be reversed at a later stage. In although the mapping can be automated, it is more desirable to let technique, but to integrate techniques for the conceptual and the logical design. In this section, we will first define our understanding of a conceptual and a logical schema and then briefly present the design techniques we have adopted, and we will motivate why we have chosen those techniques.

Following the ANSI/X3/SPARC framework [ANSI 86], a conceptual schema describes an application in an implementation-independent manner. The conceptual schema should facilitate communication between designers and users of an application by providing a common frame of reference. In our evolutionary approach, we postulate that the conceptual schema should at all times reflect as closely as possible those aspects of the real world which are of interest to an application. This implies that it must be possible to change the conceptual schema without a lot of technical or organizational overhead. The conceptual schema is usually considered to be the most stable part of an application. There are, however, many good reasons why a conceptual schema should be changed: the real world may change (changes of legal requirements, changes of business policies), or the model of the real world on which an application is based may change. It might be necessary to add more detail to the model because of new information needs, or previously independent applications might have to be integrated.

In the ANSI/X3/SPARC framework, the conceptual schema maps directly to the internal schema of a database. Parts of the internal schema, however, are highly dependent on the details of a specific database system. For this reason, it has been proposed to use a logical schema as an intermediate level for describing the structure of a database in an implementation-independent fashion. Deriving the internal schema from the logical schema is the task of physical database design which is not treated in this paper.

When choosing a modelling technique for the conceptual schema, the conflicting aspects of semantic richness and easy understandability have to be reconciled. It was soon recognized that the constructs of the original relational model are insufficient for expressing the semantics of an application [Codd 79]: the Entity-Relationship (ER) model [Chen 76] has proved to be a very successful technique for conceptual design, and it has been modified and extended in a variety of ways. The Logical Relational Design Methodology (LRDM) [Teorey et. al. 86] extends the basic entity-relationship model by generalizations and subset hierarchies, which improve the expressive power of LRDM. The Object Modelling Technique (OMT) as proposed by [Blaha et. al. 88] improves upon LRDM by adding object-oriented concepts to LRDM and by distinguishing among three levels of representation which are the conceptual, logical and internal schema in our terminology. We have adopted OMT as our technique for conceptual design because it has enough expressive power on the conceptual level, it can easily be mapped into the logical schema, and it uses a graphical representation which is understood by users after a short period of training.

OMT uses objects and object classes as basic modelling constructs; an object is a thing that exists and has identity. The notion of object is synonymous with entity in ER terminology. An object class is formed by a group of similar objects; object classes correspond to entity sets in ER. An object class has a number of class attributes; every attribute has a domain. Object classes are graphically represented by boxes. A relationship is a logical binding between objects, OMT distinguishes among three types of relationships with different semantics: Generalization, Aggregation and Association. Relationships are represented by lines between boxes. Special symbols at the end of a line indicate how many objects of one class relate to each object of another class. This is called the multiplicity of a relationship. Multiplicity can be exactly one (straight line without a symbol), zero or one (hollow circle) or many (solid circle); many means zero or more.

A generalization relationship (is-a-relationship) partitions a class (the superclass of this generalization) into mutually exclusive subclasses. It is represented by a line from the superclass to a branch point (triangle) and by a line from the branch point to each subclass. Inheritance holds for generalizations: every attribute of a superclass is automatically inherited by all its subclasses; this means that the same object is being represented at different levels.

An aggregation relationship (part-of-relationship) combines low-level objects (component objects) into composite objects. In the graphical representation, an arrow points towards the composite object. Aggregation may be recursive, so that it can be used to model "bill-of-materials" problems.

An association relationship relates two independent objects. Associations may have one or more properties. They correspond to relationships in the classical ER model.
As a modelling technique for the logical schema, we use the relational model [Codd 70], including the concepts entity integrity, referential integrity, primary and foreign keys [Codd 79], because the relational model has a sound theoretical basis and it is very well suited to describe a database structure in an implementation-independent manner.

As our example we describe SWEETY, an (imaginary) Swiss chocolate manufacturing enterprise. SWEETY's current order entry system is based on the conceptual data structure shown in figure 2. SWEETY manufactures different products like plain chocolate bars, chocolate fudge and chocolate biscuits; every product is fabricated by a single plant. Customers of SWEETY can place orders which consist of a number of order lines; on every line a given quantity of one product is ordered. Customers can have multiple addresses; every address is either a shipping or an invoice address. Every order is delivered to a (single) shipping address.

![Diagram of SWEETY's conceptual schema](image)

**Figure 2: Example conceptual schema**

- Customer (CustId, CustName, CreditLimit)
- Address (CustId, AddrId, AddrKind, Address)
- InvoiceAddress (CustId, AddrId, AttnOf)
- ShipToAddress (CustId, AddrId, ShippingMode)
- Plant (PlantId, Plantlocation)
- Product (ProdCode, Prodname, PlantId)
- Order (OrderId, CustId, OrderDate, OrderTotal)
- OrderLine (OrderId, LineId, Qty, ProdCode, LineTotal)

- Underlined: Primary Key
- Italics: Foreign Key

**Figure 3: Generic relations of example logical schema**

Figure 3 shows the generic relations of the logical schema, excluding domains and "nulls allowed" information. We do not comment here on the mappings which were used to derive the logical schema, as this will be done in the next section.

3 Mapping conceptual to logical data structures

The task of mapping conceptual to logical data structures has received some attention in literature. Most of the work has concentrated on mappings from the ER to the relational model [Azar & Pichat 86, Furtado et. al. 87] or on transformations in the ER model [Kontchov & Reiner 87]. In this section, we will examine mappings from the object model to relations, and we will classify these mappings according to two criteria: uniqueness of a mapping and impact on the logical schema.

We do not map individual constructs of the object model like object classes and relationships onto constructs of the relational model. Instead, we map design operations of the conceptual design (like 'Create Object Class' or 'Add Association') onto design operations of the logical design (like 'Create Table' or 'Define Foreign Key'). For initial design, the two approaches are equivalent, but for subsequent versions, we are only interested in the differences between old and new schema versions. It is in theory possible to extract the differences by comparing the two versions, but this requires complex algorithms, and in many cases there is no unique solution. Furthermore, the designer has proceeded from the old to the new version of a schema through a sequence of design operations, so it seems logical to protocol these operations and to use them as a basis for mappings. Section 4 describes in detail how protocolling is done; we will show there that every operation is split into one or more functions, but for the rest of this section we will ignore this (more technical) point and pretend that we were protocolling entire operations.

We classify mappings by two criteria:

- A mapping can be unique, or it can offer choices. For unique mappings, there is only one possible logical operation for a given conceptual operation; for mappings which offer choices, the designer can choose among different possible logical operations which represent one conceptual operation.
- A mapping can preserve the database structure, extend or modify it. A mapping which preserves database structure does not change the logical schema of the database; it may change some semantic integrity constraints, but neither the structure of the stored data nor the data itself. A mapping which extends the database structure does change the logical schema, but it only defines new structures, so that the stored data is not affected. A mapping which modifies the database structure does alter both the logical schema and the stored data; it necessitates database restructuring.

We will not describe the full set of mappings which are conceivable. Instead, we concentrate on a few interesting examples. We consider the following operations of the conceptual design:

- Create new object class;
- Create new generalization hierarchy;
- Add association relationship;
- Change multiplicity of an association.

[Bloha et. al. 88] suggest that every object class maps directly to one table. We adopt this by saying that 'Create new object class' maps to the creation of a new relation. As we provide no alternate mapping, this is an example of a mapping which is unique. It extends the database structure by adding a new table; existing data is not affected by this change.

'Create new generalization hierarchy' does create a number of subclasses to an existing (super)class. In this case, we have a choice of at least three different mappings:

1. Every subclass may map to a separate relation. This is the standard mapping suggested in LRDM and ORM. Every object is thus represented in different relations. If efficient access to an entire object is needed via the superclass relation, a type attribute (tag field) needs to be included in the superclass relation which indicates the subclass relation the object belongs to.
2. Every branch of the generalization may map to one relation; this is basically the same as 1., but all attributes which are inherited from a superclass are included in the relation which represents a subclass. Tuples of this relation contain the complete information of one object; the relation which was formerly used to represent the superclass can (after database restructuring) be dropped.

---

620
3. The entire generalization hierarchy may map into the relation which represents the superclass. This implies that we introduce attributes which are inapplicable to certain tuples, namely those which are specific to one subclass. In this case, the inclusion of a type attribute is mandatory. It is interesting to note that although mapping 1 will probably be considered as "most desirable" from an intuitive point of view, all three mappings produce relations in third normal form (using the same argument as Blaha et. al. 861); the second choice complicates updating of superclass attributes, because they are part of different relations. The third choice is probably best on efficiency, while it introduces some integrity constraints concerning inapplicable values. By defining (updatable) views for every subclass, these constraints can be handled quite elegantly. In our classification, the first two choices extend the database structure, while the third one modifies it.

As an example, we assume that customers of SWEETY can now also order products which are only traded, not manufactured by SWEETY. For the conceptual schema, this implies that a new generalization hierarchy must be created for products. The revised part of the conceptual schema is shown in Figure 4; Figure 5 shows the three possible mappings from which the designer can choose. As a next update of the conceptual schema, the association between Product and Plant will be replaced by an association between ManufacturedProduct and Plant.

![Diagram of new generalization in the example conceptual schema](image)

**Mapping 1:** a separate relation for every new subclass:
- Product (ProdCode, Prodname, Prodkind)
- ManufacturedProduct (ProdCode, PlantId)
- TradedProduct (ProdCode, supplier)

**Mapping 2:** one relation for every branch of the generalization
- TradedProduct (ProdCode, Protdane, PlantId)
- ManufacturedProduct (ProdCode, Protdname, PlantId, supplier)

**Mapping 3:** a single relation for the entire generalization hierarchy
- Product (ProdCode, Prodname, PlantId, supplier)

Figure 5: Example mappings of generalization to logical schema

In OMT association relationships are generally mapped to distinct tables (except for performance bottlenecks) although it would be possible to represent all associations with multiplicity other than many to many by a foreign key. As a reason for adopting this rule, Blaha et al. 861 argue that it is very often difficult to determine the correct multiplicity of an association and that an object should not be contaminated with information about another object. We support these arguments, but we still prefer to leave the decision up to the designer. Therefore we define a mapping which offers a choice between creating a new relationship table (extending the structure) and defining a foreign key (modifying the structure).

Adding an aggregation relationship can be treated in a way similar to adding an association. However, the special semantics of the aggregation can be exploited by including one or more instances of the component object class into the relation which represents the composite object. This technique is especially promising if a great majority of the composite objects have no more than one component.

Date gives an example of this kind in his textbook (Chapter 17 of [Date 861]) where 99% of all customers have a single ship-to address and where it is favourable to model this first address as an attribute of "Customer" and to introduce a relation "Additional Ship-to address". Our approach provides a basis for systematic treatment of such "real-life optimizations".

Changing the multiplicity of an association is an example of an operation where the classification of the mapping depends on previous mappings. Imagine a one-to-many association being changed into a many-to-many association. If it is represented by a separate relation, the mapping is unique, and structure is preserved; the only consequence is the removal of a unique constraint. If the association is represented by a foreign key, the mapping is still unique, but it modifies the database structure, because a new relation has to be introduced, and the foreign key has to be moved to this new relation.

Management of SWEETY decides that certain products will in future be manufactured by different plants. This means that the many-to-one association between Product and Plant has to be changed to a many-to-many association. As it was chosen to represent the association by including PlantId as foreign key of Product, the mapping changes database structure, because a separate relation has to be introduced to represent the many-to-many association.

Besides mapping a conceptual schema to a logical one, there is a second class of operations possible in logical design: in every case where we have a mapping with different choices, the designer can change his decision. This may be very convenient for databases which go through different phases: from a collection phase where the size of the data is not yet very large and the database structure changes quite frequently to a production phase with a large amount of data and the need for a stable database structure. During the collection phase, it is desirable to match the logical schema as closely as possible to the conceptual schema, and frequent database restructuring is not a problem; during production phase, the logical schema should remain stable, as database restructuring is very expensive or even impossible.

4 Evolution Control

Conceptual and logical schema are subject to changes. Although during database design efforts are made to model schemata in a stable way, changes of the real world may affect existing schemata. For productive use, only the current version of a schema is relevant; individual design steps are (from this point of view) not of any interest. However, Presto offers tools for conceptual and logical design which need to know how a new version of a schema was reached; changes of the conceptual schema must be reflected in the logical schema, and changes of the logical schema return vital information for reorganizing the structure of the production database. This implies that design operations must be protocolling for internal use. Protocolling is the topic of this section.

We use a version concept similar to the one described in [Dittrich/Lorie 881]; however, we keep versions of entire schemata, not of individual design objects. A version describes one evolution step of an application, i.e. in our case of the conceptual and logical schema: as we have seen in section 3, changing the conceptual schema may invoke changes of the logical schema; the result (i.e. new conceptual schema and new logical schema) defines a new version. We distinguish among three kinds of versions: "actual versions", "production versions" and "intermediate versions". The actual version is the evolution step which is actually being processed, or which has been processed last. A production version is an evolution step which has been generated and delivered to the user. An intermediate version is an evolution step which is used as basis for further versions; we will...
Different kinds of versions

Versions can be saved on external storage media: this action sets a checkpoint flag. "Long transactions" can be modelled by using such checkpoints. Global rollbacks of the design data can be accomplished by reloading a saved version. A version tree is defined by the sequence of evolution steps. Productive versions must have the checkpoint flag set (this is assured by the system): they are "frozen" in the sense of [Dittrich/Lorie 88]. This implies that reloading a saved version located before a production version is not allowed, since the production database could not be reorganized in all cases.

The conceptual and logical schemata are processed by two different design tools; every tool offers a set of design operations. Operations are split into functions: one function updates only one object of the design dictionary. An operation is valid only if all its functions are executed correctly (short transactions). Since operations are too complex to be protocolling, a single function represents one protocol unit. Obtaining consistent protocolling is guaranteed by providing one common interface to the design dictionary. This interface accepts function calls as input.

A function call triggers two different actions: update of a design object and creation of a "change entry" (protocolling of the function name and its parameters). Change entries of one tool within one version form a "change list"; all change lists of one tool (i.e. the change history over all versions) define the "change protocol". Protocols of all tools are part of the design dictionary; in this way consistency between design data and protocol entries is guaranteed by the transaction mechanism of the DBMS used for the implementation of the design dictionary.

In the introduction we have seen that there is a "predecessor" relationship between tools which defines a semiorder. Using centralized protocolling of functions thus requires some assertions concerning concurrent operation of tools.

Under what conditions can different tools be used concurrently? There are at least three possibilities for solving the cooperation problem. **Parallel cooperation:** no restrictions on cooperation; **sequential cooperation:** a dependent tool may operate only after the predecessor tool has completed all its changes; **locking cooperation:** parallel cooperation of tools with locking of design objects. The overhead for the implementation of parallel cooperation is high, and it requires
routines for managing multiple copies of objects and merging them. This cannot always be done unambiguously. For locking cooperation, a locking mechanism on object level is needed. In both cases a tool can process a schema at any time. Sequential cooperation does not require such mechanisms, because the view that a tool has of another tool’s changes is always up to date. The disadvantage of this cooperation is that a tool is allowed to process its schema only if all predecessor tools have completed their processing. In most cases this is not a serious restriction, because conceptual design of one version is made before logical design. With sequential cooperation, a tool is allowed to process its schema only once in every version. If looping back is necessary, a new version has to be created; the actual version probably will not be turned into a production version and remains an intermediate version; this explains the need for intermediate versions. In Presto sequential cooperation is used.

An evolution step transforms the schemata from one consistent state to another. Therefore the design data must be consistent for the actual version to be completed. We distinguish between local consistency of a tool’s design data and global consistency of the entire design. Local consistency can be checked by the tools themselves, e.g. every object class in a conceptual schema must have a unique name. Global consistency of the application design is concerned with the cooperation of different tools. When trying to complete the actual version the protocolling information is checked to find out if all change entries have been processed.

Figure 9: Protocolling Interface

In order to keep track of change entries, every protocol is referenced by one “write pointer” and some “read pointers”. Every pointer has a current position. The write pointer is updated automatically when a new change entry is protocollated. Every design tool has one read pointer to the protocol of all its predecessor tools, different tools have different read pointers. This pointer can be moved explicitly to the next change entry when desired. Multiple reading of the same entry is possible. This gives more flexibility for processing change entries. Analyzing the change entries of a predecessor tool can be done in an easy way:

While change entries are available:
1. position the read pointer to the next change entry;
2. read change entry;
3. analyse entry and, if necessary, react to the change entry, in an appropriate way.

Global consistency is assured as soon as every read pointer refers to the same change entry as the write pointer of the referenced tool. Local consistency constraints of every tool guarantee that change entries of predecessor tools are processed correctly.

5 Conclusions

In this paper, we have proposed an approach for database design which relies on well-known techniques for the conceptual and the logical database design and which tries to cope with the necessity to evolve database applications including their data. After motivating the need for evolution, we have presented a way of mapping a conceptual to a logical schema by taking the design operations, not the design objects as starting points; it allows choices for certain mappings. Finally, we have described a protocolling mechanism which can record design operations and which supports controlled cooperation of different tools. These concepts are part of Presto, a development environment for interactive database applications which we are currently working on in our research group. While implementing Presto, the ideas described in this paper have been applied (different schemata for the design of the data dictionary structure, cooperation concept), and this proved to be successful.
We are at the moment investigating a number of open questions. First of all, the mapping mechanism is extended to the design of external schemata and of forms. In addition, we are studying algorithms for database reorganization in the case of structure-modifying mappings.

Acknowledgments

The authors wish to acknowledge the work of Peter Janes, Martin Bär and the entire Presto team who provided feedback on the ideas presented here and who took part in the implementation of Presto. Thanks are also due to R.W. Marti, C. Stutz, A. Walchli and the anonymous referees for their helpful comments on draft versions of this paper.

References

[ANSI 86]

[Azar & Pichat 86]

[Blaha et al. 88]

[Boehm 81]

[Bragger et al. 85]

[Chen 76]

[Codd 70]

[Codd 79]

[Date 86]

[Dittrich & Lorie 88]

[Floyd 88]

[Furtado et. al. 87]

[Gilb 88]

[Rebsamen 83]

[Rebsamen et. al. 83]

[Rosenthal & Reiner 87]

[Teeby et. al. 86]

[Vetter 87]

[Yao 85]

[Zehnder 87]