A Macro Approach to Relational Database Modeling

Douglas M. Kline
Information Systems and Operations Management, UNC Wilmington
Wilmington, NC 28403-5611 USA

Charlene Riggle
Information Systems and Decision Sciences, University of South Florida
5700 North Tamiami Trail, Sarasota, FL 34243-2197

Abstract

A new data modeling process is presented that addresses some of the weaknesses of the traditional normalization-driven modeling process. Current approaches generally begin with forms or reports for a particular system as rough entities, then taking these entities through a normalization process, inspecting attribute functional dependencies. The proposed approach is entity-oriented, focusing more on fully developing the data entities and their relationships than on scrutinizing functional dependencies among attributes. We argue that this macro approach should result in more communicative models that are more flexible, being less specific to particular applications. We end with a discussion of other topics that arise in relational data modeling.

Keywords: data modeling, relational model, modeling processes

1. INTRODUCTION

Relational databases are the foundation of many information systems today. Compared to other forms of databases such as hierarchical, network, and object-oriented, relational databases dominate the market. The ability to have multiple divisions or departments share the same data is extremely powerful. Many of today's systems rely heavily on relational database technology to bring together data from disparate "information silos" and provide it as an enterprise-wide resource.

Relational database designs are fundamental to building systems today, due to the many demands placed on data management systems. Databases must support many different applications, so the data must be very flexible. Since bogus data entered by one system can cause problems in another system that uses a common database, the data integrity must be maintained at a high level. Since databases tend to remain as new systems are added and existing systems are customized or replaced, the design must be robust to unforeseen demands. In short, poor database design can lead to brittle systems, poor access to information, and years of effort making systems interface with the poor database design.

Relational database design is a high-level skill that is critical to building quality systems. While programming has become a trade and is being moved off-shore, design is likely to stay on-shore and be in demand (Coy, 2004, and Baker & Kripalani, 2004). Despite the flowchart-nature of traditional normalization processes, relational modeling is not an exact science. Good designers need to see the big picture, operating above the level of any particular system.

Unfortunately, relational data modeling is not an exact science, but rather an art (Frost, 1997). We cannot claim that our proposed approach results in quantitatively measurable better data models. We attempt to point out the weaknesses of the normalization-based design approach, and the strengths of the proposed macro approach.

Considering the importance of relational databases in today's systems, the relational modeling process and how it is taught deserve attention. In this paper, we will identify some of the shortcomings of the traditional normalization-oriented modeling process. First, we review some of the goals of the relational model. Then the traditional modeling approach is described and shortcomings pointed out. Then we will present a macro-level approach to data modeling that addresses some of the shortcomings of the traditional method.

2. GOALS OF THE RELATIONAL DATABASE MODEL

In this section, we review some of the goals of the relational database model. Surprisingly, the driving forces that motivated the creation of the relational model still exist today, despite technical advances.

Codd's main arguments (1970) are increasingly relevant in today's environment. Modern ERP systems are essentially "large shared data banks", as in Codd's seminal paper title. Codd's main arguments in his first work, as they pertain to this paper, were that 1) programs should not depend on data representation 2) there is a preferred data representation, i.e., normal form. Codd's second work (1982) stressed putting end-users in direct contact with the information they need, and identified "communicability" as a primary motivation for relational databases.

Program-data independence

One of the main goals of the relational model was to allow for program-data independence. Traditionally, this has meant that the programmer need not be familiar with the physical storage mechanism and physical storage model in order to write the program, i.e., the program need not know the physical data model. Reference of data values by table name, column name and primary key value allowed programs to be written without knowledge of the physical data storage details.

The program-data independence concept can be extended to the concept of decoupling the logical data design from any particular application. Today's systems commonly share a database, so a database whose logical model is convenient for one application can cause serious problems for other applications. The logical data model needs to be flexible, and readily support new systems or unforeseen features. In particular, the database model should not impose constraints on application designs.

Data integrity

Today's databases often serve as the "systems integration hub". In other words, it is where systems must meet, sharing data across organizational divisions, hardware and operating system platforms, and application programs. In this context, data integrity is critical. Erroneous or invalid data submitted from one system can decrease the value of information produced by other systems, and in the worst case can cause outright system crashes. By enforcing entity integrity, referential integrity, and performing common data validations centrally (in the RDBMS), the integrity of the data is maintained. This reduces or eliminates the well-known insert, update, and delete anomalies, as well as orphan and widow records. In short, the internal consistency of the data is maintained.

Communicability

The relational database model is table-oriented. Because tables are easily understood by technical and non-technical people, the relational model is more understandable than other forms of data models. This is important because many of the spectacular IT project failures have been caused by communication problems, not technical problems. A clear, understandable model of the organization's data resources is essential to making the best use of those resources.

3. THE TRADITIONAL MICRO APPROACH TO DATABASE MODELING

The clear arguments of Codd and others are quite convincing and not easily arguable. However, tight deadlines, scant resources, and a focus on the program being developed today can lead to a myopic view that hurts companies in the long run. Here are some typical quotes from IT professionals:

1. "We couldn't enter a record into the table, so we deleted all the relationships."
2. "We just put everything in one big table – it's simpler that way."
3. "We de-normalized for speed."

Each of the above quotes can be attributed to poor education about the relational database model. All of the above actions, although expedient in their current situation, will lead to dirty data, brittle systems with short useful lifetimes, and data that cannot be managed to create its full potential value.

Nearly every computer science and information systems program has a course dedicated to databases. So why are the above actions so prevalent in industry? We suggest that the current process of data modeling is flawed. Here is a typical process for traditional normalization data modeling.

1. Specify the requirements of a system / application / program
2. Specify the forms and reports for the system
3. Normalize the forms and reports to 3NF

The third step above involves transforming the data model through a series of normal forms. To transform a data model first normal form, for instance, all first normal form violations are identified and removed. Once all first normal form violations are remedied, the model is said to be in first normal form, and the modeler begins seeking out second normal form violations. The model is successively taken through the normal forms, stopping at third normal form, as a rule-of-thumb, even though at least five normal forms have been identified (Kent, 1983). The following sections point out the shortcomings of this approach.

Database design begins with program design

Most database designs are driven from the requirements for a single application. This inevitably leads to a database design that is biased toward the project at hand. Using a construction analogy, the house is designed before the foundation design or site plan. When the foundation is meant to support a single house, this is a natural analogy. However, databases are usually meant to support multiple systems. A more accurate analogy

might be that the foundation will have to support a house, future additions to the house, and most likely a number of other buildings to be specified at a later time.

Budget and timeline constraints also contribute to the single-application bias. Experienced project managers tightly manage the scope of their project, so the database design is constrained to the specific needs of the current project. The system is considered to be the focus of the project, so the database is merely a supporting technology, which may lead to insufficient time, expertise, and effort spent on the database design. This is ironic, since the data will generally far outlive the system.

Database design driven from forms and reports

Many professionals and textbook authors suggest that the forms and reports generated from a requirements analysis are a good basis from which to develop a data model. Certainly this is a good starting point, since the forms and reports will generate a list of attributes that must eventually be stored in the database. The danger lies in considering the forms and reports to be drafts of the final entities in the data model. To use an automotive analogy, the process starts with a "Yugo" and attempts to transform it into a "Cadillac", rather than designing the Cadillac from the ground up. As a result the data model will always have some bias toward the application that drove its design, and will thus prove problematic in supporting new features and future systems. This specific-program-driven approach has the real possibility of leading to data models that favor an "access path", as described by Codd (1970).

To a large extent the classic normalization process is to blame. It was designed as a method for improving traditional flat-file systems; the goal was improvement. However, normalization has never guaranteed high-quality data models, only the improvement of poor ones. Certainly eliminating redundancy can reduce systematic anomalies, but may not improve the model's ability to support new systems and features. The conventional practitioner wisdom of "3NF is good enough" also suggests that eliminating easily identifiable normal form violations will produce a high-quality data model. However, the lack of bad characteristics does not necessarily guarantee good characteristics – the resulting data model may or may not be high quality.

Another danger is limiting the design to supporting only data necessary to deliver the required reports. At the time of requirements analysis for a particular system, only a small number of reports are known. This is only a small subset of the future ad-hoc reports that will be needed. So basing the data model on this small subset of requirements will lead to a data model that does not support additional requirements.

Attribute-oriented

The traditional normalization process operates at the attribute level. Attribute dependencies are examined and used to identify normal form violations to be resolved. This "tree rather than forest" approach can lead to myopic designs that can have higher-level design problems while satisfying the normal forms. In a sense, a focus on attribute-level problems is treating the symptoms of a poor design, rather than addressing the overall design. The classic symptoms are repeating groups, multi-valued attributes, transitive dependencies, etc. However, none of these symptoms address large issues such as:

- Should two entities be combined?
- Should a recursive relationship be used?
- What is a good name for this entity?
- Would a many-to-many relationship increase the flexibility of the model?

Loose Diagramming

Some academics and practitioners confuse *modeling* with *diagramming*. In general, producing a diagram documents a model. Diagramming languages such as Entity-Relationship Diagramming and Unified Modeling Language serve this purpose nicely. However, diagramming languages tend to be general tools, and have been specifically designed to be very flexible. Unfortunately, these languages are equally adept at diagramming good and *bad* designs.

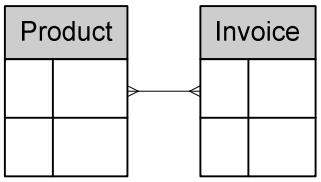

Figure 1:

All the well-known normal form violations can be readily diagrammed in UML or E-R Diagrams. Clearly, knowing a diagramming notation is not equivalent to knowing how to model data. The Crow's Foot notation (Watson, 2003) is typically better at enforcing well-formed relational models. However, even this notation is frequently loosened to allow a single relationship line with a crow's foot at each end. Using loose diagramming notations can make spotting poor models difficult, and can actually promote poor models.

As an example, consider the common method of representing a many-to-many relationship with a single line with crow's feet on each end. When modeling an Invoice and a Product (see Figure 1) entity with an N:M relationship, this notation hides the fact that an important entity, Invoice Line, is missing from the model. As a result, the Invoice Line entity may never be identified, developed, and documented. No one denies that the N:M diamond will eventually become a table in the implemented database. However, the Invoice Line entity becomes an implementation detail, rather than important entity that deserves developing.

4. MACRO APPROACH TO DATABASE MODELING

In this section, we present a "macro" approach to database modeling that focuses on modeling the data rather than creating a database to support an application. In general, the approach tends to focus on *entities*, rather than *attributes*. It should also be noted that the modeling process is not viewed as a supporting activity in a particular systems development project. Rather, the data modeling process should be an activity that is a worthwhile undertaking of itself, and meant to support multiple systems.

The Process

Identify strong, independent entities: Through standard systems analysis efforts, entities that are central to the organization should be identified. Entities are typically nouns that appear in documentation. Identifying entities is not typically a problem – identifying the strong ones can be a problem. Strong entities[i] are important to the organization, and will have a special importance in the model. Strong entities tend to get mentioned over and over during conversations, and appear repeatedly in documentation. Strong entities are, in many cases, physical entities that are tangible in the real world. They furthermore tend to be clear and understandable to laymen who do not have a great deal of domain-specific knowledge. The strong entities will not be exclusive to a single feature in a particular system, but will be used across systems and features.

The goal in this step is to identify the core entities that are important to the organization. Typically, this is a relatively small number, perhaps six or eight. These entities are likely to be related to many other entities, and take on the independent role in these relationships. Identifying these core entities will require acquiring domain expertise and a high-level view of the organization.

Choose a single entity to develop: From the entities identified, choose a single entity to develop. In the beginning stages of modeling, the choice of the entity is very important. A good choice will ease the modeling effort, while a poor choice will make it more difficult and confuse the modeling process.

In the beginning stages of modeling, the chosen entity should be a strong entity that is not tied to a single feature in a single system. The entity should have a clear understandable name, in singular form. Naming is critically important in a database model. First, SQL is exclusively based on naming, so the chosen entity names will be used for years to come. The name will either confuse or clarify the model for all those years. Second, changing the name of an entity later can be extremely confusing to all involved. Finally, a good name goes a long way towards understanding. Compare the name "Invoice Line" to "Invoice-Product". The name "Invoice Line" is immediately understandable by anyone familiar with an invoice. The nature of "Invoice-Product" might be inferable by database professionals, but is not necessarily clear to others.

In the later stages of modeling, weaker, more peripheral entities will be developed. These weaker entities are typically arrived at in a derivative manner, i.e., they are derived from the strong, central entities. It is difficult to derive a strong entity from a weak entity. Identifying and developing strong entities early can ease the modeling effort, and help to maintain a good conceptualization of the model.

Add attributes to the entity: To further develop and clarify the chosen entity, attributes should be added. This is an important step, as the name alone is not enough to clearly understand the nature of the entity. As an example, consider entities named Order, Sale, and Invoice. At first glance, it may appear that these are a single entity with various names for the entity, and should be combined into a single entity. However, they may actually represent different stages of a transaction, and be very different entities. Adding attributes will clarify the nature of the entity.

It is not uncommon at this stage to realize that a single entity that is being developed has attributes that indicate that the single entity is actually several entities. Using the example above, one might start with an Order entity, and add PaymentMethod and PurchaseOrderNumber as attributes. Assuming that the model needs to represent the various stages of the revenue cycle as separate entities, the data modeler should realize that the PurchaseOrderNumber belongs in an entity that represents the initiation of the purchase, i.e., Order, and the PaymentMethod belongs in an entity that represents the end of the purchase, i.e., Receipt. As with entities, the naming of attributes is very important for clarity.

Choose an identifier for the entity: Choosing an identifier further clarifies the nature of the entity, for both the data modeler and for developers of systems that use the data. Many developers skip this decision by using synthetic, or non-meaningful, identifiers for every entity. There are arguments for and against this strategy. Regardless of whether a synthetic identifier is used or not, all alternate keys must be documented so that unique indexes can be set at implementation time. It is not uncommon to have several alternate keys for entities, and identifying them is helpful in understanding the entity.

Explore relationships between entities in a pairwise manner: After the entities have been developed, the relationships between entities can be explored. This is a much easier task with the entities fully developed. Exploring relationships can be quite confusing with a cloudy understanding of the entities involved. Relationships can be explored in the usual pair-wise manner. In addition to identifying the relationship and cardinalities of the entities involved, it is helpful to document if there is an existence dependency.

Develop any associative entities: Associate entities may be identified from the previous step if there is a many-to-many relationship. These associative entities should be developed in the same manner as other entities. In fact, associative entities can sometimes be the most important entities in a data model, in terms of understanding the nature of the data, and how business entities relate. Associative naming should be avoided. Instead, data modelers should strive for a clear, concise name, as would be expected for a strong entity.

As an example, consider the entities Student and Class, with an associative entity representing a student that is enrolled in the class. Associative naming conventions might suggest the entity be named Student-Class, as in Figure 2. Does Student-Class represent that a student has shown interest in the class? Does it represent that a a student teacher is observing a class? Does it the entity represent degree requirements? However, a name such as ClassMember, as in Figure 3, is much clearer and understandable. It means that the student is a member of the class. Furthermore, with a clear name, the entity appears more significant, substantial, and worthy of further development.

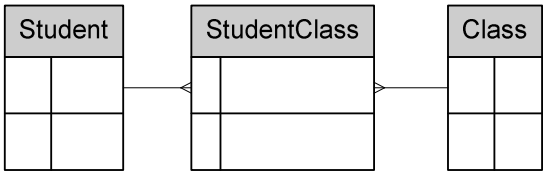

Figure 2

Data modelers should further strive to find attributes that clearly belong in the associative entity, and not in the independent entities involved. This further clarifies the associative entity and makes it more significant. In the above example, an attribute such as EnrollmentDate and Auditing (true or false) clearly belong in ClassMember, and not in Student or Class. Suddenly, an associative entity that is weak in relation to both Student Class becomes an important entity, and may actually participate as a strong entity in other relationships, as in Figure 4.

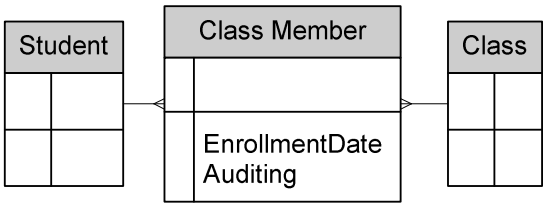

Figure 3

If associative entities are fully developed, it is quite common to have an associative/weak entity in one relationship become a strong/independent entity in relationship to another entity. Continuing the above example, a ClassMember might relate to an Grade entity in a one-to-many manner, with existence dependence (an AssignmentGrade can't exist without a ClassMember). If the associative entity hadn't been fully developed and understood, this relationship would have been more difficult to discover, i.e., a relationship involving a relationship.

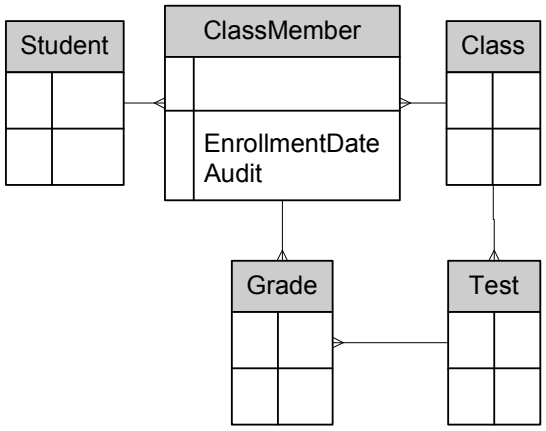

Figure 4

Add weak entities to support features: Once the core entities have been developed, entities that support peripheral system features can be added. At this point it is a good sign if adding support for peripheral features does not significantly change the data model. This indicates that the data model is robust to added features or new uses. By "significantly change", we mean changing the nature of the existing entities or relationships. Adding attributes to existing entities, or adding entities does not significantly change the existing data model.

Other helpful concepts

The above section describes an entity-oriented process that approaches data modeling from the top down. However, there are several other concepts to keep in mind that are very helpful in the modeling process.

Focus: Working on a large data model that must support many features and multiple systems for the foreseeable future is a daunting task. It is often overwhelming to consider all entities for a data model at once. To deal with the complexity, it is helpful to focus on one to several entities at a time. Focus on a single entity as the entity is being developed, then focus on a small group of entities as the relationships are developed.

Associative entities are entities: Associative entities are usually discovered when a many-to-many relationship between existing entities is discovered. They are sometimes treated with less respect and attention than entities that are discovered in other manners. As a result, associative entities are not even considered entities, and are diagrammed in a different way than other entities. This can lead to serious deficiencies in data models, since the entity is not explored or modeled explicitly, but is left as an implementation detail.

A full development of associative entities almost always results in the realization that it is a "true entity". Asso-

ciative entities almost always have multiple attributes and clearly represent an identifiable business entity. In many cases, these are very important to the business and to accurately modeling the business.

Entity terminology: Because the described approach is entity-oriented, it is important to have terminology that describes entities. This terminology is not new, and has been used before, but takes on new importance in an entity-oriented approach to data modeling.

The notion of weak/strong, or dependent/independent entities is useful on two contexts. First, when considering a pair of entities and how they relate, it is helpful to recognize that one of the entities might be dependent on the other. Sometimes, this is existence dependence, e.g., a Room cannot exist without a Building. Sometimes, this is identity dependence, e.g., the Room primary key has the Building primary key in it. The second context involves considering an entity in the overall data model. For instance, an auto-dealer's information system revolves around automobiles, and thus the Automobile is likely to be a strong, independent entity in the data model. This manifests itself in the data model as the Automobile entity is involved in many relationships, typically on the "one-side", and there may be tree, or hierarchical, structures emanating from the Automobile entity.

"Odd" model forms: Several model forms are not commonly given adequate recognition in the normalization process. Neither recursive relationships nor one-to-one relationships are recognized at all by the normalization process. These are sometimes presented as odd or rare model forms that are perhaps of interest in only academic circles. A sampling of data models (Silverston, 2001) and our personal experience suggests that these are not at all rare, and arise in nearly every data model. Not only are these model forms common, but they serve very important roles in data models. A one-to-one relationship (particularly when describing subtypes) allows for abstraction, generalization, and re-use of source code. Subtypes are quite necessary to reduce redundancy, resolve transitive dependencies, and for enforcing relationships. Recursive relationships are also quite common, and cannot be easily modeled in any other way.

De-normalizing: Sometimes data models that rigorously enforce data consistency in a high-quality data model are relaxed in implementation, or "de-normalized". The ostensible motivations are improved performance and ease-of-use. This is regrettable, because the costs of inconsistent data and inflexible databases are real, if not easily quantified. Both of the motivations are questionable. Performance can be improved in many ways before resorting to de-normalization: reformulating the sql, using a stored procedure, indexing, caching, hardware solutions, etc. In most cases, the decision to de-normalize occurs well before the system is in production, and thus the need for de-normalizing was never truly evaluated in production. In effect, practitioners are paying in data quality and inflexibility for a performance boost they may never get.

The second motivation for de-normalizing is ease-of-use, i.e., the data is stored in the form the end users want, so the data need not be processed to generate the report. This clearly violates the program-data independence objective of the relational model, and reduces the flexibility of the data model. A database designed in this manner will only be useful for the system it was designed for, and will resist any attempts to add features or functionality. Furthermore, any current or future representation of the data is easily generated from a high-quality data model using SQL.

5. CONCLUSION

Relational database design is fundamentally important to systems that share information. Because of the foundational role that databases play in today's systems, it is vital that we teach relational database design well. The traditional, normalization-oriented design process that is driven from a particular application's needs can produce data models that are lacking in flexibility and clarity.

We presented a macro approach to data modeling that addresses some of the weaknesses of the traditional approach. The macro approach focuses on entities rather than attributes, accommodates more complex model forms, and produces models that are not tied to a particular application. Students who learn this approach will be learn to design databases that minimize program-data dependence, maintain data integrity, and communicate clearly the available data resources.

6. BIBLIOGRAPHY

Baker, Stephen and Manjeet Kripalani (2004) Software, Business Week, March 1, 2004.

Codd, E. F (1970) A Relational Model of Data for Large Shared Data Banks, Communications of the ACM, 13(6), pp. 377-387.

Codd , E.F. (1982) Relational database: A practical foundation for productivity, Communications of the ACM, 25(2), pp. 109-117.

Coy, Peter (2004) The Future of Work, Business Week, March 24, 2004.

Date, C.J. (1995) An introduction to database systems, 6th Edn., Reading, MA:Addison-Wesley.

Frost, Raymond D. (1997) Teaching Design to Solve Business Problems, Journal of Database Management 8(3), pp. 37-38.

Kendall, Kenneth, and Julie Kendall (2001) Systems Analysis and Design, 5th Edn., Prentice Hall, New Jersey.

Kent, William (1983) A Simple Guide to Five Normal Forms in Relational Database Theory, Communications of the ACM, 26(2), pp. 120-125.

Kroenke, David M. (2004) Database Processing, 9th Edn., New Jersey: Prentice Hall.

Russell, Tom, and Rob Armstrong(2002) 13 Reasons why normalized tables help your business, Database Administrator, April 20, 2002.

Silverston, Len (2001) The Data Model Resource Book, Revised Edn.: Volume 1: A Library of Universal Data Models for All Enterprises, NY:New York, Wiley.

Watson, Richard T. (2003) Data Management: Databases and Organizations, 4th edn., John Wiley & Sons, New York.

---

[i] This is a loose usage of "strong", which is sometimes defined rigorously in modeling texts. Our usage focuses more on importance to the organization, than strict diagramming notations. However, the two usages commonly coincide; entities important to an organization usually participate as strong entities in relationships.