1

OF A
THREE-PART
SERIES

Database Modeling and Design in Information Systems

By Josh Jones and Eric Johnson

CA ERwin

INTRODUCTION

Database modeling has long been an overlooked process in information technology. Too often the feeling is that while application code needs to be tested and refined, you can just toss your data into any database and all will be well. This couldn't be further from the truth. Proper data models will not only aid application performance by speeding up data retrieval and data writing, they ensure scalability and flexibility for future development. These days, seconds can mean money and users that have to wait for slow applications tend to look for alternative solutions. You need to invest time and resources into your data model if your applications are going to be the best they can be. In this whitepaper, we will look at exactly what this thing called data modeling is, why it's important, and the important concepts and practices behind it.

TABLE OF CONTENTS

THE ROLE OF DATABASES IN APPLICATION DEVELOPMENT

Databases play a huge role in application development. Most applications require that some form of data be stored for future use. When it comes time to save this data, the logical choice in most cases is a relational database. There are other options—flat files, XML, persisted record sets, or even custom file formats—but none of these is robust or safe as a database. Databases allow you to analyze the information you have collected through tools such as reports and data warehouses. Additionally, most Relational Database Management Systems (RDBMSs) allow you to backup your database in the event of a failure, as well as offering you methods to implement high availability solutions to minimize downtime due to failure. These benefits alone should push you towards using a database as your solution for information storage and management when you develop applications.

DATA MODELING BASICS

Data modeling is really simple on the surface; it's the process of mapping real world information to logical representations of that data. In other words, how do I store information about my customers in a data model? A lot of this is based on the judgment of the person building the model, but there are a number of key concepts that will help a data modeler make good decisions. Also, when modeling, you need to think about data from a logical standpoint without being concerned with how tables and columns will look in your database. This is called Logical Modeling, where the only goal is to build a model that represents real world objects. Once your logical model is complete you can start thinking about your physical database model. Keeping the logical and physical model separate will help ensure that you build a solid database.

DESIGNING A DATABASE

During application development, there is usually a phase very early in the project where designers and/or application developers spend a great deal of time gathering requirements for the application that will be built. This information is usually gathered through a sequence of interviews with users and managers, as well as observing the existing system (even if it is a purely manual system). The end result will be a fairly detailed set of requirements, up to and including use cases, system diagrams, and mock ups of the application interface. Once all of the key players in the project are satisfied, application development begins to take place. One of the first things that most application developers need to do is create a repository for the data that the new application will be working with; in other words, the application's database.

In many cases, the database is developed to meet the physical requirements of the application interface. That is, once there is a mockup of how the application will accept and display data, the application developer can then create an empty database in the RDBMS. Once there is a table (or set of tables) that completely define all of the data from the interface, the application developer can then start storing data, and writing code for the application to communicate with the newly built database.

While this approach is far from perfect, it can work, at least in the short term. If done very carefully, the resulting database may even function, provided no changes are needed and a very small amount of data is actually stored in the database. However, databases that have been designed this way will inevitably experience a sever scalability problem, and are often extremely difficult to modify as time goes on. We all know that applications will eventually be modified, and even rewritten, to introduce new functionality (or retire older functionality); however, we usually want to retain our old data. This means we end up doing piecemeal design work in the "old" database to add the new functionality without risking any data loss. This will degrade performance, as we are often adding new database structures (tables and views) as well as complicated Structured Query Language (SQL) logic to present the old and new data in the same interface. Building a proper logical data model prior to doing any actual database development will help eliminate these problems, as well as reveal any missing requirements in the data layer before a single line of code is written.

DATA MODELS VERSUS DATABASES

Data models are created early in the design phase, usually towards the end of the requirements gathering phase of a project. Once all of the user interviews and observations have been completed, a data model is created to document the data that the application manages. This model will present, logically, all of the pieces of information needed by that application. It will describe how different pieces of data relate to one another. The model also provides a non-technical users view of data, and can be very useful in gaining user acceptance of the overall application design.

Strictly speaking, a relational database is an organized collection of tables that store data. More practically, a database is the collection of tables, views, and stored procedures (depending on RDBMS) that store and access data. These structures are defined using the language of the RDBMS, which is usually some form of SQL. The RDBMS stores that data in files in the operating system, and has functionality in place to manage the files, manage security, and tune performance of the queries used to manipulate the data. From an application standpoint, the database is where the data goes to and comes from.

The data model, unlike a database, does not represent

physical storage of data. Whereas a database defines how data is stored, what actual relationships are enforced in the manipulation of that data, and allows for programmatic access to the data, a data model simply states what data exists and how the different bits of information relate to one another. When properly designed, a data model then becomes the logical blueprint for a physical database to be developed. Because of this relationship, data models are often required to be platform agnostic; a data model can be used to create a physical database in Oracle 10g, SQL Server 2005, or MySQL. However, that's not to say that data modeling lives in ignorance of the RDBMS which will eventually be used. There are certain situations where having a forward thinking view into which RDBMS will be used for your database can have an impact on the data modeling process.

WHY DATA MODELING IS IMPORTANT

Why are good data models important? First, a good data model will provide better performance, particularly for online transactional processing systems (OLTP). Secondly, a solid data model will ensure a well documented, scalable database. Failure to develop a good model will cause you pain when it comes time to add new data to your data model, and eventually your physical database will suffer.

PERFORMANCE

Proper data modeling allows your RDBMS to perform well. Primarily, following standard data modeling rules will help you eliminate data anomalies, such as duplicating data, which will eventually prevent the need to embed additional logic in the application to handle those data anomalies.

Additionally, when data is stored in a structured format, the query engine can find and retrieve it faster than if it were stored in a flat file or a bad structure. This leads to improved performance of your applications and/or reports.

DATA ACCESSIBILITY

A good data model makes the data in the database easy to understand. If you have well defined entities and tables that categorize the data you are working with, then analyzing your data through reporting or data warehousing will be considerably easier. If you can't find it, you can't use it.

One major benefit of a solid, well thought out data model is realized during the development of the physical database. Particularly in large projects, where multiple database developers may be engaged, a comprehensive data model will allow different developers to work on different subsections of the database while still understanding the entire database. This can prevent duplicate work, allow more flexibility in development assignments, and provide a faster, more cohesive development phase, at least where the database is concerned.

BUILDING DATA MODELS

In order to actually start building models, you will need to understand entities, attributes, and relationships. These concepts are the building blocks on which data models are built. We will take a close look at each of these structures and how you can use them to build a logical model.

ENTITIES

Entities are the primary object used in a data model. An *entity* is a representation, or grouping, of a single type of data. Customers, orders, sales, and products are all examples of entities. Logically, each entity is a collection of *instances* of that type of data. An *instance* is a single occurrence of that piece of data. To clarify, think of the relationship of rows to tables in a physical database; rows are to tables what instances are to entities.

Note that while there are similarities between tables/rows and entities/instances, do not be tempted to always think of entities as tables. Remember that an entity is a logical representation of a type of data, and a table is a physical storage structure for data. In the end, one entity may be physically implemented as one or more tables, or conversely, a single physical table may account for multiple entities (though the latter is truly rare).

When evaluating your requirements for a data model and trying to determine which pieces of information are entities, look for certain keywords such as customer, vendors, products, etc. Usually, these are nouns mentioned in interviews and notes taken during observation. A good rule of thumb is that nouns will equal entities. Whenever you see a certain type of data being referenced as an item or object, those are usually your entities as well.

ATTRIBUTES

An *attribute* is a descriptor about the instances in an entity. An entity usually has several attributes. For example, suppose you have created an entity named DogCollars, which represents the dog collars that a pet store sells. For each dog collar, there are a set of physical attributes: color, length, width, etc. These are also the attributes for the entity: Color, Length, Width, Brand, etc. When constructing a data model, each entity will have a collection of attributes defined that describe the data stored in that entity.

One very specific problem with attributes tends to be deciding which entity a given attribute actually "belongs" to. This is primarily a mistake made when designing an application (and its data model/database) based on a physical process (something that doesn't currently have any applications involved). In the classic retail example, customer information such as phone numbers and addresses are frequently stored with each order that a customer makes. When creating a data model, it may be easy to convert the physical

documentation (such as a spreadsheet) directly to entities. In the order example, the Orders entity might have customer attributes attached to it. Logically, this is incorrect. An order is a logical piece of data, or object, and a customer is a logical object. Phone number and address attributes should be attached to the customer, not the order.

RELATIONSHIP

Relational databases are based on the concept that data inside that database references, or relates, to other pieces of data inside that same database. Using a database without defined relationships is similar to dumping all of your receipts, paychecks, and financial statements into a large trash bag for storage. Without knowing how all of those pieces of paper relate to one another, how would you manage your finances, or do your taxes? Not having any relationships in your data model is similar; the different entities and their attributes must relate to other entities and attributes, and we need to document what those relationships are.

There are three basic types of logical relationships in a data model: one-to-one, one-to-many, and many-to-many. Each relationship describes a different manner in which two pieces of data relate to one another.

One-to-One Relationships

One-to-one relationships are perhaps the easiest to understand, but unfortunately not the most frequently used. A *one-to-one* relationship means that for every instance in the parent entity, there is exactly one instance in the child entity, no more and no less. Think of playing catch with a ball; only one person can actually throw the ball, and only one person can actually catch the ball.

One-to-Many Relationships

This type of relationship is easily the most common type of relationship, and one that most people are inherently familiar with. In a *one-to-many* relationship, one instance in the parent entity relates to many instances in the child entity. In reality, this type of relationship can actually be named one-to-one-or-more, as typically there is at least one relevant instance in the child entity, but there usually can be more. This is, semantically, more important when modeling the physical database, as the functional definition needs to specify exactly how many records there are in the parent (table) and how many there are in the child. A simple example of the one-to-many relationship is the order and order detail model. For every one order instance in the Order entity, there can be several line items, with each one being an instance in the Order Details child entity.

Many-to-Many Relationships

Of the three basic types of relationships, the many-to-many relationships is the most difficult to understand, and usually the most complicated to model. A *many-to-many* relationship exists when one (or more) of the instances in a parent entity can relate to one (or more) of the instances in the child and entity, AND vice versa. For example, imagine a data model for auto parts, specifically the seats for a sedan-style vehicle. A sedan has different types of seats; it will have to bucket seats for the driver and front passenger, and a bench seat for the passengers in the rear of the vehicle. So, in the data model, there will likely me a Models entity for the sedan, and a Seats entity for the seats. Initially, this looks like a one-to-many relationship from Models to Seats. However, many auto makers use the same seats in different models. So, there is a relationship between several instances of Models and several instances of Seats. This is a basic many-to-many relationship.

Once again, this is the logical version of that relationship; physically implementing a many-to-many relationship in a database requires the use of a third object (table) to physically enforce the rules that maintain the relationship. Because of that, it may be a good idea to model the relationship using a third entity, in order to fully describe the existence of the relationship, and to facilitate development of the physical database.

DOMAINS

As a data model develops, and attributes are defined for each entity, there are almost always attributes that exist in multiple entities, as they are a fairly generic type of data. For example, you may store addresses for customers, vendors, and employees. Those are all separate, explicit entities, and each needs addresses stored. In order to ensure consistency, you can create *domains* of attributes that are the same across entities. A *domain* is a definition of an attribute that is maintained within the logical model, but not directly attached to any given entity. Instead, when you have a domain created, you can add that domain to the relevant entities, and that domain shows up as an attribute within the entities. However, that attribute can't be edited in the entity without first intentionally separating it from the domain. This helps ensure data consistency across entities that have the same attributes. So, we can create an address domain that defines the data type and length of the address field. Doing this makes application code a little easier, as we always know how to input address data, regardless of the entity.

NORMALIZATION

Normalization is the process of grouping data in a logical way to avoid duplication and data complexity. There are many levels (or forms) of normalization, each getting more and more refined, as we walk through each form we'll help you understand where the limits of reality enter into the process.

These rules for normalization were originally developed by E.F. Codd, who was a researcher at IBM. The rules are applied as progressive forms of a data model, each one stricter than the previous. There were initially three; however, further research revealed certain potential update anomalies, and two additional forms were defined to eliminate those anomalies. Together, these are known as the Normal Forms, and are named simply according to their occurrence in the sequence, with one exception, which we'll get to in a moment.

FIRST NORMAL FORM (1NF)

For a data model to meet 1NF, all of the entities in the model must have a primary key. The primary key of the entity is an attribute, or collection of attributes, that define a unique instance in that entity. Obviously, the primary key can never be null, as it is the defining attribute for every instance. Additionally, no two instances can have the same values for their primary keys.

1NF also specifies that there can be no repeating groups. A *repeating group* is where any attribute has multiple values that relate to a single value in another attribute in the same instance. For example, if you have an entity named Artists that stores information about recording artists, you are likely to have names of albums they have recorded. If an artist has recorded more than one album, you could have an AlbumName attribute that stores the album names for each artist. However, what this means is that you have multiple values of an attribute (AlbumNames) that relate to a single value (ArtistName) for the same instance. To fix this, you'd need to split albums out to their own entity, and create a relationship between the two entities.

SECOND NORMAL FORM (2NF)

2NF specifies that all non-key attributes rely on the entire primary key. Obviously, for entities whose primary key is a single attribute, this is not a problem. However, if your entity has a composite primary key (where the primary key is made up of more than one attribute), then 2NF means that all other attributes in the entity relate to all of the attributes in the primary key. If there is an attribute that relates to only a part of the primary key, then it is likely that attribute should be part of a different entity, with a relationship back to the current entity via a foreign key. A foreign key is an attribute that is a non-key attribute in a child entity that specifically correlates to a primary key attribute in the parent entity.

A good example is an entity that stores product information. For example, there is a Product entity, whose primary key is the ProductID (a numeric identifier for the product), ProductName, and WarehouseID (to identify where the product is). Other attributes in the entity are WarehouseAddress, ProductDescription, VendorName, and VendorID . Obviously, WarehouseAddress does not actually relate to a product name; the warehouse has an address regardless of what products are stored there. So, it's very likely that there should be a Warehouse entity, and the Product entity would have a relationship to that entity defining where each product is stored.

THIRD NORMAL FORM (3NF)

3NF is the Normal Form to which most data models and databases are normalized. 3NF specifies that a data model is in 2NF, and adds the requirement that there be no transitive dependencies. A *transitive dependency* exists when a non-key attribute of an entity relies on another non-key attribute that relies on the primary key in the same entity. In other words, if there is an attribute that relies on another attribute that isn't part of the primary key, then that attribute has a transitive dependency.

As with 1NF and 2NF, the typical method of resolving this scenario is creating a new entity with the violating attributes. In the recording artists example, creating a new entity for warehouse information allows us to remove all warehouse specific information to the new entity. However, we still have the vendor information. Knowing the name of a vendor that sells a product is relevant logically, but does not efficiently allow us to store information about vendors. In this case, the vendor name is dependent on the vendor id, which is a non-key attribute. So, to resolve the issue, we create a Vendor entity and remove all vendor information to that entity (leaving VendorID in the Product entity to create a relationship between the two).

Usually, when you normalize a database to 2NF, you will almost sub-consicously normalize it to 3NF. This is because you will create entities and store only their attributes in each entity. Then, when creating relationships, you'll add only those attributes that relate to different entities, without adding attributes that belong only to the parent entities. However, be sure to review your model for transitive dependencies, to make sure you don't run in to update problems in the physical database.

BOYCE-CODD NORMAL FORM (BCNF)

Boyce-Codd Normal Form is a form that was created to handle a specific issue that can occur in data models that are normalized specifically to 3NF. Basically, 3NF never actually specified that an entity could not have more than one candidate primary key (singular or composite). It's fairly obvious that any entity with more than one primary key should be split into multiple entities; BCNF formally documents this as a rule.

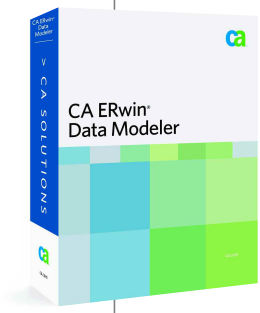
THE OTHER NORMAL FORMS

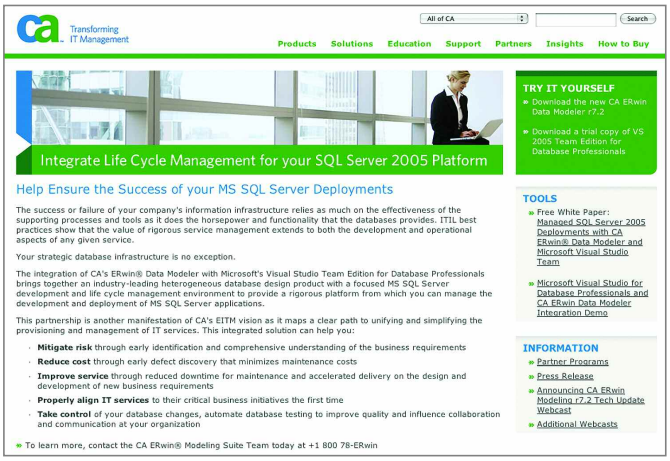
Formally, there are at least two more normal forms, 4NF and 5NF. 4NF specifies that a primary key cannot have a multivalued dependency. *Multivalued dependencies* exist when an

attribute in a composite primary key relates to multiple values in another attribute within the primary key. As with most of the normal forms, this usually requires a new entity to resolve.

5NF specifically deals with relationships amongst three or more entities, often referred to as ternary relationships. In 5NF, the entities that have relationships specified must be able to stand alone as individual entities without dependence on the other relationships. However, because the entities relate to one another, 5NF usually requires a physical entity that acts as a resolution entity to relate the other entities to one another. This additional entity will have three or more foreign keys, based on the number of entities in the relationship, which specifies how the entities relate to one another. This is how many-to-many relationships are actually implemented. So, if a many-to-many relationship is properly implemented, the database is in 5NF.

There is work being done on even more normal forms, particularly in the spatial and temporal database realms. However, those have not been finalized, and are subject to change, so they fall outside of our discussion here. For most relational data models, and their databases, these normal forms will provide a stable, workable solution.





Visit: ca.com/erwin/msdn. *CA ERwin Data Modeler provides a way to create a logical structure of your data and map that structure to a physical data store.*

DENORMALIZATION

While a properly normalized database provides high performance and flexibility for online transactional databases (OLTP), such as ordering systems, there are certain situations where you may need to denormalize entities/tables for performance reasons. Particularly for reporting and warehousing situations, the queries run require joining large amounts of data together and performing aggregations against that data. Running these types of queries against a perfectly normalized (3NF or 4NF) database will result in the need to join a large number of tables together, which increases overhead and slows performance. In order to satisfy those queries, you may find it necessary to create tables that contain multiple entities worth of attributes that have a loose, logical relation to each other. Typically, a separate database will be created that contains the denormalized versions of the tables, and a data load process will be created that moves data from the normalized database to the denormalized database for reporting and warehousing purposes. Do not be afraid to denormalize for this reason; just be cognizant that you should create separate objects to satisfy these requirements; if you denormalize the objects that an OLTP system is using, you'll take a performance hit.

CONCLUSION

Data modeling is key to the successful design of any database. By gathering requirements and designing a data model before any actual development work is done can help tighten data definitions for both the database and the application code, increasing developer efficiency and reducing confusion. Additionally, a properly normalized database will perform well for most read/write, online transactional processing databases.

Josh Jones and Eric Johnson are the Operating Systems and Database Systems Consultants, respectively, for Consortio Services (www.consortioservices.com). Eric and Josh have written a combined three books including the forthcoming Architecting Database Models for SQL Server.