2

OF A THREE-PART SERIES

Building and Maintaining a Database from an ER Model

By Josh Jones and Eric Johnson

CA ERwin

TABLE OF CONTENTS

Designing and building a database begins by building a high quality data model. Without having the guidance of a data model, database design tends to be very inefficient. Often, developers who build databases without using a data model find themselves constantly revisiting the database, changing its design over and over again as the application is written. This slows down the developer, and will result in a database that can only support that specific version of the application, with very little flexibility and an unnecessarily high potential for data corruption.

To avoid these issues, you should always develop a data model during the initial stages of application design. In this paper, we will look at how you go about building a data model and ultimately a database. First we need to look at exactly what our database will contain. From there we will build the logical model and then move that logical model into a physical SQL Server database.

THE PROJECT

We are going to be working with a database for tracking customers and orders. If you are familiar with any writings or classes you have probably seen the customer/order model used many times. The reason for this is that it fits very well into the concept of a model and it's a real world example to which we can all relate. Here we model the four major data components—Customers, Orders, Order Details, and Products. In addition, this example makes it easier to understand and work with these concepts in context, so our database is being built for a small online movie store whose primary products are DVDs.

Some aspects of the model are overly simplistic for real life, as we are using this to illustrate a point, not to actually serve as a product and order management system. If you were building a real model, you would first sit down with you customer in an attempted to flesh out all the requirements for the model and arrive at a list of entities and attributes that contains the objects we will work with here.

BUILDING THE DATA MODEL

When you first sit down to build your logical model, you will start with the list of entities and attributes you obtained from your customer and by analyzing the problem domain. This information will help you design the first pass of the model. We say first pass, because you will likely need to make adjustments to this model after you and your customer review the initial design. So let's use CA ERwin® Data Modeler and get to work on building that first cut of a model.

ENTITIES

We need to begin laying out our model by placing the appropriate entities into the model. Depending on the information you have gathered and your preference, you might add your new entities complete with attributes and relationships at this stage, and that's fine. We, however, are going to build this model in discrete stages to help you understand each component. For this first pass, we are going to add the entity alone, no attributes and no relationships. The only exception to this rule will be our primary key. We are going to use a surrogate PK named ObjectID of type integer for all tables. To start adding entities, you will need to refer to the list of entities you developed during requirements gathering. Our list is shown below.

- Customers
- Orders
- Order Details
- Products

All we are going to do at this point is add each entity to the model and set up the primary keys for each, which gets us to the model shown in Figure 1.

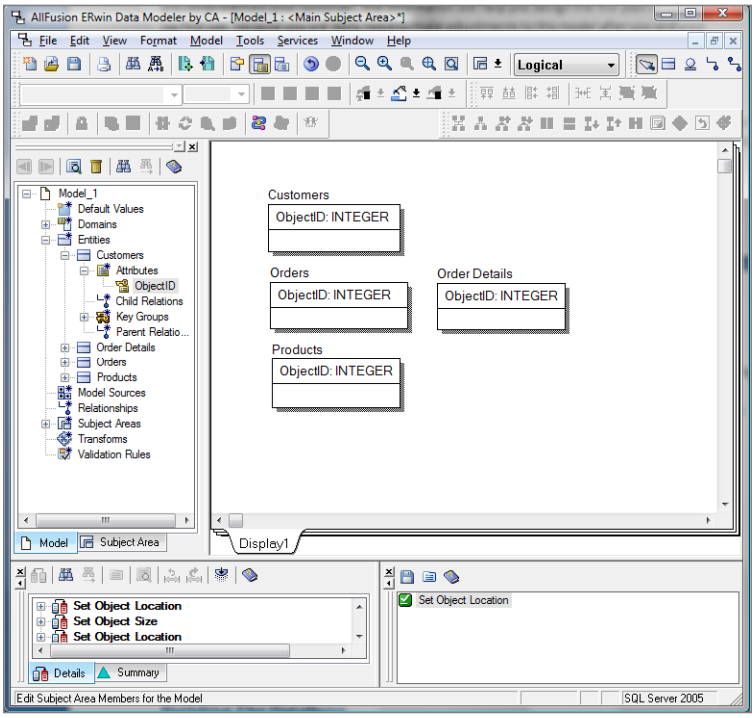


Figure 1: *The logical model with only entities at the start of the modeling process.*

ATTRIBUTES

Our next step will be to add all the attributes to the various entities. Keep in mind that as you do this, you may end up with additional entities. We will look at an example where this is true. While working with the customer, we decided that the following information needed to be tracked for each entity.

- Customers
  - First Name
  - Last Name
  - Phone Number
  - Address
- Orders
  - Order Number
  - Date and Time
  - Total Cost
  - Ship To Address
- Order Details
  - Product Ordered
  - Quantity
  - Price
  - Total Price for the Line
- Products
  - Product Type
  - Title
  - UPC Code
  - Genre

From this list, we need to figure out all of our entities and data types. Some of this you can infer and some of this will require you to ask the customer for more detail. In addition, a few of these elements could necessitate that we create additional entities to store the information. Let's take these one at a time. For customer, we need to track first and last name, phone number, and address. The first three pieces of data will make fine attributes, so we add them to the entity. Names are not generally too long so we will make the name attribute with varchar(50) data types. Phone numbers will be stored as numbers only and formatted in the application, so we will store these using the integer data type. Our entity up to this point is shown in Figure 2.

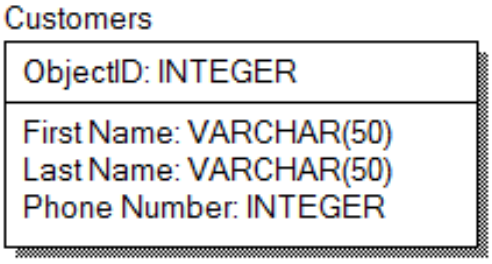


Figure 2: *The Customers entity without address detail.*

To complete the Customers entity, we need to store the address information. Address cannot be just one entity (we may want to query by state, for example), so in this case we need to make it five attributes.

- Address Line 1
- Address Line 2
- City
- State
- Zip Code

We also made some decisions about data type based on our understanding of each of these attributes. In addition, we defined which columns can and cannot contain NULL data. Basically, if the attribute is required, you define it as NOT NULL, and if it's optional, you define it as NULL. In the case of the Customers entity all the data is required except for Address Line 2. Once we add these elements to the Customers entity, we have the entity shown in Figure 3.

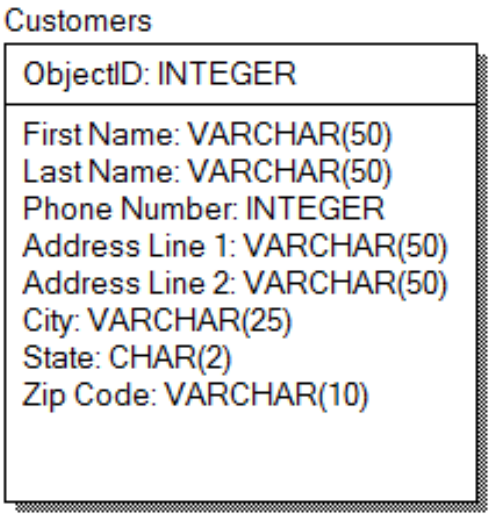


Figure 3: *The Customers entity with address detail.*

We do the same thing for the Orders and Order Details entities. These two entities are both pretty straightforward, except for the Product Ordered in Order Details. This attribute references an instance of another entity; as such this will likely be a relationship so we will deal with it in the next section. As for the other attributes, we just need to decide which data types to use and define whether or not the attribute is required. Here is the complete list of attributes for both the Orders and Order Details entities with data types and null settings.

- Orders
  - Order Number VARCHAR(20) NOT NULL
  - Order Date Time DATETIME NOT NULL
  - Total Price MONEY NULL
  - Address Line 1 VARCHAR(50) NOT NULL
  - Address Line 2 VARCHAR(50) NULL
  - City VARCHAR(25) NOT NULL
  - State CHAR(2) NOT NULL
  - Zip Code VARCHAR(10) NOT NULL
- Order Details
  - Quantity INTEGER NOT NULL
  - Price MONEY NOT NULL
  - Total Price MONEY NOT NULL

Last, but not least, we have to take care of the Products entity. On the surface this table seems to be straightforward. It only contains a couple attributes, but Genre throws a wrench into the works. If we just create an attribute named Genre, each title would only be able to belong to a single Genre; however, in reality, some movies exist in multiple genres. To solve this problem, we are going to make a new entity called Genre to hold all the possible genres. We will join Product and Genre later when we talk about relationships. After we create the new Genre entity, Figure 4 shows what our model looks like up to this point.

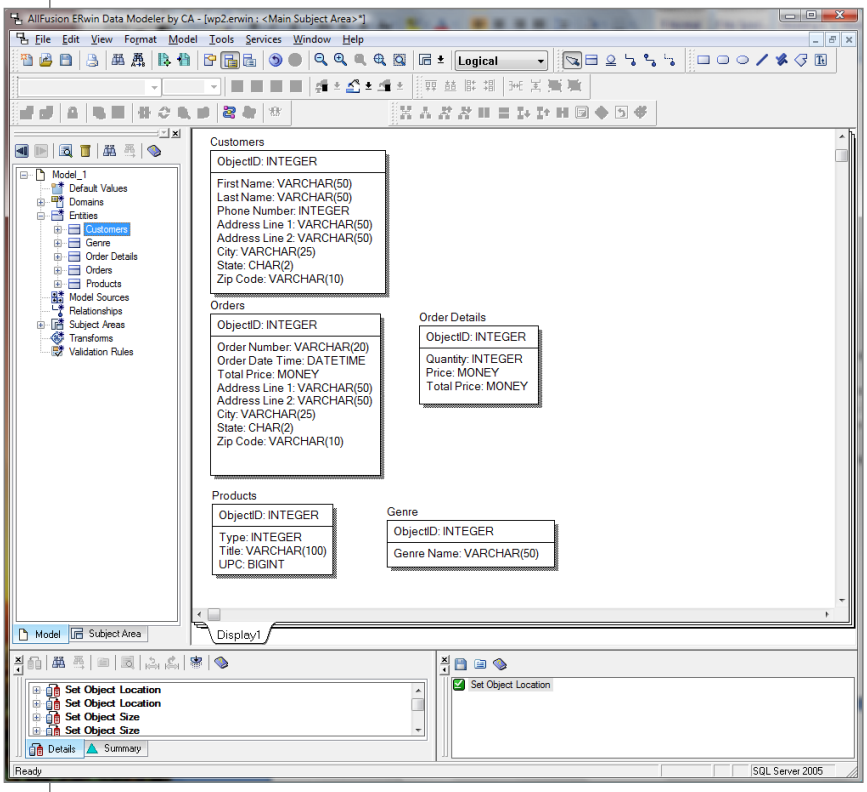


Figure 4: *Data Model with all of the entities.*

RELATIONSHIPS

The last big step in order to get our model into a working state is adding the relationships. Sometimes relationships are quite obvious and other times we need to go back to the requirements in order to determine where they belong. One trick you can use is to look for certain keywords when you read through the description of the model. In general, nouns are entities or attributes and verbs will denote relationships. What do we mean by this? To illustrate, let's examine this brief description of the project.

> ***"Our customers place orders through our website. Each order will have one or more items with various quantities for each item. Each order item in an order will contain one of our movies. We also need a way to place each movie into one or more genres when storing information detail."***

If you look for the verbs in this description, you will see that "Customers place Orders", "Orders have Items", "Items contain Movies", and "Movies belong to Genres". We reworded these statements a little to show the relationship between each entity.

Now that we see where the relationships belong, we have to decide which relationships to use. Let's start with the relationship between Customers and Orders. Customers can place more than one order, and each order can only belong to one customer. This would require a one-to-many relationship, or simply 1:M. The same goes for Orders to Order Details, as each order can have more than one detail line, but each detail line belongs to just one order; another 1:M. It's almost the same for Order Details to Products, except that it's reversed. In this case, each Order Detail can be for one product, but each product can exists on multiple detail lines, so we have a M:1. This is still a 1:M relationship when looked at from the perspective of the Products entity. Finally, we have the relationship between Products and Genres. Each product can be a part of one or more Genres, and each Genre can contain one or more Products. Here we have a M:M relationship. Now all we need to do is add these relationships to our model, as shown in Figure 5.

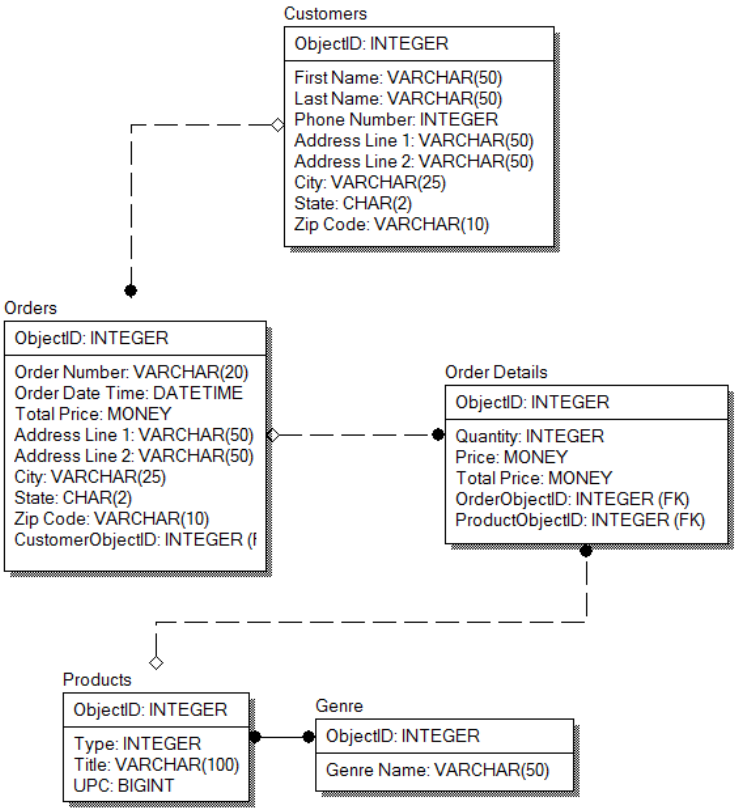


Figure 5: *Data Model complete with relationships*

At this point, we have a working logical model. There are many other areas you can work on in your logical model, such as constraints, but we are not going to go into further detail here. The next step is to convert what we have into a physical database.

BUILDING THE DATABASE

Once you have a logical model, you need to create a physical model. This will be what you implement in SQL Server. You have two options here; you can just start creating your database objects directly in SQL Server, or you can model them in CA ERwin DM. We recommend the latter, as it automates the process, allowing you more flexibility, documentation for your physical database as well as a clear mapping between your logical and physical designs. When setting up your physical model, much of what you have done in the logical side will be used as a starting point. As you add attributes to entities in CA ERwin DM, you are able to give each a name and a column name, as seen in Figure 6. This allows you to account for naming differences between your logical and physical model. If you do this as you go along, you will have less work to do when you create your physical model.

To begin your physical model, switch to the physical model view in CA ERwin DM. Here you will see all your entities with the same names and relationships that exist in your logical model. The attributes however will contain the Column Name you provided. Now it's just a matter of

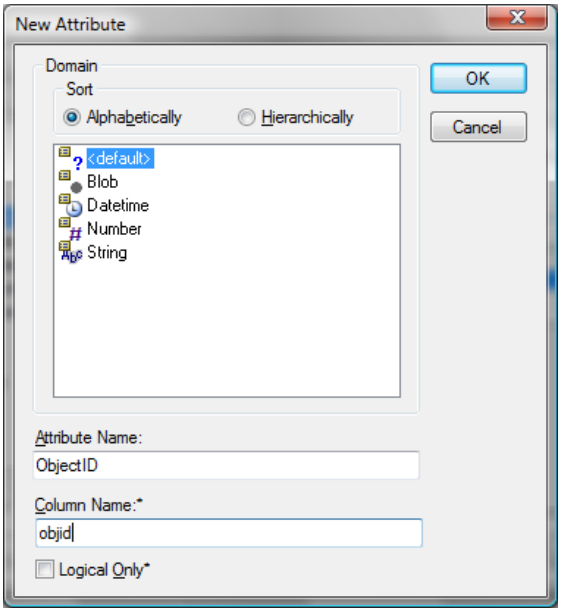


Figure 6: *The new attribute dialog box in CA ERwin DM.*

changing some names and making any modifications to the physical model that are not represented in the logical. This may mean splitting an entity to be stored in two or more tables or storing several entities in one table. There is no hard and fast rule on when to do this, it will be different for each database you build.

One thing you must do however, is physically implement your M:M relationships. In a logical model, M:M relationship are just a line between entities, in a physical database you need a place to store the PKs of each table to create the relationship. This is done via a 3rd table known as a join or junction table. To implement the join table in your model:

1. Set the Logical Only property in the M:M relationship in your logical model (This removes the relationship from the physical model);
2. Create the join table in your physical model with the Physical Only property set;
3. Add an indentifying 1:M relationship between each of tables in the M:M and the new join table.
4. That's all there is to it. This set of two 1:M relationships physically implement a M:M.

Once you have your physical model ready to go, CA ERwin DM can help you create and

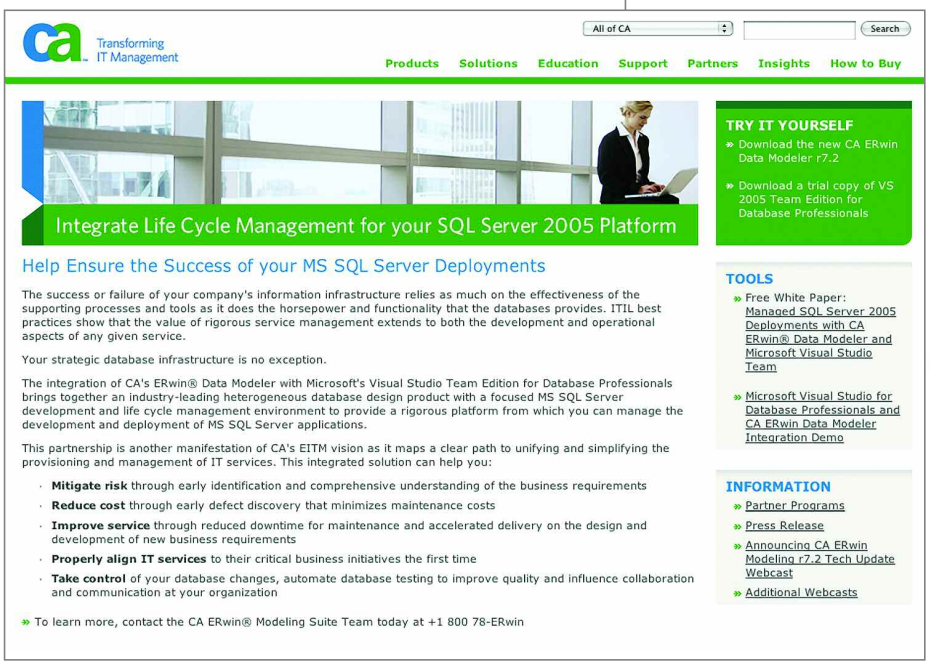
deploy the creation scripts for SQL Server. If you took our advice and used CA ERwin DM to do your physical model, you have a great document outlining your database and one place to easily make and track changes.

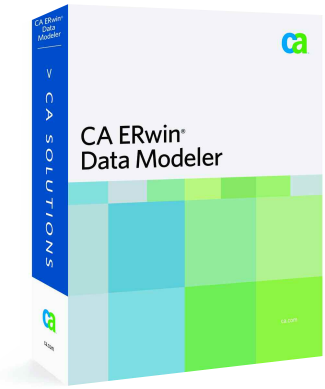
CONCLUSION

Without a doubt creating a data model will enhance physical database development. Having a well thought out data model will make creating the physical database quicker, and will help ensure consistency as the development process moved forward. It will also ensure that as new developers become involved in current or future iterations of the application, they can readily understand the data model and use it effectively. There is a historical record of how the database was developed, and a basis to add new functionality without breaking the original design. This example and explanation provide a short foundation about creating models and about how CA ERwin DM can help you to streamline this process.

ABOUT THE AUTHORS

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.





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.*

ca.com/erwin/msdn