Chapter 2
Developing a Logical Data Model

Data is the heart of our Visual FoxPro applications. Getting data safely stored and updated in our databases is our "bread and butter." Crucial to doing this is good design for our data model. We need to understand the difference between the logical data model, discussed here, and the physical data model, discussed in Chapter 8, "Creating a Physical Database," as we move through this design process. Without good design, we'll always be fighting the awkwardness and integrity problems to which poor design gives rise.

Dr. Edgar F. Codd's landmark paper, "A Relational Model of Data for Large Shared Data Banks," was published in *Communications of the Association for Computing Machinery* in June 1970. In it he gave us a definition of a good database structure and a process for "normalizing" our data by sequentially applying rules to the tables or "entities." Others, including Raymond F. Boyce, joined Codd in refining the definition and the process of normalizing the data.

The goal of data normalization is to avoid problems caused by poor design of the database. Every type of duplicate data and every type of dependency adds complexity to our database that could easily be avoided by better design. Let's avoid patched, duct-taped spaghetti code from the beginning by understanding the rules of normalization and applying them. In addition to the normalization process, we should also understand the circumstances where it makes sense to denormalize our data.

Part of the difference between the "logical" data model, which we are discussing here, and the "physical" data model, discussed in Chapter 8, "Creating a Physical Database," is the way we think of key fields. In the logical model, we think of key fields as a way to logically join tables. In the physical database, we use these logical keys as the basis for our index expressions.

Normalization rules

Applying "rules" to the data in a sequential fashion carries out the process of table normalization. Then, if there are specific reasons, the data may be denormalized. Here's a statement of the rules, with more detailed explanations following:

- First Normal Form: There should be no repeating fields in the table.
- Second Normal Form: The table is in 1NF, and all non-key fields depend on the whole primary key for their value.

- Third Normal Form: The table is in 2NF, and there is no co-dependence between non-key fields.
- Boyce-Codd Normal Form: The table is in Boyce-Codd Normal Form if it is in 3NF for any fields that are alternate candidate keys.
- Fourth Normal Form: The table is in 3NF, and it has no independently multi-valued primary keys.
- Fifth Normal Form: The table is in 5NF if it is in 4NF and there are no pairwise cyclical relationships in primary keys that have three or more components.

What's really important here is not some academic definition of normalization rules, but how we should design our databases under these rules.

First Normal Form (1NF)

Normalizing a table to 1NF means that fields with repeating values should not be allowed, but should be moved to a different table. Cindy's automobile insurance company's database is an example of data that is not in 1NF. When her third teenager's car was added to the insurance policy, Cindy wondered why she started getting two insurance bills to pay. The insurance company only had room for four automobiles on one policy. Guess why! **Figure 1** shows what Cindy imagines their table design looked like.

***Figure 1**. Non-normalized table limiting a policy to four vehicles.*

The solution to this problem is to leave the policy owner's demographic information in the Owners table and move the automobiles on the policy to a Vehicles table. The automobiles would then be associated with the policy owner in the Owners table through an OwnerID primary key, as shown in **Figure 2**.

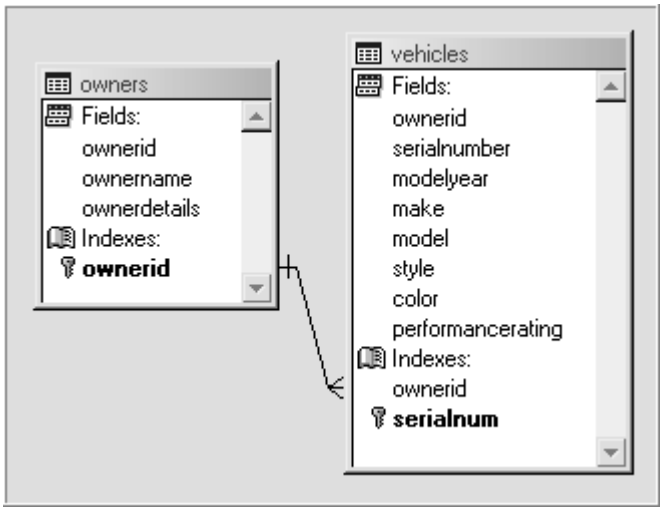


***Figure 2**. Owner and vehicle data in first normal form. Policy owners may have as many vehicles as necessary on one policy. It is difficult, however, to change the performance rating of all vehicles of this ModelYear/Make/Model when the tables are not normalized.*

Second Normal Form (2NF)

A table in 2NF must not have fields that do not depend on the whole primary key for their value. Suppose the insurance company kept track of "high-performance" cars because their drivers exhibited more "risky" driving behavior and should have a different insurance rate. Their Vehicles table currently stores OwnerID, SerialNumber, ModelYear, Make, Model, Style, Color and PerformanceRating. The primary key in this table is SerialNumber. While ModelYear and Color are both dependent on SerialNumber, PerformanceRating depends only on ModelYear, Make and Model.

The insurance company might decide that Cindy's 1996 magenta Neon Sport's PerformanceRating is really more like that of the Neon ACR than the basic Neon model and wish to change the insurance rate on all Neon Sport vehicles. In the current schema (see Figure 2), they would need to change all records with ModelYear/Make/Model = "1996/Plymouth/Neon Sport" to PerformanceRating = "High."

A properly normalized database (see **Figure 3**) would have a Model table where the PerformanceRating was stored. The insurance company would only need to change the value of PerformanceRating once in the Model table to adjust every policy owner's rate for owners of that type of vehicle.

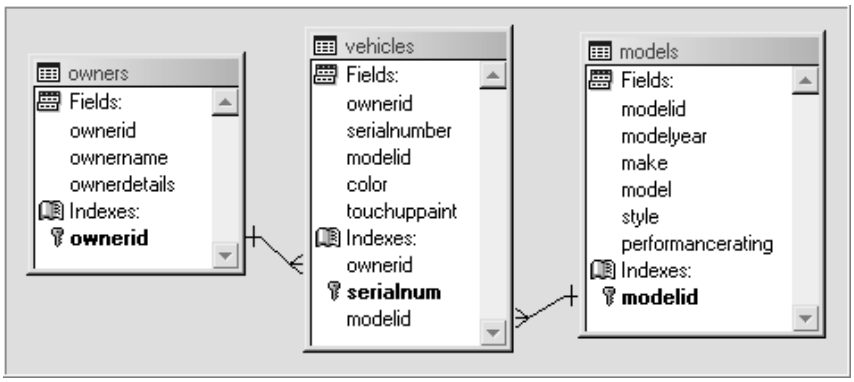


***Figure 3**. Owner and vehicle data in second normal form—it's easy to change the performance rating of a particular vehicle type when the tables are normalized.*

Third Normal Form (3NF)

Tables in 3NF do not have dependencies between non-key fields. Suppose the insurance company stored the touch-up paint color number for Cindy's Neon in the Vehicles table (see Figure 3) in case the vehicle was damaged. The data is not normalized because TouchUpPaint is not dependent on either key field. In **Figure 4**, the touch-up paint color number is only dependent upon the ColorID field and has been moved to a table other than the Vehicles table, normalizing the schema again.

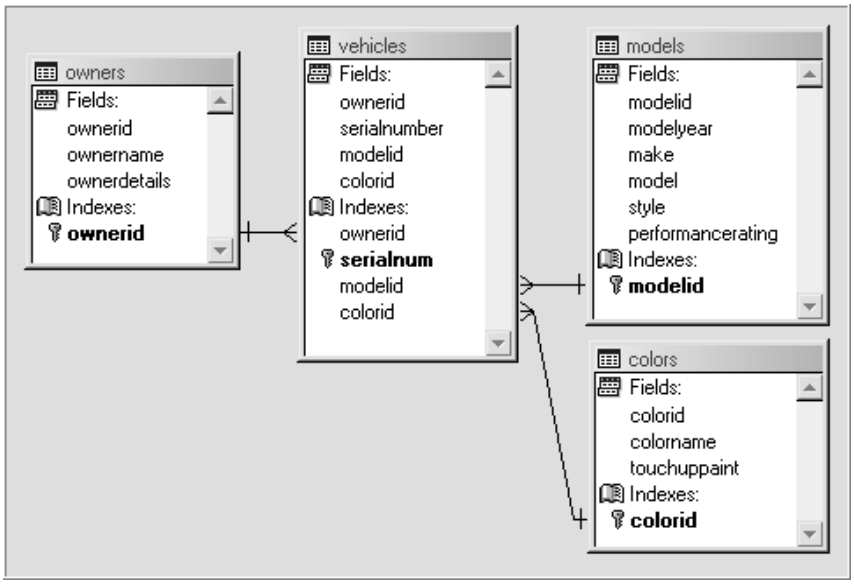


***Figure 4**. Owner and vehicle data in 3NF—the logical data model is normalized when TouchUpPaint color number depends only upon a key field in the table where the value is kept.*

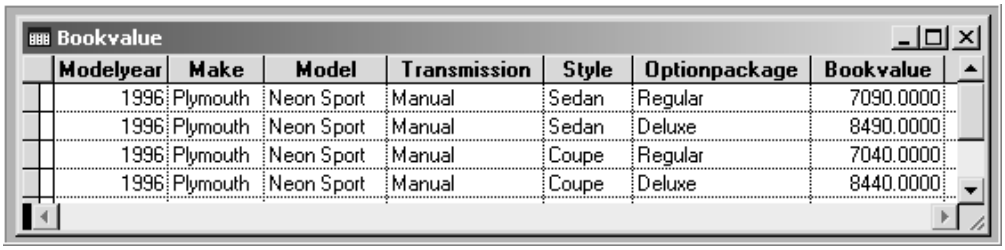
Many people stop at 3NF in designing their databases, and in fact only more complex databases need to deal with forms beyond 3NF.

Boyce-Codd Normal Form (BCNF)

Boyce-Codd Normal Form deals with the existence of candidate keys. Suppose automobile manufacturers imprinted a unique number on the engine block as it came off the line and the insurance company added this field to their Vehicles table. Because each vehicle already has a unique SerialNumber and can have only one EngineBlockID, either could be removed and the table would still be in 3NF. Specifying that EngineBlockID must be unique and that the table must be in 3NF when EngineBlockID is the primary key puts this table in Boyce-Codd Normal Form.

Fourth Normal Form (4NF)

Fourth Normal Form requires that the table have no independently multi-valued primary keys. Suppose the insurance company is planning their BookValue table. Shown in **Figure 5**, it contains the ModelYear, Make, Model, Transmission, Style, OptionPackage and BookValue. The first six fields together (ModelYear + Make + Model + Transmission + Style + OptionPackage) comprise the primary key—you need all six to access the BookValue. Transmission, Style and OptionPackage are each independent multi-valued attributes of ModelYear + Make + Model.



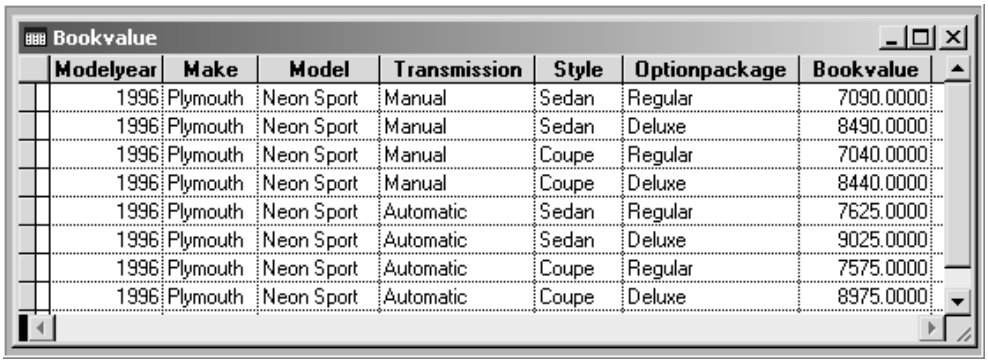
Bookvalue

Modelyear	Make	Model	Transmission	Style	Optionpackage	Bookvalue
1996	Plymouth	Neon Sport	Manual	Sedan	Regular	7090.0000
1996	Plymouth	Neon Sport	Manual	Sedan	Deluxe	8490.0000
1996	Plymouth	Neon Sport	Manual	Coupe	Regular	7040.0000
1996	Plymouth	Neon Sport	Manual	Coupe	Deluxe	8440.0000

***Figure 5**. Transmission, Style and OptionPackage are independently multi-valued attributes of ModelYear + Make + Model. What happens when we want to add BookValue data for Automatic transmissions?*

If the insurance company needed to add the "Automatic" transmission to the table, they would need to add one record for each combination of Style + OptionPackage as shown in **Figure 6**.

You can put a table like this into 4NF by creating multiple tables to handle the different multi-valued components. That means creating separate tables (see **Figure 7**) for the value added to the BookValue by each of the Transmissions, Styles, and OptionPackages choices. Those who have ever read *Kelly's Blue Book* are familiar with value being added to a base price for each of these.



Bookvalue

Modelyear	Make	Model	Transmission	Style	Optionpackage	Bookvalue
1996	Plymouth	Neon Sport	Manual	Sedan	Regular	7090.0000
1996	Plymouth	Neon Sport	Manual	Sedan	Deluxe	8490.0000
1996	Plymouth	Neon Sport	Manual	Coupe	Regular	7040.0000
1996	Plymouth	Neon Sport	Manual	Coupe	Deluxe	8440.0000
1996	Plymouth	Neon Sport	Automatic	Sedan	Regular	7625.0000
1996	Plymouth	Neon Sport	Automatic	Sedan	Deluxe	9025.0000
1996	Plymouth	Neon Sport	Automatic	Coupe	Regular	7575.0000
1996	Plymouth	Neon Sport	Automatic	Coupe	Deluxe	8975.0000

Figure 6*. In order to add BookValue data for vehicles with Automatic transmissions, we need to add a total of four records to this table for each ModelYear + Make + Model combination.*

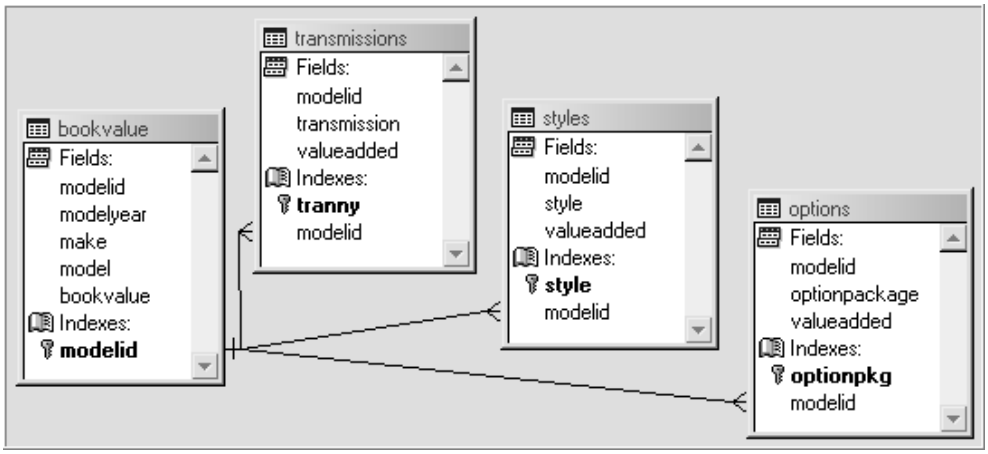


Figure 7*. Vehicle options normalized to Fourth Normal Form. Each multi-valued component, and the value it adds to the BookValue of the vehicle, is stored in its own table.*

Fifth Normal Form (5NF)

A table in 5NF must not have pairwise cyclical dependencies of the primary key that are composed of three or more component fields. This deals with the situation of a table whose primary key is composed of three or more fields, an easy occurrence in a many-to-many table like what the insurance company might see when processing a claim.

Suppose that the insurance company has a table for processing claims that matches Branch Offices, Claims Adjusters and Outside Agencies. When a new Adjuster is employed, serving all branch offices in the state, a record needs to be added for each combination of outside agencies. For any one value, you always need to know the other two values in order to define a

record. The solution to this problem is to separate the Outside Agencies into its own table that has a many-to-many relationship with the OfficesAdjusters table.

An even better plan is to separate all three types of data into BranchOffices, ClaimsAdjusters and OutsideAgencies. Then, when a claim is processed, it can have a foreign key to the BranchOffices to show which office handled it, a foreign key to the ClaimsAdjusters to show which adjuster was assigned, and the ClaimID can be a foreign key in the ClaimsAgencies table, which serves as a junction between the claim and the OutsideAgencies involved.

Most of us never see these 4NF and 5NF problems occurring, because our design was done properly from the beginning! In general, we should not be "penny wise and pound foolish" about adding an additional field to a table to serve as a primary key, eliminating the need to deal with keys made up of several values combined together.

The primary key

The primary key is a value that uniquely defines a record. It can be numeric or character, a single value or a combination of several values. In Visual FoxPro, we can define a Primary index on a field in a table within a .DBC and the FoxPro data engine will automatically prevent us from entering any duplicate values. We can have only one Primary index, but for other fields that could be used as a primary key—for example, the EngineBlockID field mentioned earlier—we could specify that the field have a Candidate index. FoxPro gives us the same benefit for a Candidate index as for the Primary index—it prevents the entry of duplicate values.

In practice, the primary key should never be meaningful, real-world data that the user enters. No Social Security numbers, no traffic ticket numbers and so forth. Cindy's middle child went through half his life with the wrong SSN, and Cindy had quite a time getting things corrected in several systems that disregarded this rule!

We will discuss primary keys further in Chapter 8, "Creating a Physical Database."

Relationships

Entity relationships fall into three categories: one-to-one, one-to-many and many-to-many.

One-to-one relationships

One does not see the one-to-one relationship often in database design. Any tables with a one-to-one relationship between their records could logically be combined into one larger table. Reasons for separating the tables might include the need to make a very large table smaller, to have more than FoxPro's 254 fields, or to avoid keeping seldom-used data open and available while an application is running. As you can see, these constraints come into play when the logical database design is translated into the physical database design.

One-to-many relationships

One-to-many relationships are at the heart of relational database design. They are often called "parent-child" or "master-slave" relationships. We see these relationships in our everyday lives. For example, there are natural one-to-many relationships between customers and orders, orders

and line items, installment loans and monthly payments, and the Owners and Vehicles tables in the insurance company examples described earlier.

Many-to-many relationships

We also see many-to-many relationships in our daily lives. Teachers and students, physicians and patients, library materials and patrons, and parts and orders are all examples we are familiar with. Data for instances of a many-to-many relationship is housed in a type of table called a junction, or linking, table. Sometimes all this table contains are the primary keys of each of the tables it is joining, and other times it contains additional data. For example, many library patrons may borrow and return many books. After each return, the condition of the book may be noted. The junction table would contain the book key, the patron key, the borrow and return dates, and the condition of the book when it was returned.

In our insurance company example, the Vehicles table (see Figure 4) is a junction table for the many models and many colors possible in the vehicles covered by the company's policies. It stores additional information, including the policy owner and serial number of the vehicle in question.

A many-to-many relationship can be thought of as a combination of two one-to-many relationships. One merely needs to put blinders on and look only one way out of the junction table to avoid confusion.

Foreign keys

A foreign key is a field in the current table that holds the values of the primary key of another table. In the child table, the foreign key field contains the primary key of the child record's parent record. Foreign keys are used to define relationships among tables and in the JOINs of views. A junction table may contain only two fields, both of which are foreign keys. Because it is a junction table, it essentially has two parent tables. Again, let us mention that primary and foreign keys are used to define logical relationships in the logical data model and to define indexes in the physical data model. Visual FoxPro uses indexes to carry out the business of persistent logical relationships between tables.

Referential integrity

Referential integrity refers to the validity of relationships between entities in a database. A database can be said to have referential integrity when there are no unmatched foreign key values. That means that there will never be a "dead link" between a foreign key in one table and a record in the table where the foreign key is primary. Broken referential integrity—for example, deleting a parent record when there are child records present—leaves what are commonly called "orphan" child records.

FoxPro provides us with database insert, update and delete triggers that allow us to cascade changes made in a parent record to all of the child records. (The same triggers also allow us to prevent those changes in the parent should that be our choice.) These database capabilities make enforcing referential integrity much easier than hand-coding each time and in each part of the application that contains code to insert, update or delete records. These triggers will be discussed more in Chapter 8, "Creating a Physical Database," where we go into detail about physical design issues.

Business rules, data integrity and the data model

The business rules for data integrity are a part of the logical design. When the insurance company transfers a claims adjuster from the Northern territory to the Southern territory, what should happen to the outstanding claims left in the Northern territory? The customer would sensibly tell us that the claims overseen by the adjuster should not be deleted from the database when the adjuster is deleted from the list of adjusters who are active in the Northern territory. This would lead to orphan child records. They might prefer to keep the adjuster active in the Northern territory until all of his claims are resolved, transfer the claims to the Southern territory with the adjuster even though the addresses aren't in the south, or transfer the claims to another adjuster who is active in the Northern territory.

The customer might even dictate that the child records be ignored when the parent record is deleted. It might be acceptable to the public library to have orphan borrowing transactions for a book that can no longer be identified, but we would advise the insurance company that their clients would not be happy should their claim records become orphan children assigned to no claims adjuster!

The types of rules mentioned previously are "row-level" rules because they are applied to a whole record of data at one time. There are also business rules for individual fields. For example, in the physical data model, it makes sense to specify that the insured value of an automobile cannot be blank or a negative number. A business rule for the insured value of an automobile, determined by the insurance company, might specify that the insured value of an automobile can only be equal to Blue Book value if the customer has purchased the "Basic" or "Standard" insurance options, and that no towing claims can be filed by customers who have not purchased towing insurance.

Business rules for data integrity float in a "gray area" between the logical and physical designs, and in a "gray area" between the presentation tier and the data tier of the application. You'll find that many of the topics we mention here will be covered again in Chapter 8, "Creating a Physical Database."

Denormalization

Denormalization is the process of "breaking the rules" for a specific reason. Reasons to denormalize include:

- Reporting
- Data warehousing
- Performance

Denormalization for reporting

The first and most obvious situation we think of when we denormalize data is for purposes of reporting. When we create a view or select data into a cursor for our reports, we fill in all of the parent data, lookup values and so on, so they will be readily available for presentation. This is not "real" denormalization, though, because the data does not persist in this format after we have presented it to the user.

Denormalization for data warehousing

Data in a warehouse is usually static. An example of this is the decision support database Cindy uses to track trends in the hospital where she works. Last month's physician productivity figures are "old news" in the sense that there will be no more activity added after the closing date. Since Cindy and other users access this data often, and usually in the same way, the DBA keeps a denormalized copy of the data available, speeding up queries. Since the data is mostly static, there aren't problems with updating the non-normalized fields when the source data changes.

Denormalization for performance

A fully normalized database might require many JOINs among its tables on the way to constructing views that are commonly used by the application. JOINs can be costly in time and server resources, and selective denormalizing can really be beneficial. Perhaps the insurance company will want to see the performance rating of a vehicle every time the policy information for that vehicle is accessed. They know that performance ratings won't change often, and they want to be able to bring up policy information as quickly as possible when an agent is on the telephone with a customer. Storing the performance rating in the Vehicles table would result in denormalized data, but it would add a significant gain in access time. The downside—having to update the Vehicles table appropriately when the performance rating for a particular vehicle changes—seems a small tradeoff.

Sample Questions

The purpose of a foreign key is to:

A. Define the relationship of the current table with a table containing foreign-language translations of all the data in the current table.
B. Define the relationship of records in the current table with their child records in the child table.
C. Define the relationship of records in the current table with their parent records in the parent table.
D. Get you into the washroom in the Embassy.

Answer: C

The insurance company needs to assign claims to categories for the home office, the traffic safety board and the vehicle safety board to track factors involved. Each of the agencies that wants category assignments has its own list of factors to be evaluated for each claim. The insurance company "doesn't want anything complicated" and suggests some logical-value columns added to the Claims table denoting such categories as "weather," "vehicle malfunction" and so on. You tell them that there will always be new categories to add, that each of the outside agencies will have its own definition of "weather" or "vehicle malfunction," and that these may change over time. You also tell them that table structure changes are costly.

You recommend:

A. Adding a ClaimType column to the Claims table and adding a Factors table with a one-to-many relationship between the FactorID and the ClaimType in the Claims table.
B. Adding a Factors table and a junction table with columns FactorID and SerialNumber creating a many-to-many relationship between the Factors table and the Vehicles table.

C. Adding a Factors table and a junction table with FactorID and ClaimID creating a many-to-many relationship between the Factors table and the Claims table.
D. Adding a Factors table and a FactorAgencies table with a many-to-one relationship between the AgencyID foreign key in the Factors table and the AgencyID in the Agencies table.

Answer: C

You see design flaws in the examples at the beginning of this chapter. You are ready to write to the authors, saying that:

A. Using the vehicle SerialNumber as a primary key in the Vehicles table breaks the rule about using a meaningful value as a primary key.
B. Using the vehicle SerialNumber as a primary key in the Vehicles table breaks the rule about using a user-entered value as a primary key.
C. The primary keys of the Transmissions, Styles and Options tables are ModelID + either Transmission, Style or OptionPackage, and multi-part primary keys may cause problems later on. A single numeric or character value would be a better choice.
D. A and B only.
E. A, B, and C.

Answer: E

Further reading

- *Joe Celko's SQL For Smarties: Advanced SQL Programming, Second Edition*, Joe Celko, Chapter 2, "Normalization"
- *Effective Techniques for Application Development with Visual FoxPro 6.0*, Jim Booth and Steve Sawyer, Appendix Two, "Relational Database Design"