

Keys in Relational Databases: Theory and Practice. Part 3.

Dr. Tom Johnston
MindfulData.com

Key Problem #3. Embedded and Non-Embedded Foreign Keys.

Key problems 1 and 2 are caused by homonyms, by one syntactic structure (primary keys or foreign keys) playing multiple semantic roles (identifying and describing, or relating and describing). The third problem I will discuss is what goes wrong when more than one syntactic structure is used to implement one function. These different syntactic structures, because they are semantically identical in function, are synonyms.

In relational databases, there are two syntactic structures used to implement relationships. One is a foreign key “embedded” in one of the two related tables. The other is a third table which relates two tables by pairing foreign keys, one for each of the two related tables. This table is usually called an associative table or, more informally, an “xref” (for “cross-reference”) table.

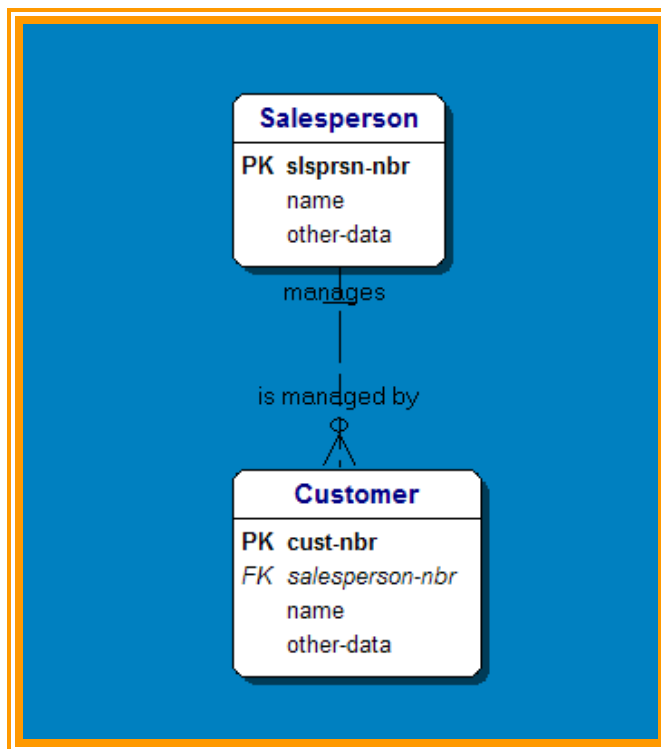
To illustrate an embedded key, let’s use a Customer and a Salesperson table. We will assume that the semantics of the relationship between salespersons and customers includes the rule that one salesperson may manage any number of customers, but that each customer may be managed by at most one salesperson. This is a *one-to-many* relationship from salespersons to customers. It is illustrated as sample data in Figure 9, and as a data model diagram in Figure 10.

<u>slsprsn-nbr</u>	name	other-data
S1	Smith
S2	Jones

<u>cust-nbr</u>	name	<i>slsprsn-nbr</i>	other-data
C1	Brown	S2
C2	Peters	S1
C3	Morris	S2

Figure 9. Sample Data: Embedded Foreign Keys.¹

Slsprsn-nbr is a foreign key from the Customer to the Salesperson table. It exists as one of the columns in the Customer table, and for that reason I call it an “embedded” foreign key. It is one of two syntactical forms that implement relationships in relational databases.



¹ In Sample Data tables, my notational conventions are these. Primary keys are the left-most columns, and are underlined. Foreign keys are italicized.

Figure 10. Data Model Diagram: Embedded Foreign Keys.

Now let us suppose that our company has found that some customers have become so important that we need to assign two or more salespersons to them. Since one salesperson can still manage several customers, this change means that there is now a *many-to-many* relationship between customers and salespersons. We cannot express this relationship by putting a foreign key in either table, because that would permit rows in whichever table we made the child table to be related to at most one row in the parent table. So instead, we must create a third table, technically called an associative table, but frequently just called an xref table. It looks like this:

<u>slsprsn-nbr</u>	name	other-data
S1	Smith
S2	Jones

<u>cust-nbr</u>	name	other-data
C1	Brown
C2	Peters
C3	Morris

<u>slsprsn-nbr</u>	<u>cust-nbr</u>
S1	C1
S1	C2
S2	C3

Figure 11. Sample Data: Non-Embedded Foreign Keys.

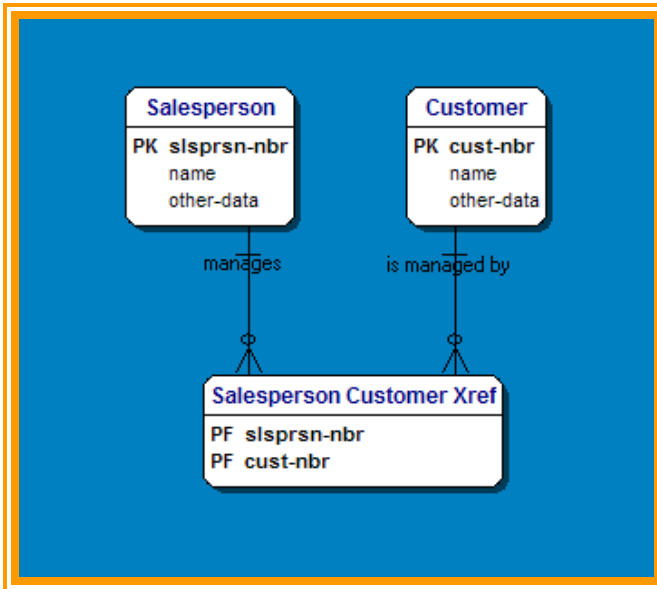


Figure 12. Data Model Diagram: Non-Embedded Foreign Keys.

In this case, neither of the two related tables contains an embedded foreign key. Instead, the foreign key for each of the two tables is contained in a third table. For this reason, I call these foreign keys “non-embedded”. It is the second of two syntactical forms that implement relationships in relational databases.

Our first two key problems were instances of *semantic homonyms* – one syntactic component implementing multiple semantic components. This third key problem is a problem with *semantic synonyms* – two syntactic components implementing one semantic component – and with the early binding of that semantic component to one or the other of the two syntactic components.

With our first two key problems, *one* syntactic component implemented *two* distinct semantic components – primary keys which both identified and described, in the former case, and foreign keys which both identified and related, in the latter case. With this third key problem, *two* syntactic components (embedded foreign keys, non-embedded foreign keys) implement *one* semantic component, that of a relationship between a pair of rows.

The problem is this: every relationship is implemented with one or the other of these two structures. But when the semantics of a relationship changes in specific ways, the relationship must be un-bound from its then current structure and bound to the other one.

Briefly, it happens like this. Relationships, as we have seen, have both a minimum and a maximum cardinality. When the minimum cardinality of a relationship changes, no change is required in the syntax of its implementation. Instead, the foreign key used to implement the relationship is either changed from nullable to non-nullable to change a relationship from optional to required, or else from non-nullable to nullable to change the relationship from required to optional.

Actually, this oversimplifies things a bit. For one thing, there is no minimum cardinality choice to make when the relationship is implemented by means of an associative table. Both foreign keys in an associative table are required; neither can be null. The “semantics of optionality”, if the relationship is indeed optional (as it almost always is), is supported by the existence or non-existence of the relationship instance – the row in the associative table – not by nulls in foreign keys. If the relationship exists between a pair of rows, then there is a row in the associative table which combines foreign keys for each of the related rows. Otherwise, there is no such row.

So minimum cardinality, as well as maximum cardinality, is expressed in two different ways, the semantics of a relationship being required or optional for a participant expressed in two different syntactic forms.

For a second thing, a nullable foreign key only establishes minimum cardinality for the *child* row in a parent/child relationship.² For example, if an invoice header foreign key, in an invoice line item table, is nullable, that means that invoice line items can exist which are not related to any invoice header (a very untypical situation, of course). But if the foreign key is not nullable, that means that every line item must be related to (at least one) an invoice header. And since there is only one such foreign key column in the line item table, that means that each line item is related to at most one invoice header. Combining “at least one” with “at most one”, we get the full minimum cardinality of the relationship, for the child rows: each child row must be related to “exactly one” parent row.

Note here the two very different ways that these two semantic features of relationships are enforced. Each line item row is related to *at most* one header row. How is this enforced? By the simple fact that each line item row has only one column to hold an invoice header foreign key.

The other semantic feature is that each line item row is related to *at least* one header row. And how is this feature enforced? The first part of the enforcement is to make the foreign key column non-nullable. This means that the column must contain a value. The second part of the enforcement is the referential integrity constraint. This insures that the value in that column, for every line item row, points to a row in the header table. Neither piece of the mechanism that enforces the “at least” semantics requires any programming. Both are

²

A “child” row in a one-to-many relationship is a row on the “many-side” of the relationship, and a “parent” row is a row on the “one-side”. For example, if one invoice header can be related to many invoice line items, the parent is the header row, and the children are the line items rows.

“declared” to the DBMS, and then enforced by code internal to the DBMS itself.

What about the parent row, however? It, too, has a minimum cardinality. Our invoice header row may or may not require at least one line item row in order to exist. But relational theory and relational DBMSs provide no way to express or enforce the difference. If it is a business rule for our company that an invoice must have at least one line item, then developer-written code must ensure that no line-less invoice headers are added to the invoice header table. This requires code which makes the insertion of a new header row, and the insertion of its first line item, an atomic transaction. It also requires code which makes the deletion of the last line item for an invoice (should such an event ever happen), and the deletion of the header for that invoice, an atomic transaction.

Figure 13. Aside: Issues With the Minimum Cardinality of Relationships.

But key problem #3 is a problem with the maximum cardinality of relationships, not their minimum cardinality. Relational theory and DBMSs lack support for the minimum cardinality of parent rows in parent/child relationships, leaving that support to developer-written code. But relational theory and DBMSs provide two forms of support for the maximum cardinality of relationships, thus early-binding maximum cardinality semantics to the structures which provide their syntactic realization.

Foreign keys that implement a *many-to-many relationship* exist in a separate table whose primary key is a pair of foreign keys each of which points back to one of the two related rows. Foreign keys that implement *relationships of any other cardinality* exist in one of the

related tables. But these are two different syntactic structures, to which the semantics of relationships are early-bound. Let's see how.

The semantics, in this case, is the maximum cardinality of the relationships. For example, if a company says that only one salesperson may be assigned to any one customer, that is what is commonly known as a "business rule". It is what I would call, viewing data modeling from the perspective of semantic theory, a "semantic constraint". It expresses part of the *meaning* of the relationship, in this case that salespersons are not restricted to serving just one customer but that customers are restricted to being served by just one salesperson.

If that same company decides later on to permit multiple salespersons to be assigned to its major customers, the business rule changes. It now states that a customer can have one or more salespersons assigned to him. This is a change in semantics; the *meaning* of the salesperson / customer relationship has changed. It should not require a change in syntax, i.e. a change in the way physical data is stored.

Our astute reader might ask whether the use of the term "meaning" isn't a bit overblown here. How is a change in relationship cardinality a change in meaning? Don't we change meaning by changing the *definition* of something?

I think it's important to address this question because until it is answered, the whole paradigm of semantic theory, as applied to data modeling, will feel a little awkward. Lacking formal training in philosophy or linguistics, most IT practitioners are accustomed to thinking of meaning as something that words have, and that is expressed in the definitions of those words. A dictionary is where we go to learn what a word means,

and we learn that by looking up the definition of the word.

We will become comfortable with the language of semantics being applied to data modeling only when we can see word-meanings and dictionaries as just one special case of meaning in general. I will make my own attempt to describe this broader sense of meaning. And because so much has been written about meaning, over the course of centuries, I would be remiss to not provide at least a brief review of that material, of what others from Plato to Lakoff have said about it.

Figure 14. Aside: Is “Semantics” Overblown?

But unfortunately, it is usually difficult to make this kind of change. To change cardinality to or from a many-to-many cardinality requires a change to or from the “separate table” syntax for relationships. In this case, the change involves the following actions:

1. The foreign key must be removed from the Customer table.
2. A new “associative” table must be added, whose primary key is (a) a foreign key to the Salesperson table, plus (b) a foreign key to the Customer table.
3. The Customer table must be unloaded and its schema altered.
4. The new associative table must be added.
5. Code must be written to load the new table, and the altered Customer table, from the unloaded data.

6. All SQL and procedural accesses to the database that referenced customer data must be rewritten.

Figure 15. What Must Be done When Relationship Cardinalities Change To or From Many-to-Many.

And all this is completely unnecessary.

To summarize: the problem is that using foreign keys to make relationships requires two different syntactical structures, depending on the cardinality of the relationship. Those structures are (a) the associative table; and (b) the “embedded” foreign key, i.e. the foreign key which resides in one of the related tables.

But the cardinality of a relationship is the most important aspect of its *semantics*, indeed the only aspect of its semantics which the DBMS can enforce. For example, if only one of a pair of relationships can apply to any row in some table, the DBMS *cannot* enforce this semantics, i.e. it cannot prevent both foreign keys from being referentially valid.

Another part of the semantics of relationships is contained in the labels attached to the relationships in the data model. In this case, the labels are “serves” and “is served by”. These labels, of course, don’t make it into the physical DBMS schemas. They are just labels on diagrams.

Moreover, as all data modelers know, relationship labels aren’t very important. If they were important, we would pay more attention to them, and we certainly would not change them as casually as we do. Next week, I might change “serves / is served by” to “manages / is assigned to”. And if I do, no one will care. These are just labels on a diagram, suggestions to the reader of the diagram. In other words, although they carry semantic content, that content does not make its

way into the formalism of the DBMS schemas, and it cannot be manipulated by the formalism of SQL-implemented predicate logic.

This distinction between semantics which are expressed in formal syntactic structures and manipulated by formal logic and other mathematical transformations, and semantics which are not formalized, is of the greatest importance. If we can formalize our semantics, then we can use logic and mathematics to deduce new information from information we already know.

The computer science use of terms like “ontologies”, “taxonomies”, and “semantic constraints” all designate the *leading edge* of mankind’s attempts to formalize semantics so that the abstract machines of logic and mathematics can extract new information from given information.

Data modelers, in contrast to these computer scientists, work with a *well-established* formalism, that of relational databases. Like any formalism, it consists of well-defined structures and well-defined transformations of those structures.

It is important for us to understand the progress that mathematicians and computer scientists have made in the formalization of semantics. But it is even more important for us to do the best we can at using the formalism we work with on a day-to-day basis. And that is relational theory and relational databases.

**Figure 16. Aside: Ontologies and Data Models.
Leading-Edge and Well-Established Formalizations
of Semantics.**

Returning to the problem of maximum cardinalities, we have seen that in both relational theory and relational DBMSs, relationship semantics are early-bound to one of two structures – embedded foreign keys or associative tables. If the semantics changes, to or from a many-to-many maximum cardinality, then the implementation must change, as described in the list of six actions above.

Like a butterfly flapping its wings, this small perturbation in the implementation of relationships has had a dramatic effect on the entire IT industry, one which can be, in its destructive consequences, not unfairly compared to the butterfly's hurricane.

From the point of view of an amateur linguist, a professional philosopher, and an experienced data modeler, I believe that Dr. Codd made a mistake in replacing DBMS-internal mechanisms for identifying and relating with business data. It exposes keys to the costs and risks of change, and requires *synonyms* – two different syntactic structures – to implement relationships. I believe that this mistake has cost and continues to cost companies more than any other identifiable component of their IT maintenance budgets.