

# Avoiding the Early Binding Traps of Relational Theory.

Dr. Tom Johnston

MindfulData.com

## Introduction.

### *Relational Database Components.*

We can distinguish the following components of a relational database.

1. A table represents a type, for example a customer.
2. The rows of a table represent particular things which are instances of a type, for example individual customers.
3. The primary key of a table is a set of one or more columns whose values serve as a unique identifier of each row. Every row in a table is distinguished from every other row by means of its primary key values.
4. The foreign keys in a table are a set of one or more columns that relate the rows of that table to other rows, in the same table in the case of a recursive relationship, or, more commonly, in other tables.
5. A value in any column of a table may also describe the particular thing its row represents, e.g. the name of a particular customer.
6. A domain is a set or range of values that are all the valid values for one or more columns. Sometimes a domain is co-extensive with the data type for a column, and sometimes it is a proper subset of that data type.

The structures used to manage data in a relational database are tables, rows and columns. Every DBMS-managed piece of data in a relational database exists as a value in one column of one row of one table. Domains define the values which are valid for each column.

### ***Current and Future Relational DBMSs.***

We are in a period of relatively rapid change in relational Database Management Systems. New releases of relational DBMSs affect DBAs by providing new features that improve performance, such as federated queries. They affect designers of data marts by providing better support for star schema data cube structures. They affect SQL writers – end users or developers – by providing inheritance along a type hierarchy, recursive queries and other SQL DML enhancements. But new releases of relational DBMSs do *not* affect data modelers. There are no features in today's logical data models that cannot be found in logical data models of a decade ago.

But things are changing. A major area in which data-modeling-relevant capabilities are evolving is that of domains. Traditionally, relational DBMSs have treated values as atomic. But relational theory now recognizes, and relational products are beginning to support, non-atomic domains.

For example, a row in an invoice header table could contain one column which contained all the detail line rows for that invoice. This will obviously affect the data model. Traditionally, documents such as invoices have been modeled with two tables, a header table and a detail line table. But using non-atomic domains, only one table would be needed.

This raises an interesting question, which we will discuss later on. It is a question directly relevant to the ontologies we choose to implement in our databases.

Invoice line items, to consider the example just mentioned, are a type of thing. And as I said before, types are represented by tables in a relational database. But if invoice line items are modeled as a non-atomic domain, then types can also be represented by domains.

So the question is: when should types be represented by tables, and when by domains? Is there a substantive difference here, one based on a criterion? Or does it ultimately come down to aesthetics, to a matter of preference, of data modeling style?

**Figure xxx. Aside: Types as Tables and Types as Domains.**

Although some of today's DBMSs provide rudimentary support for non-atomic domains, I don't know of and have never heard of a single data model that uses them. In addition, the major data modeling tools I am familiar with – Erwin and Desizn for Databases – provide no support for non-atomic domains.

1. *to identify* by providing unique identifiers for rows within tables;

2. *to describe* features of the things our business is interested in (the basic job that all business data does); and
3. *to relate* by putting copies of unique identifiers in rows we wish to relate to the rows whose identifiers they are.

**Figure 2. Three Semantic Functions in Databases:  
Identifying, Describing, and Relating.**

So if we apply the Information Principle, we do away with DBMS-specific pointers and other system-created and managed links among business data. Which is exactly what has happened with relational DBMS products.

At the time, this seemed like a good idea. As designers and architects, we are always looking for simplicity. It is a surer sign of goodness in our designs than anything else – simplicity, elegance, indeed beauty. So once Dr. Codd realized that the internal mechanisms of DBMS pointers were not needed, he proceeded to work out the mechanisms by which business data could take over the roles which those pointers had been used for. These were the dual roles of identifying and relating.

But no matter what else it does, all business data *describes*. That's what business data is,

But no matter what else it does, all business data *describes*. That's what business data is, data which describes something of interest to the business. If this data is also used to identify, it is performing two functions. If it is also used to relate, it is performing a third

function. In any case, data which performs more than one function is a homonym, in the strictest semantic sense.<sup>1</sup> Let's stop for a moment and see how.

Functions (in the informal sense used here, but also in the formal, mathematical sense) provide information just as much as descriptions do. For example, relating an invoice line item row back to its corresponding invoice header role provides information because, without that relationship, we obviously don't know something. Without that relationship, when we look at the invoice header, we don't know that that line item belongs to it. Without that relationship, when we look at the line item, we don't know which invoice it belongs to.

So: if all three of these functions provide information, then using the same syntactic structure to implement two or more of them means that the structure is a homonym. Specifically: just as using one word (one physical, syntactic, object) to represent two or more meanings (logical, semantic objects) creates the semantic anomaly we call a homonym, so too does using a primary key (a physical, syntactic object) to perform two or more functions (identifying, describing and relating) create a homonym.

This is not merely an analogy. To identify, to describe and to relate are functions, as we have just seen, that convey information. So they are therefore functions with semantic content. Implementing two or more of those functions in a single syntactic element is therefore to create a homonym.

As we shall see, homonyms are bad. As are synonyms. As is ambiguity and vagueness. These are, one and all, *semantic* anomalies, and we will continue to develop less than optimal data models for as long as we don't know how to recognize and remove these anomalies.

---

<sup>1</sup> Earlier material will have introduced basic concepts of semantics like synonyms, polysemes, homonyms, antonyms, etc.

This is what Part II is concerned with, to point out and describe the problems created by these functional homonyms, as they are manifested in primary and foreign keys, and to suggest ways around them. As I see it, there are three of these “key” problems.

### ***Key Problem #1. Primary Keys Which Also Describe.***

The first problem I will describe is what goes wrong when *identifying* and *describing* are implemented in the same syntactic structure, i.e. when one “piece” of the database both identifies and describes. This piece, this syntactic structure, is the primary key. In both identifying and describing, primary keys play two semantic roles. In doing so, they are homonyms, just as much as the one word “bank” is a homonym representing either of two meanings – “the sloping sides of a river” and “a place to deposit and withdraw money”.

Business data can indeed be used not just to describe, but also to both identify and relate. This means that relational DBMSs don’t need the cumbersome internal pointers of earlier database management systems like IMS, IDMS, TOTAL and other pre-relational DBMSs. Neither is the linking mechanism for relational databases part of the DBMS “machinery”. It too provided by the business data itself. That linking mechanism is the foreign key.

Business data is used to distinguish every row in a relational table from every other row. The set of one or more columns, or one of several uniquely identifying sets of columns, is chosen as the primary key of the table. Being a unique identifier of a row, its appearance in any other row of the same table, as a primary key, is forbidden.

However, its appearance in a separate table, whether or not as part of the primary key of that table, does relate the row for which it is a primary key to the row in which it appears as a foreign key. Also, its appearance in the *same* table, but not as all or part of the primary key, relates those two rows of that single table. Finally, its appearance in a third

table whose primary key was the pair of keys for the related rows, also relates those two rows.

This appearance of a unique identifier elsewhere in the database is called a “foreign key”. By these means (“means”, in the plural; for note that, as we have just seen, the term “foreign key” refers to several different implementation mechanisms), the functions of uniquely identifying a row in a table, and of relating that row to any number of other rows, can be implemented without resort to complicated DBMS pointer mechanisms.<sup>2</sup> By these means, related data can be gathered together using set-at-a-time operations.

Since (non-surrogate) primary keys are made up of business data, it follows that, when what that data describes changes, the primary keys must change also. But the values in these primary keys may exist in any number of other tables (or even in other rows and columns of the same table), as foreign keys. So the change in one value must be propagated to any number of rows, in any number of tables. As we shall see, this can become an extremely expensive process.

This is a basic observation. In brief:

Business data describes.  
When what it describes changes, the data must change.  
But changing primary keys is expensive  
because it requires changes to the related foreign keys.  
It is also unnecessary.

---

<sup>2</sup> “Function” is being used here in the non-mathematical sense of “role” or “purpose”. It is not to be confused with mathematical functions, the ones alluded to in the phrase “functional dependency”.

### **Figure 3. The Problem with using Business Data in Primary Keys.**

When business data is used in primary keys, we have what practitioners call an “intelligent key”. So the way to avoid key changes is to avoid intelligent keys. More precisely, the solution is to keep intelligent keys (which I will call “semantic keys”), but not to use them as primary keys (which I will call “syntactic keys”). Practitioners have been doing this for at least a decade. We call it using “surrogate keys”.

I will make three recommendations for improving on this “best practice”:

1. Take it seriously. If it’s good to use surrogate keys for some tables, it’s better to use them for all tables. If it’s good to use them for an Invoice Header table, it’s also good to use them for the related “child” table – the Invoice Line Item table. If it’s good to use them for a Salesperson and a Customer table, it’s also good to use them for the associative table between them.

2. Entity integrity is a good idea. But if having keys which are unique within a table is a good idea, having keys which are unique across all tables in a database is a better idea. For reasons I will explain later, I call this “object integrity”. And having keys which are unique across an entire “namespace” of multiple databases, is the best idea of all.

Some practitioners may suspect that I am talking about GUIDs – globally unique identifiers. And I am. I'll describe why they are important for relational databases, not just for object-oriented systems. And I'll describe a special kind of GUID which I recommend for use with relational databases.

3. Specific rules must be followed to create well-formed semantic keys, and to avoid any cross-over from semantic onto syntactic keys. I will discuss these rules later on.

**Figure 4. Three Recommendations for Improving Surrogate Keys.**

### ***Key Problem #2. Foreign Keys Which Also Identify.***

The second problem I will describe is what goes wrong when *relating* and *identifying* are implemented in the same syntactic structure, i.e. when one piece of the database both identifies and relates. This piece, this syntactic structure, is the foreign key component of a primary key. And just as traditional primary keys, i.e. ones made up of business data, are homonyms, so too are their corresponding foreign keys. Primary keys are homonyms because they both identify and describe. Foreign keys are homonyms because they both relate and identify.

Such foreign keys, i.e. ones that are part of primary keys, have come to be known as keys which create an “identifying relationship”.<sup>3</sup> And the way to avoid the problems they create, of course, is to eliminate them. This means that we should never use a foreign key as part of a primary key.<sup>4</sup> No foreign key should ever establish an “identifying relationship”.

An astute reader, at this point, one who is picking up on the language of semantics, as it applies to data modeling, might want to raise an objection. It would go something like this:

“Ok. Maybe you will present us, later in Part II, with a good argument for keeping foreign keys out of primary keys, the ones you call syntactic keys. But surely it is part of the *semantics* of many tables that part of what distinguishes the rows of those tables from one another is a relationship to other rows. How will you preserve those semantics if you exclude foreign keys from primary keys?”

For example, two invoice line items could be for the same product, in the same quantity, at the same price, etc, etc. They could each be line-nbr 3 on their respective invoices. What business data makes them unique? What is their semantic key?”

I usually have a mental image of the columns of a table as existing left-to-right, although schema definitions, and the major data modeling tools, list columns top-to-bottom. So I think of our Invoice Line Item Table as looking something like this:

---

<sup>3</sup> A foreign key can be a complete primary key, all by itself, but only in special cases. One special case is a one-to-one relationship in which the primary key of one table is also used as the primary key of the related table. The other special case is the subtype relationship, in which the primary key of a subtype table is a foreign key from its immediate supertype table. But by far the most common cases of foreign keys being used to identify is when a foreign key is *part* of a primary key.

<sup>4</sup> As previously noted, a foreign key can be a complete primary key when it is the primary key of either a subtype table, or a table in a one-to-one relationship with another table.

The top row contains the column headings. The second row indicates the data type of each column. The third row is the one that different values appear in. It is like a “viewer” in which rows of the table appear, one at a time, as the cursor scrolls through them.<sup>5</sup>

As our cursor scrolls through each row of the table, a different set of values appears in the viewer. Let’s say that we have just moved the cursor to some row, and the following set of values pops up:<sup>6</sup>

Two such line items are shown in Figure 5.

Line-nbr	prod-code	qty	unit-price
Int	Char(12)	Int	Dec(6,2)
3	4X18 WM Rolls	24	\$225.00
3	4X18 WM Rolls	24	\$225.00

**Figure 5. Sample Data: an Invoice Line Item Table.**

Continuing on, our astute reader might even come up with the following suggestion. “Clearly, what distinguishes those two invoice line items is that they appear on different invoices, i.e. that they are related to different rows in an Invoice Header table. So if

---

<sup>5</sup> I will frequently drop the word “rows” when the context guarantees that no ambiguity could result. So in the sentence above, I could have said “The third row is like a “viewer” in which invoice line items appear, one by one, as the cursor scrolls through them.” In this context, it is clear that I am referring to rows in a table, not to the line items which those rows represent.

<sup>6</sup> For this example, we ignore primary keys altogether, and assume that none of the columns are primary key columns.

foreign keys are not allowed in primary keys, then something in their *semantic keys* must point to related rows in that invoice header table.”<sup>7</sup>

invc-nbr	ship-date	terms
Char(8)	date	Char(8)
MW-4413	5/17/06	net 30
CC-0337	4/5/06	net 60

**Figure 6. Sample Data: an Invoice Header Table.**

This is an astute reader indeed, one who is already beginning to think about data from the point of view of what the data *means*. And she is correct. What is wrong with my solution, at this point, is that an essential part of the very *meaning* of being a specific invoice line item is not reflected in its semantic key. What is lacking is the simple fact that a specific invoice line item is a line item of a specific invoice! Nothing in the invoice line item table refers to invoices, i.e. to invoice headers.

The most obvious candidate to carry out this referring function is the semantic key of the related row itself. In this case, the related row is a row of the invoice header table, and we may assume that its semantic key is invc-nbr, i.e. that invc-nbr is a unique identifier for that table. So adding this semantic key to the line item table gives us the following modified line item table:

---

<sup>7</sup> Question: what is the semantic key of the line item table shown in Figure 1? Answer: apparently none. Absent a (very unusual) business rule to the contrary, there is nothing to prevent two rows from occurring in this table that have the same value in all four columns, as I have shown in Figure 1. (Later, I will collect all questions and move them to the end of each chapter, with the answers as appendices at the back of the book.)

invc-nbr	line-nbr	prod-code	qty	unit-price
Char(8)	int	Char(12)	Int	Dec(6,2)
CC-0337	3	4X18 WM Rolls	24	\$225.00
MW-4413	3	4X18 WM Rolls	24	\$225.00

**Figure 7. Sample Data: Semantic Key Association of Headers and Line Items.**

This solution gives us a key / foreign key mechanism, but one implemented at the level of *semantic* keys, and completely distinct from the level of DBMS-managed *syntactic* keys.

It turns out, however, that this is a self-defeating recommendation, one which re-creates the original problem but now at the level of semantic keys. Moreover, it does not simple re-create the original problem. It makes it worse, for now entity integrity (as the uniqueness of semantic keys) and referential integrity (as the validity of references) cannot be enforced by the DBMS. They become “do-it-yourself” integrity constraints.

However, there is a solution that solves the problem without creating other problems. We will discuss that solution later in Part II.

### **The Foreign Key Ripple Effect.**

But precisely what problems am I referring to? What are the problems that result from using foreign keys in primary keys? The basic problem is the one I have called “the foreign key ripple effect”. Let’s suppose that we are using descriptive data in our primary keys. In that case, when some of those key values change, we will have to go to every table that contains a foreign key with that value, and make the change there, as well. Moreover, this must be implemented as an atomic transaction, i.e. the change must be

made to the primary key value, and also to all the foreign key values, or else made to none of them. A simple SQL UPDATE statement will not do this.

Let's further suppose that, in the case being considered, five other tables have foreign keys pointing back to the original table. In that case, a total of six tables are affected by the change. But now let's suppose that three of those related tables use that foreign key as part of their own primary keys. Now the impact of changing a primary key value can "ripple out" to tables not just directly related to the original table, but in addition to tables which have a foreign key pointing back to one of those three tables (which in turn have foreign keys pointing back to the original table). These last-mentioned tables are tables affected by the original change through one level of indirection, i.e. through one intermediate table. But it should be clear that a ripple effect need not stop there. It can affect tables related to the original table by two, three or more levels of indirection.

Here's another way to look at this problem. *Foreign keys effectively denormalize a database.* If invc-nbr MW-4413 occurs both as a primary key, and as a foreign key in ten related line item rows, then there are eleven occurrences of the value "MW-4413". And like any duplicate data caused by denormalization, updates to a primary key and all of its foreign keys must be applied as all-or-nothing atomic transactions.

We have been taught not to think of foreign keys as denormalizations. But if they create duplicate values, and if they can be updated, then they cause the same problems that denormalization, as traditionally understood, cause. The updates must be applied to all occurrences of the data, and they must apply all-or-nothing.

If we IT practitioners continue to use relational databases, we will have to use foreign keys. There is no way around it. But if we avoid using primary keys that can change, then we have, ipso facto, also avoided using foreign keys which can change. And it is the changing that causes all the problems. Duplicate data is no problem at all if it is data that never changes.

Our astute reader might point out, however, that there is no reason why surrogate keys – which I have indicated are part of the solution I will propose – can't change as well. While that is true, it misses the point. The point is that there is no business reason for changing surrogate keys, because they don't describe anything. If surrogate keys carry no business meaning, then they are merely mechanical identifiers and pointers, part of the physical implementation of a database, but not part of its descriptive content. And if the business doesn't ask us to change surrogate keys, then being aware of the problems which arise when we do change them, we are certainly not going to choose to change them ourselves.

Back when hierarchical and network DBMS such as IMS, IDMS and TOTAL dominated the marketplace, we IT developers simply dealt with the fact that to relate data in different structures, we had to create and manage *pointers*. These pointers were not business data; they were purely an implementation mechanism. So at this point in history, we had a clean separation between business data (semantics, logical stuff) and DBMS pointers (syntax, physical stuff).

I look back on this, and think it was a good thing. Dr. Codd looked on it and thought it was a bad thing. This is the over-arching theme of Part II.

So our conclusion is:

Codd's Information Principle is wrong.  
DBMS mechanisms for identifying and relating  
are not the problem. They are the solution.

**Figure 8. Conclusion: Codd's Information Principle is Wrong.**

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

The third problem I will describe is what goes wrong when more than one syntactic structure is used to implement one function. 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.

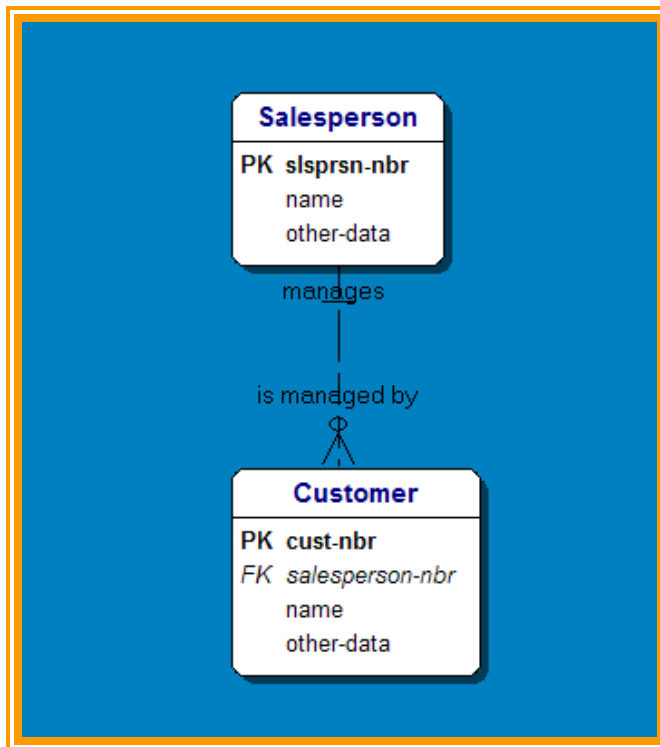
<u>slsprsn-nbr</u>	name	other-data
<u>Char(10)</u>	Char(25)	.....
S1	Smith	....
S2	Jones	....

<u>cust-nbr</u>	name	<i>slsprsn-nbr</i>	other-data
<u>Char(10)</u>	Char(30)	<i>Char(10)</i>	....

C1	Brown	S2	....
C2	Peters	S1	....
C3	Morris	S2	....

**Figure 9. Sample Data: Embedded Foreign Keys.**<sup>8</sup>

*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.



**Figure 10. Data Model Diagram: Embedded Foreign Keys.**

<sup>8</sup> In Sample Data tables, my notational conventions are these. Primary keys are the left-most columns, and are underlined. Foreign keys are italicized.

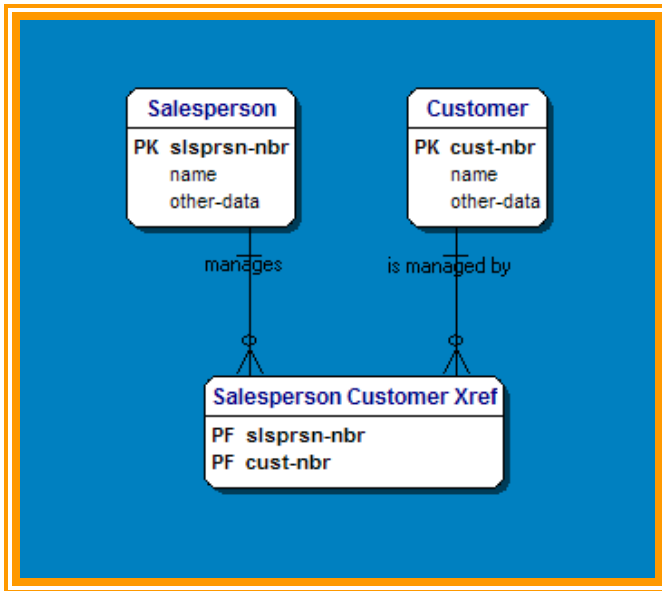
Now let us suppose that our company has found that some customers have become so important that we need to assign two or more salespersons to them. Since one salesperson can still manage several customers, this change means that there is now a *many-to-many* relationship between customers and salespersons. 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, and colloquially called an xref table. It looks like this:

<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	.....

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

**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.<sup>9</sup> 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 is expressed in two different ways, the semantics of a relationship being required or optional for a participant expressed in two different syntactic forms.

---

<sup>9</sup> An allusion to material which will appear earlier in the book, but which is not part of this sample chapter.

For a second thing, a nullable foreign key only establishes minimum cardinality for the *child* row in a parent/child relationship.<sup>10</sup> 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.<sup>11</sup>

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

---

<sup>10</sup> 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.

<sup>11</sup> “Asides” are another proposed structural feature of this book. They are appendix-type material that is brief enough that we *can* include it in-line with the text, and pertinent enough that we *should*.

requires any programming. Both are “declared” to the DBMS, and then enforced by code internal to the DBMS itself.<sup>12</sup>

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.<sup>13</sup>

**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.

---

<sup>12</sup> The declarative vs. procedural distinction will have been clarified in an earlier discussion.

<sup>13</sup> I’m undecided about two things here. First: should this aside be enlarged and made an appendix? Or perhaps covered in earlier chapters that introduce data modeling (the preliminary chapters for the material on normalization). Second: even if left in-line, would this material benefit from its own set of illustrations?

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" (or, to use a less bellicose term, a "semantic *feature*"). 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.<sup>14</sup>

**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:



---

<sup>14</sup> Haven't figured out yet where to put this material I am referring to. Probably prior to this set of chapters. It's certainly too long for an appendix. (I anticipate 15 – 20,000 words.) And also certainly too important to a book which has as a core theme the practice of data modeling understood from the discipline of semantics.

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 seven 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 ersatz 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.