

Keys in Relational Databases: Theory and Practice. Part 2.

Dr. Tom Johnston
MindfulData.com

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”.¹ 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. 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 2, 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?”

¹ 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.

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?”

Two such line items are shown in Figure 5. For purposes of this example, we ignore primary keys altogether, and assume that none of the columns are primary key columns.

line-nbr	prod-code	qty	unit-price
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 foreign keys are not allowed in primary keys, then something in their *semantic keys* must point to related rows in that invoice header table.”²

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

Figure 6. Sample Data: an Invoice Header Table.

² 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.

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:

invc-nbr	line-nbr	prod-code	qty	unit-price
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. Remember, none of these columns are primary key columns.

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 simply 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 2.

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 should be implemented as an atomic transaction, i.e. the change should 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 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 that 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 2.

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.

{11/2007. On re-reading this stuff, I now think that this statement is misleading. It is too confrontational. Instead, I should have said: Codd's Informational Principle is correct as a principle about semantics. But it is not correct as a principle about syntax. This will be one of the themes of my talk on primary keys at the March, 2008 DAMA Conference.}