Keys in Relational Databases: Theory and Practice. Part 5.

Dr. Tom Johnston
MindfulData.com

Propagation of Primary Key Changes to Foreign Keys.

The primary key values in an enterprise's most important tables are ubiquitous. Besides their appearance as primary keys, they occur elsewhere as foreign keys.

1. They occur in other columns of other rows of the same table in which they are primary keys.

2. They occur in other tables in the same database.

3. They occur in other company databases, such as the enterprise data warehouse, downstream data marts, operational data stores, and other on-line or batch transaction processing systems.

4. They even occur in the databases of other companies the enterprise does business with.

Figure 26. The Ubiquity of Foreign Keys.

Referential integrity – the validity of foreign keys – is usually enforced only within the physical database that contains the referenced table itself. But foreign keys referring to a specific table also occur in tables that are in *other* databases than the table referred to; and when primary keys change, these "non-native" foreign keys must also be changed. Moreover, foreign keys exist not just in these on-line databases, but also in off-line electronic archives—not to mention pdf files and hardcopy where, of course, they *cannot* be changed.

© Copyright 2008, Dr. Tom Johnston. Only personal, non-commercial copies, are permitted.

When Propagation is a Work in Progress.

Consider the effort to propagate changes in just one table’s primary keys to their foreign keys in that table's own database, and to their foreign keys in physically distinct databases. Ideally, a change to a primary key value should be done as one atomic transaction. This means that while the changes are taking place, no one can see the altered rows until all of them have been changed (or, in case of error, until all changes have been rolled back). In this way, queries will never take place against a database in which some of its rows have the original value as the foreign key, while other of its rows have the new value.

But transaction atomicity usually isn’t possible when large databases are involved, and is hardly ever possible when physically distinct databases are involved. So in these cases, there is a period of time—sometimes weeks and months, if many databases are involved—over which the changes take place. During this time, joins using the new foreign key values will fail for the foreign key instances which haven’t yet been changed. Joins using the original foreign key values will fail for the foreign key instances which have been changed. So consequently, the databases will report back incorrect information when queried.

This is a fact of life, as IT practitioners well understand. It’s not much help to say that it shouldn’t happen, or that DBMS vendors should improve their products to eliminate the problem. Instead, we must find ways to minimize this *semantic dis-integrity*, ways to “live with it” at minimal cost.

The best thing, of course, would be to never incur the cost at all. And that means, to tip my hand a little, that we must never change a primary key value. But for now, we are considering what happens when a primary key *does* change – which is a not uncommon occurrence when natural keys are used.

If only a handful of primary key values are being changed, then we might simply require the query user to specify both the old and new values in his joins, until we inform him that all old values have been replaced by new values. For queries embedded in programs, we might hardcode the old and new values, replacing an “equals” clause with an “in” clause followed by a list. However, in the case of wholesale changes to thousands or millions of primary key values, this is impractical.

© Copyright 2008, Dr. Tom Johnston. Only personal, non-commercial copies, are permitted.

But when do such wholesale changes occur? The answer is that they occur when *intelligent* keys outgrow their own data type constraints. This special case of changing primary key values – the case that takes us from a manageably small number of key value changes to a very large number of key value changes – will be discussed below, in the section "Intelligent Keys".

In such cases – those in which the change in a primary key value cannot be propagated to its foreign key occurrences as an atomic transaction, and in which it is impractical or impossible to hardcode all old and new value pairs into our SQL statements – some kind of cross-reference table will be needed until all foreign key instances have been changed to the new values. Each row in such a cross-reference table contains an old value paired with a new value. Cross reference tables contain one row for each foreign key that must be changed from an old value to a new value.

But cross-reference tables don't *do* anything by themselves. After they are created, queries that contain joins based on the foreign keys must be altered to add an intermediate join to the cross-reference table. This can be a very expensive process, depending on how many such queries there are, and also on how difficult it may be to find them all.

For example, besides all the queries managed by the IT department, what about all the queries written by "power users", either directly in SQL, or indirectly with such products as Crystal Reports? As long as any of these queries, which are not under the control of the IT department, are not changed to use the cross-reference table, they will return incorrect results during the hours, days, weeks or months that old values are being replaced by new values.

But finally, at some point, the propagation of all new key values will be complete. At that point, we have a choice. We can either leave the cross-reference table in place, or we can remove it. Leaving it in place is generally not a good idea. It adds an extra join to all queries that use it, a join that is now unnecessary. But if we discard the table, we must change all the affected queries back to their original form, and stop using the cross-reference table.

© Copyright 2008, Dr. Tom Johnston. Only personal, non-commercial copies, are permitted.

Again and again, with one database after another, this lengthy and expensive process must take place, until all the foreign key instances of all the primary keys of that original table have been changed. Let's summarize what has to be done when changes to primary key values cannot be propagated to their foreign key occurrences as an atomic transaction, and when hardcoding old value / new value pairs is undesirable or impossible.

1. A cross-reference table must be created, which pairs the original value with the new value that replaces it.

2. All SQL and procedural code that includes joins based on that primary key and its foreign keys must be altered to include the cross-reference table as an intermediate table in those joins.

3. After all value changes have been propagated to all foreign keys, the SQL and procedural code must be changed back to their original forms.

4. The cross-reference table may then be dropped.

Figure 27. Steps in Changing Key Values When Propagation Is a Work in Progress.

It is steps 2 and 3, of course, that are the most costly and time-consuming. Consider a Fortune 100 company, which decides to change the primary key on its Customer Master table, thus requiring changes to the primary key value for all of its customers. Let's suppose that they have a million customers. Since customer is so important a concept, in any company, we can be sure that there are literally dozens of other tables that have a foreign key to the Customer Master table. Being a world-wide company, with divisions that manage their own set of physically distinct databases, there are probably many dozens of tables that have foreign key values which reference the corporate Customer Master table (but which cannot be managed by the DBMS as foreign keys because they point outside the database in which they occur).

© Copyright 2008, Dr. Tom Johnston. Only personal, non-commercial copies, are permitted.

An IT project to manage this change could not be completed in under a year, no matter how many resources were assigned to it. The magnitude of the project is such that corporate management may decide not to undertake it, and to direct the rest of the company to find ways around the problem. Yet the impetus for the changes certainly didn't come from the IT department. It came from the business community, who certainly wouldn't have asked for the change unless there were very good reasons for doing so. From this we can be sure that the decision not to make those changes will be a costly one.

I said earlier that these kind of costs are the greatest single source of IT costs, of whatever kind. I think that the material that follows will support this claim. But regardless of the precise magnitude of the costs, the situation itself is inherently absurd! The business community requests an important change. The IT department has designed its Customer Master table so that the requested changes are *very* expensive, indeed sometimes *prohibitively* expensive.

The absurdity lies in the fact that it doesn't have to be this way. Instead of a project that will take over a calendar year and any number of man-years to complete, the requested change could have been completed in a few weeks. The source of the problem is that, in such cases, the semantics of a company's business key for customers is *early-bound* to the syntax of primary and foreign keys in relational databases across the company.

Two Further Complications – One Real, One Not.

Many changes to keys involve a change to the syntax of the key. The syntax of a key can change in either or both of two ways. First, the data type or length of a primary key column may change. Secondly, the number of columns making up the key may change. These syntactical differences make changing primary keys even more difficult to manage. They are also the prime cause of *mass* changes in key values, the kind of changes that, as explained above, may take months or even years to complete.

However, we really only need to worry about the first kind of syntactic change. The reason is that if columns are added or dropped from a primary key, we aren't talking about a change to the key of a given table. Although that may be true from a physical, DBA point of view, from a semantic point

© Copyright 2008, Dr. Tom Johnston. Only personal, non-commercial copies, are permitted.

of view what we're talking about is the *replacement* of one table with another table. Let's see what the difference is.

Suppose we have a Customer Master table, with cust-nbr as a single-column primary key. And now suppose we add a date to the primary key. Does the table still mean the same thing? In other words, does it still represent customers?

No, it does not. It now represents snapshots, or else versions, of customers.[1] Each row describes the customer *as-of* the indicated date. So there can be any number of rows for the same customer.

What a table *means* is the (usually) time-varying set of whatever it is that is represented by rows of that table. This statement is true of all tables. So what a Customer table means is whatever can be represented by rows of that table. And if our table is well-named, what each of its rows represents is a customer.

However, this gets things backwards. It uses what is to be explained as the explanation. We have *not* explained what a Customer Master table means by saying it is a table of customers, because this explanation tells us nothing more than what the commonsense understanding of the word "customer" conveys. Instead, we must think of our Customer Master table as a mathematical set, and its rows as the members of that set. To say what that table means then is, precisely, to state its set membership criteria. This means to state the necessary and sufficient conditions that must be true in order for us to (a) add a row to the table, and (b) remove a row from the table. As most of us who have worked in more than one company know, these criteria are hardly ever exactly the same from one company to the next. Which is to say that the meaning of a Customer Master table, i.e. the specific meaning that "customer" has for the business community that uses that table, is highly variable.

[1] Later on, we will clarify the distinction between snapshots and versions. Basically, snapshots occur at pre-planned, regular intervals. Versions are taken as and when changes important enough to keep track of occur. Important changes can be missed when history is recorded with snapshots. Trending over time is less accurate when history is recorded with versions. *{11/2007. Snapshots and versions are discussed in an ongoing series in DM Direct that I am co-authoring. See my name under the author index at DMReview.com. There are also links to these articles on my website Publications tab.}*

© Copyright 2008, Dr. Tom Johnston. Only personal, non-commercial copies, are permitted.

And significantly, as we shall see when we examine the comp-sci issue of semantic interoperability, these various meanings of the word "customer", these various ways in which, from company to company, the term is differently used, form what we can visualize as a set of overlapping circles – a Venn diagram. If these overlapping circles have an area common to all of them, we are in the realm of standard Aristotelian definitions, and on solid ground. This is illustrated in Figure 28.

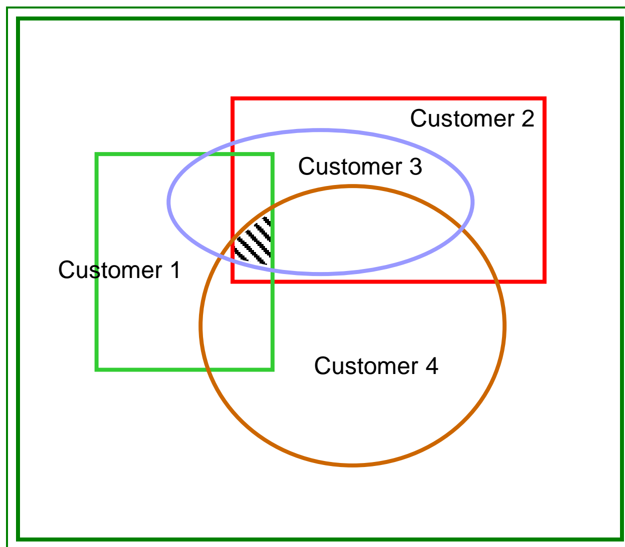

Figure 28. Venn Diagram of Aristotelian Definition Semantics.

Figure 28 shows four sets, each one corresponding to one definition of "customer". Here we mean "definition" in the formal sense of set membership criteria. The shaded area is the one area that all four sets have in common. It represents a (proper) subset of all the four sets of set membership criteria, specifically of those criteria which all four definitions have in common. Exactly how such definitions make semantic interoperability possible across all four companies' Customer Master tables, is a topic for Part 2.

However, sometimes definitions do *not* converge on a common set of semantic components. It is quite possible that our four companies all have what they call Customer Master tables, but that an examination of the set membership criteria shows that there is nothing they all have in common. This is illustrated in Figure 29.

© Copyright 2008, Dr. Tom Johnston. Only personal, non-commercial copies, are permitted.

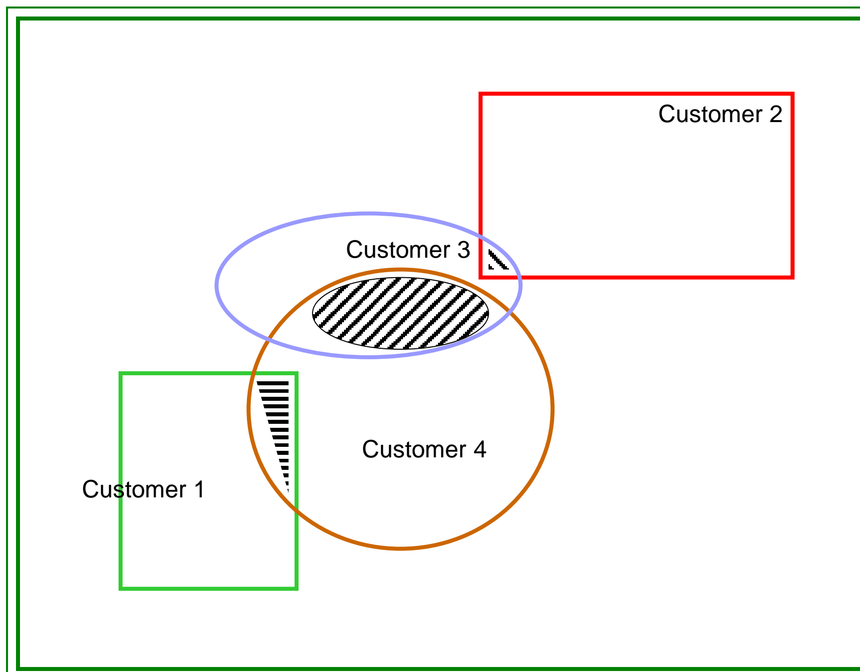

Figure 29. Venn Diagram of Wittgensteinian Family Resemblance Semantics.

Here we see (a) a small overlap in the definition of "customer" between Customer 2 and Customer 3, (b) a somewhat larger overlap between Customer 3 and Customer 4, and (c) a third overlap between Customer 4 and Customer 1.

This is a situation in which a set of definitions have what Wittgenstein called "family resemblances" among them, but no standard definition that applies to all of them. Family resemblances, as we shall see in Part 2, create difficulties for semantic interoperability.

Consider, as a second example, adding a market region code to the primary key. Is the original Customer Master table – the one that originally had only cust-nbr as a primary key, still a table of customers? No, it is not. For with this pair of columns as the new primary key, we could have two or more rows in the table with the same cust-nbr. If we did, what would each of those rows mean? What would each one stand for?

© Copyright 2008, Dr. Tom Johnston. Only personal, non-commercial copies, are permitted.

Each would stand for that customer *in that region*. If no other changes were made to the table, it would then not be normalized; it would contain a large amount of redundant data. For example, the table probably has corporate address columns in it. Well, the corporate address will be the same for the customer no matter what region he is associated with. So if corporate address columns remain part of this table, the corporate address for a customer will be repeated *n* times, where *n* is the number of regions which that customer shows up in.

This is pretty familiar stuff to most data modelers. It is, of course, a second normal form violation, a case in which columns of a table are dependent on part of the primary key, but not on all of it.

But let's not forget the point I was illustrating. When you add a column to the primary key of a table, you have changed what that table *means*, i.e. what its rows refer to. The same is true if you remove a column from the primary key of a table. Semantically, it just isn't the same table anymore.

Therefore, the only syntactic changes to primary keys that we are concerned with are changes to the data type and/or length of one or more of the columns of that primary key. And in nearly every case that I know of, syntactic changes are made because the primary key is an intelligent key.

© Copyright 2008, Dr. Tom Johnston. Only personal, non-commercial copies, are permitted.