

Keys in Relational Databases: Part 7.

Dr. Tom Johnston
MindfulData.com

Entity Integrity and Object Integrity.

However, surrogate keys have their own problems. More precisely, “plain vanilla” surrogate keys have their own problems. The basic problem with them is that, like primary keys which are also business keys, surrogate keys satisfy the *entity integrity* constraint but do not satisfy what I will call the *object integrity* constraint.

Entity integrity is the constraint that the primary key values in a relational table must be unique within that table. Object integrity subsumes entity integrity under a stronger requirement. It requires that the primary key values across a given set of tables must be unique both within its own table, *and also* across all those other tables. No row, in any table in the set, can have a foreign key value identical to any other row.

The entire set of tables, across which a primary key value must be unique, forms a *namespace*. This namespace might consist of all the tables in a single database. But the larger the namespace, the greater the benefits of object integrity. So a business would do well to include all the tables in all of its principal databases in its initial namespace. Later, a namespace shared with the enterprise’s principal suppliers, partners and customers could be created, at little incremental cost.

Today's RDBMSs do not enforce object integrity. Consequently, a database can contain two or more rows (in different tables) which happen to have the same value for their primary keys.

It happens this way. Sometimes, the primary keys of two or more tables in a database have the same "syntax" (*i.e.* the same data type and length) and also have identical or overlapping domains (permissible values). When we have these two conditions, the primary key of a row from any of these tables may have the same value as a primary key from a row of any of the other tables. If the keys in each of a pair of tables are created by a sequence number

generator which increments by 1 each time, then if the rows in those two tables have primary keys from 1 to \underline{n} and 1 to \underline{m} , where \underline{n} and \underline{m} are the maximum number of rows in each of the tables, then there will be \underline{j} cases where the key values are identical (i.e. where they “collide”), where \underline{j} is the lesser of \underline{m} or \underline{n} .

Enforcing object integrity instead of just entity integrity is one of the ways in which relational databases must become more object-like. Entity integrity is not a strong enough semantic constraint. The object integrity constraint for primary keys is included in the SQL99 standard, although the standard does not require it to be used. So it will be several years before its implementation is both ubiquitous and standard. And in the meantime, expensive reengineering projects are in planning or are already underway.

Enterprise Keys.

I call keys which satisfy the object integrity constraint *enterprise identifiers*. These are primary keys which have all the semantic features of object identifiers ("OIDs") which, in implementation, are known as GUIDs (globally unique identifiers). The object integrity constraint should replace the entity integrity constraint which all today's relational DBMSs support, as I am about to demonstrate.

What are these features of relational primary keys that can bestow object integrity on them? There are three of them.

1. There must be no descriptive information in the key.
2. The key must provide unique identification across an entire database, or even across multiple databases—not just across the rows of a single table.
3. There must be no type information in the key. In object contexts, this means that the key must not indicate the class of the object. In relational contexts, as we shall see, this means

that it must be possible to create supertype/subtype hierarchies without requiring changes to the syntax of the primary key of any table, or changes to the primary key values of any of the rows in those tables.

Figure 37. Requirements for Primary Keys to Satisfy the Object Integrity Constraint.

Business keys, including unintelligent keys, do not fulfill any of these criteria. Standard surrogate keys satisfy the first criterion, but not the second two. And it's their failure to fulfill the second two criteria that makes them unacceptable as semantically complete identifiers. In other words, "plain vanilla" surrogate keys are not enough. Let's see why.

Why We Need EIDs as Primary Keys.

Suppose our company has just completed two intelligent key to unintelligent key reengineering projects, one for customers and their households, and another one for the company's own organizational structures and its employees. So there are four unintelligent keys, now. They are:

- cust-nbr
- hshld-nbr
- emp-nbr
- org-nbr

The data models for these two projects are shown in Figures 38 & 39.

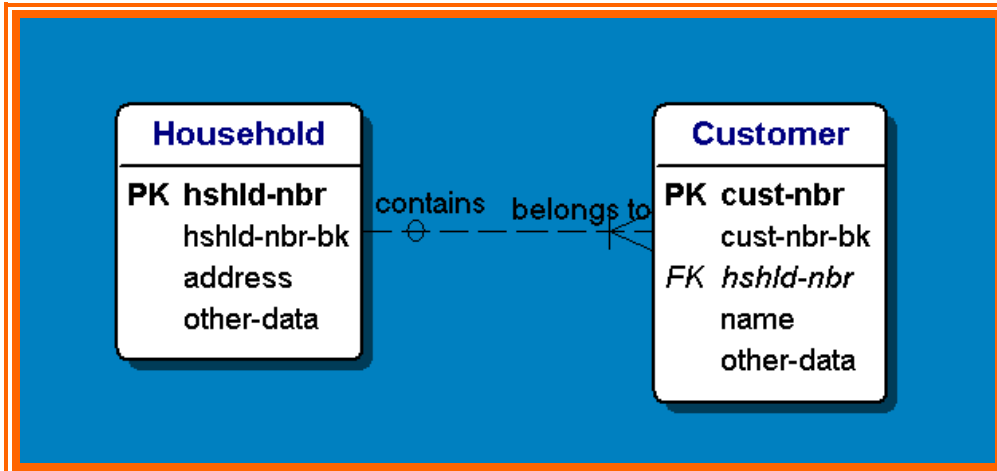


Figure 38. Data Model Diagram: Households and Customers.

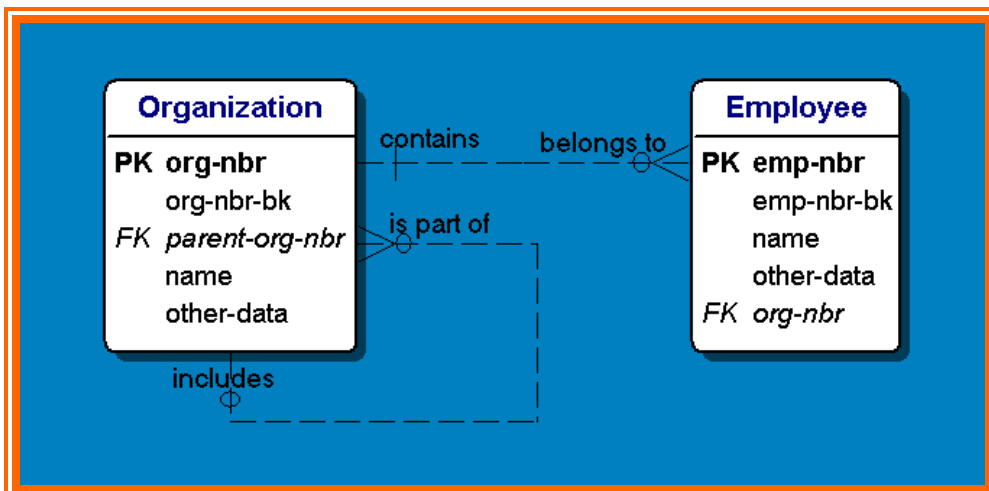


Figure 39. Data Model Diagram: Organizations and Employees.

In the Organizations and Employees model, there is a recursive relationship on the Organization table, to support a hierarchical structure of divisions, departments, groups, etc. Employees must directly belong to one and only one organization structure. In the Households and Customers model, customers don't have to belong to households, but a household must have at least one customer in it.

The four unintelligent primary keys of these tables may or may not be standard surrogate keys, having the same syntax, *i.e.* the same data type,

length and domain. There are problems with these keys in either case, so let's start by assuming that they are standard surrogate keys, and that a sequence number is used for all of them.

The problem with surrogate keys like these is that the value for any instance is unique only within its own table. It is not unique across all tables in its database, and certainly not unique across all tables in all databases in the enterprise. And, of course, this is what we are used to. Indeed, we are so used to it that we can hardly see why it is a problem.

But without object integrity, it is very difficult to create supertypes of two or more tables, at a later point in time, because doing so almost always incurs a substantial and often prohibitive foreign key ripple cost. Let's see how.

Consider our Customers and Households data model, and our Organizations and Employees data model, neither of which has any supertypes. Now suppose that we begin to look at these models from an enterprise perspective. In doing so, we may be struck by the similarities between households and organizations, on the one hand, and between customers and employees on the other. If we are, we will then generalize to supertype entities. One is a "Group" supertype, whose two subtypes are Household and Organization. The other is a "Person" supertype, whose two subtypes are Customer and Employee.

In each case, we will then move attributes and relationships that are common to the subtypes, into their supertypes. Only the attributes and relationships that belong to one subtype only, remain with that subtype. Any particular customer, for example, will then be characterized by both the attributes found in his row of the Customer table, and also the attributes "inherited" from his row in the Person table, as well as attributes inherited from Thing, the supertype of Person. Of course, if our RDBMS does not directly support inheritance, we will have to implement it with code, *i.e.* with joins between the supertype and subtype tables. If this is done to create a view, the SQL which accesses that view will have the benefits of inheritance without doing its own join.

In the model shown below, we have recognized that every table in our model represents an “object of interest” to us. This is represented by the “Thing table”, which is a supertype to all other tables in the combined model. We then move any attributes which all of its immediate subtypes have in common, from those subtypes into the supertype. In this case, there is only one such attribute: name.

Because the Thing table is a supertype to all other tables in the model, every row in every other table corresponds to one and only one row in the Thing table. This makes Thing a convenient place to add the often-used metadata columns create-date and last-update date (often accompanied by create-person and last-update person).

Combining our two models, generalizing to supertypes, and moving common attributes into supertypes, results in the model shown in Figure 40. This process is, in miniature, how one would go about creating an enterprise data model if one were starting from data models of existing systems. This is a “bottom-up” approach to building enterprise data models, and the process is called “reverse-engineering”. Of course, to build a good enterprise data model, it needs to be supplemented with a top-down, forward-engineering process which is based on both near- and long-term future requirements gathered from SMEs.

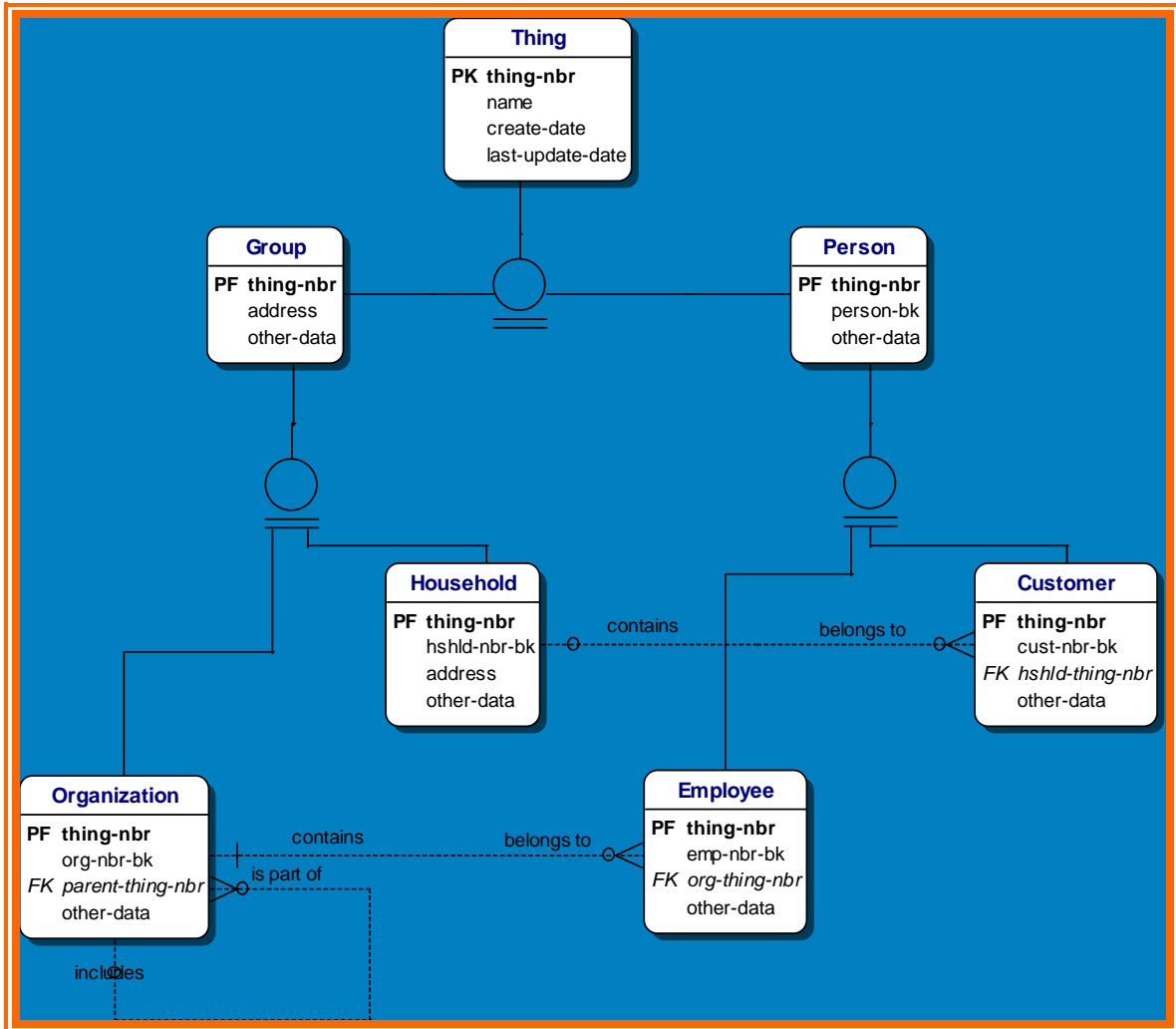


Figure 40. Data Model Diagram: Combining and Generalizing.

In a relational model, the primary keys of supertype tables are also the primary keys in each subtype table. In addition, these subtype table primary keys are also foreign keys, pointing back to the supertype. The relationship is one-to-one, required from the subtype table and optional from the supertype table. So because all other tables are either directly or indirectly a subtype of the Thing table, their primary keys are also foreign keys of that table.

Now we are approaching the heart of the matter. We have used sequence numbers to generate primary key values for each of our four original tables.

So now, when we try to bring Employees and Customers together under a Person supertype, n of their keys will collide (where n is the number of employees or customers, whichever is less). And the same problem exists for Households and Organization-Structures.

But suppose that each of our two original models had included a supertype right from the beginning—Person for one model, and Related-Organization for the other. Then there would have been no key collisions in either model. However, even if we had been foresighted enough to have included a supertype in each of our two models, the key collision problem would have arisen when we attempted to bring Person and Related-Organization together under an Object-of-Interest (Thing) supertype.

Is this a contrived example, which looks bad only because we used sequence numbers as an example? Not at all. Suppose, instead, that the keys were date-timestamps. If we did not include superatypes in either model, it would be possible, on a fast machine, with a date-timestamp precision to the second only, that some pairs of households and company organization structures would get the same key value, and similarly for pairs of employees and customers. The key conflicts would be much less frequent in this case, but they could happen and, in some environments, almost certainly would happen.

An Easy Way Out?

Earlier, we said that there were two options. One was for multiple tables to have keys with the same syntax, *i.e.* the same data type and length, and overlapping or identical domains. The other was for the tables to have keys with different syntax. If two tables which we are about to put together under a single supertype have primary keys with different syntax, then a key value conflict cannot occur. So does this solve the problem?

It does, but it raises an equally fatal problem. Suppose that the primary key of Employee is a two-attribute key consisting of (a) date of hire, and (b) sequence number. And suppose that the primary key of Customer is a date-timestamp. What, then, is the syntax of the primary key of the supertype table for these two tables? At best, it can be the syntax for the key of one of

the subtype tables. And if it is, then the other table cannot be made a subtype at all, because the primary keys of its rows could not have the same primary keys of any rows in the supertype table, and therefore could not be foreign keys of the supertype table.

Now let's see how to create enterprise identifiers, how to enforce the object integrity constraint, and how to solve these problems.