

Keys in Relational Databases: Part 8.

Dr. Tom Johnston
MindfulData.com

Three Ways to Guarantee Object Integrity in Enterprise Keys.

Root Class Tables.

One technique for creating enterprise keys is to always use a “root class” table in every database. The Object-of-Interest table in Figure 40 is a root class table. Attributes of this table, besides its primary key, are attributes that all its subtypes need, such as name, abbreviation, description, creation-date, last-update-date, and so on.

In a database with a root table, no row can be created in any table until a row is first created in the root table. Since the primary keys of all its subtypes—in our examples, Person and its two subtypes, and Related-Organization and its two subtypes—must be foreign keys back to this root table, all primary keys, in all tables, are necessarily unique across all those tables. Since they all have the same syntax, they are also type-neutral. Type neutrality is important because occasionally a data model will evolve by eliminating a table and replacing it with two or more tables. With type-neutral primary keys, rows of the eliminated table can migrate to any existing or newly created table, without altering their primary keys. (Of course, some non-primary key columns would probably have to be dropped, and others added; but the cost of doing this is trivial compared to the foreign key ripple cost.)

Database, Table and Row Id Concatenation.

A second technique is to construct primary key values according to rules defined for a namespace. This is done by combining one or two bytes which uniquely identify a table, with however many bytes are needed to create a value that is unique for each row in that table. Then the complete key will be unique across all tables in that database. To make a key unique across multiple databases, just put one or two more additional bytes as a prefix, to uniquely designate the database within the enterprise. Now each key is

unique across all tables in the enterprise. Finally, to anticipate making the key unique across all the databases of an entire supply-chain consortium of related enterprises, leave one or two additional bytes at the front of the key, for this future expansion. Finally, standards like X.500 formalize this process, and make it possible to create keys which are unique within a universal name space.

The first approach lets the DBMS do the work of enforcing uniqueness. The second approach requires code to create the unique primary key values. However, the first approach has a drawback of its own. It won't work across physically separate databases unless a two-phase commit transaction is used each time a key value is created. These transactions are needed to insure that no other database with the authority to create an instance of that type could do so concurrently and, in the process, create a duplicate key value.

With the second approach, all we need is an agreement on a prefix which uniquely identifies each database, and each table within a database, across all the participating databases. Then keys can be created in each database with the assurance that they are unique across all instances of all tables of all participating databases. Moreover, guaranteeing this uniqueness does not require a two-phase commit transaction every time a new primary key instance is created.

EID Creation Routine Id with Timestamp Concatenation.

{11/2007. As of rel 8.1, DB2 UDB has a GUID generation routine. My approach is to use an for each guid creation routine + a guaranteed unique timestamp. Guarantee is for routine to read clock on entry, and only return with a timestamp different from that initial reading. Details in another pre-publication fragment I'll put on the website later.}

Five Drawbacks to Traditional Primary Keys.

The previous section considered single-column keys. But traditional primary keys may have any number of columns in them. There are five problems with such multi-column primary keys.

1. The identifiers are unique only within one table. They are not unique within the entire database, across all the tables in that database. Nor are they unique across databases.
2. The identifiers in different tables can have different numbers of columns. So there is no common syntax for all the identifiers in a database. Nor is there a common syntax for all identifiers in all databases in the enterprise.
3. Even if all identifiers had only one column (as many do), there is no common identifier syntax—a data type plus length—for all identifiers.
4. Some identifiers implement relationships, as well as identify instances of things. They are identifiers that contain foreign keys. So if the instance changes what it's related to, its primary key changes, and so it has effectively changed its identity. But things in the real world change their relationships all the time, and don't change their identity in the process.
5. Some identifiers describe, as well as identify. They are the ones based on natural keys (intelligent or not). So if the description of the instance changes, it has effectively changed its identity. But again, this isn't faithful to how things really are.

Figure 41. Five Problems With Multi-Column Primary Keys.

As we shall see, each of these shortcomings is important, in the sense that each has a significant bottom-line cost associated with it. And as we shall also see, there is a better way to do primary keys. The cost of doing primary keys this better way is low, and getting lower. The benefits of doing primary keys this better way is to eliminate all five of these shortcomings; and the bottom-line value of that benefit is very high.

True object identifiers (in the technical, object-oriented sense) have none of these shortcomings. And the SQL99 standard defines a new kind of identifier that can be used with some kinds of SQL99 relational tables, and that also has none of these shortcomings.

But for traditional business databases, it may well be a long time before they are re-engineered to use object capabilities such as GUIDs, inheritance and polymorphism. Nonetheless, these traditional databases are constantly evolving.

Some of these changes are relatively simple, for example adding a few new columns or a few new tables. But many of these changes are not merely extensions to an existing database. They are re-engineering efforts, undertaken because over time the database's users have come to understand its inadequacies, and have begun to demand that they be fixed. For example, many re-engineering efforts these days are to replace intelligent keys with unintelligent keys.

Intelligent keys are so common in today's databases because until recently, they seemed to be a very good idea. The key was a single field, and this simplified things for the DBA and the programmer. The tables were usually physically clustered on primary key, and generally the intelligent key was a meaningful way to cluster rows together. Finally, the user knew his intelligent key so well that he could see all the multiple pieces of information at a glance; he didn't need those pieces split up into separate attributes.

But over time, the disadvantages of intelligent keys became apparent. And so, gradually, it has become conventional wisdom among today's relational modelers that unintelligent keys are better. Thus, in one company after another, one project after another is being undertaken to replace intelligent keys with unintelligent ones. Indeed, two major corporations I know of are currently engaged in enterprise-wide efforts to replace an intelligent customer identifier with an unintelligent one.

But converting over to unintelligent keys, without addressing the first four of the five shortcomings I have listed, is a costly mistake. It's costly because changing key values in both primary keys and foreign keys, and in every on-

line and off-line database they occur in, is costly. It's a mistake because for a negligible incremental cost, the other four shortcomings could be eliminated also—by using enterprise identifiers, EIDs. EIDs look like standard, single-attribute primary keys to relational DBMSs, but they provide unique identification of a single row across all tables, in all databases within an enterprise—hence the term “enterprise identifier”.

Messing Up Meaning.

Figure 42 shows a sample population for a Location table.

<i>location- eid</i>	<i>non-identifier attributes for Locations</i>
NEBL1
NEBL2
NENY1
NENY2
SEGA
SEMMI
SWDL1
SWLA1
.....
.....
.....

Figure 42. Sample Population of a Location Table.

In this Location table, we may conjecture, the first two characters of the Location identifier--”NE”, “SE”, “SW”--which we are told designate regions, probably stand for “Northeast”, “Southeast” and “Southwest”. And the last three characters, which we are told designate specific locations within those regions, probably stand for “New York 1”, “New York 2”, with

“BL”, “GA”, “MMI”, “DL” and “LA” standing for “Baltimore”, “Atlanta”, “Miami”, “Dallas” and “Los Angeles”.

In just such ways do businesses enable their internal users to quickly decipher the reports and screens produced by their computer systems.

But these days, most data modelers understand *why* we should avoid primary keys like this. One reason is that users will eventually want to change many of the values used in these primary keys. In some cases, they will even request a change to the internal structure--the data type and/or length--of those keys.

What happens, for example, when the users want to replace the existing set of two-character region codes--which constitute the first two characters of our Location key? Suppose, for example, that they want to replace the original set of four codes--“NE”, “NW”, “SE” and “SW”--with a set which includes these four, but adds to them “CA”, “MW” and “MA”, standing for, respectively “California”, “Midwest” and “Mid-Atlantic”.

The first thing the Data Administrator must do is change the codes in the on-line database--including all replicated copies of the data. Then she must go back (if she is thorough about it) and change the codes in the archived copies of the database, and in the data warehouse copies of that same data.

Even then, hardcopy will inevitably exist which shows the old region codes. And so users will have to keep in mind that hardcopy or pdf files from before the change-over date will have the old region codes in the Location identifier, while everything after the change-over date will have the new region codes. They might even need to keep a printed “cheat sheet” in their back pockets, correlating the old and new codes!

Even more confusing is that all the old codes have been retained, and folded into the set of new codes. This causes confusion because, necessarily, the geographic boundaries, *i.e.* the very *meaning* of “NE”, “SE” and the rest, had to be redefined to make geographic room for the additional codes--“CA”, “MA” and the rest. “SW” just doesn’t *mean* the same thing when it includes California as when it doesn’t.

Here's why that's such a bad thing. Consider the Location formerly known as "SWLA1", and known, after the swap-out, as "CALA1". As foreign-keys for this location are changed, one by one, to the new value, in tables spread across independent distributed databases, the totality of the set of those databases is in an inconsistent state. For until the change-over is complete, some foreign key instances of that location are designated as "SWLA1", while others are designated as "CALA1".

So until the entire set is changed to "CALA1", the database will be inconsistent. As a consequence, SQL, or code, which attempts to use a foreign key relationship to pull data together for location CALA1 will fail in cases where it ought not to fail. It will fail because some tables use one version of the Location primary key while other tables use the other version. Therefore, until the possibly months-long transaction is complete, the inconsistency is visible to the enterprise--visible in the form of incorrect results to queries which rely on the location identifier.

"Old pro" users can (and universally do) take pride in their knowledge of arcane "features" of their computer systems like this one, for it distinguishes them from more novice users. But the enterprise pays a price for such usually undocumented knowledge, as it always does. Errors will occur, for example, because of users who naturally, but incorrectly, assume that by counting all customers with "SW" in the fourth and fifth positions of the customer number, they will have a count of all customers in the Southwest region. They simply don't know that to get the desired count, they must count customers with *either* "SW" *or* "W2" in those positions of the customer identifier.

So yes, we should avoid primary keys with intelligence built into them. But the argument of this essay is that identifiers should be subject to more stringent constraints than this one. It is that all the identifiers, of all the tables in the scope of a model, should (a) be system-generated values which (b) all have a common structure (basic data type and length), and (c) are each unique across, not just the table they are defined on, but the entire set of tables included in all the databases in the enterprise. This kind of primary key is an enterprise identifier, not just an unintelligent key. This kind of primary key eliminates all five of the shortcomings listed at the beginning of this chapter, not just one of them!

