Keys in Relational Databases: Theory and Practice. Part 6.

Dr. Tom Johnston
MindfulData.com

Intelligent Keys.

Let's examine intelligent keys more closely, to see the kinds of problems they create for those who maintain databases. It is intelligent keys that most frequently cause the business community to request syntactical changes in primary keys.

Intelligent keys are, of course, quite easy and inexpensive to create. The problems and costs come when we have to change them.

For many years, intelligent keys seemed like a good idea. An example is the customer number we have already been using as an example. The first three characters of cust-nbr indicate the city of the customer's corporate headquarters. The last four are a sequence number to make the key unique.

cust-nbr	other-data
ALB-0388	....
ATL-1832	....
NYC-2453	....
ATL-9668	....

Figure 30. Sample Data: Intelligent Key Customer Numbers.

By now, data modelers realize that intelligent keys like this aren't such a good idea. The problem in this example is that when you try to create the 10,001st customer for any city, you can't do it, because there is no more

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

room in the sequence number range. One option is to do some kind of "split". The other option is to change the data type, i.e. allow one more position in the numeric part of the key.

Let's consider splits first. The advantage of splits over adding one byte to the key is that no structural, syntactic, changes to the database are required.

There are two kinds of splits that can be done.

Dumb Splits.

One kind of split lets the intelligent key get "dumber". For example, "ATL" may remain "ATL" for the existing 9,999 customers in Atlanta, and "AT2" can be used for the next 9,999 customers. But now the key isn't so intelligent anymore, because the user must remember that both "ATL" and "AT2" designate customers in Georgia. He must also understand that the distinction between "ATL" and "AT2" doesn't *mean* anything. These are just two designations for Atlanta—a pair of synonyms which both mean the same thing.

The problem is that by doing a "dumb split", we have placed a semantic burden on the user. In querying the database, he must now be cognizant of the pair of synonyms created by the split, and include both synonyms in all his queries. In this case, the synonyms are not a distinct column of data by themselves, but rather are part of one physical column of data – the first three characters of cust-nbr, indicating city.

Throughout the history of IT, system users have learned to carry sometimes considerable semantic burdens. In doing so, they are compensating for the poor semantics of their systems. This compensation, whatever its specific form, always involves resolving *semantic anomalies*.

1. Sometimes, it involves knowing where ambiguity and vagueness lurk, and knowing enough about the actual data to remove those anomalies and produce a result set with a clear meaning.

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

2. Sometimes, it involves understanding which columns of which tables are homonyms, and always including some other information in queries which use those columns, information that will in effect translate the homonym into one of the multiple concepts which it represents.

3. Sometimes, it involves understanding which columns of which tables are synonyms, and always including all the synonyms in queries that must produce all the objects of interest, however they are called.

Figure 31. Discovering Semantic Anomalies.

Here, for example, all queries and reports that look for or that count the number of customers in Atlanta must now use "'ATL' OR 'AT2'" in their WHERE clauses, instead of just "ATL". As customers in other cities reach the 10,000 mark, the problem only gets worse. When a city reaches 20,000 customers, the problem occurs again.

Smart Splits.

A second kind of split keeps the intelligence in the intelligent key. For example, "ATL" can change its meaning to designate just the city of Atlanta, and "ATM" can be added to handle customers who are in the Atlanta Metropolitan Area, but outside the city limits.

One problem with this kind of split is that the *meaning* of "ATL" has changed. For all queries and reports that ask for customers in Atlanta, we must now decide whether we want just customers inside the city of Atlanta, just the non-city metropolitan area customers, or both.

Another problem is that by now, the customer numbers for those first 10,000 Atlanta customers are embedded in the enterprise's databases; and a large number of them are going to have to change. All existing Atlanta customers who live outside the city limits will have to change to an "ATM" customer number. They will have to change wherever they occur; and this foreign key

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

ripple effect is often a tidal wave, capable of swamping a project well before it reaches its destination.

Expanding the Key.

Dumb splits should only be done as an emergency measure. Their advantage is that they can be done quickly. In the example above, we need only add "AT2" as a valid value for the first three characters of cust-nbr, and change the code which creates customer numbers to start using "AT2" for Atlanta customers, and stop using "ATL". This can be done literally overnight.

But over time, as more and more cities require multiple three-letter abbreviations, the semantic burden placed on the business user becomes more and more onerous. Experienced and careful users will probably make few mistakes in their queries. But the plethora of synonyms for "Atlanta" and for other cities will inevitably cause some users to formulate incorrect queries, ones which fail to include one or more synonyms. As a result, counts will be less than they should be. Result sets will miss out rows that should be included.

Therefore, dumb splits should be adopted only as an interim measure. What, then, is the long-term solution? This is really a topic for a case study in data modeling, so I won't go into it in any detail. But, in brief: one solution is to change the flat list of three-character abbreviations into a *hierarchy* of abbreviations. The root node of the hierarchy will always stand for a city. Leaf nodes under it will be the one of more abbreviations that each identify some of the customers in that city. The trade-off here, of course, is the complexity of queries based on a hierarchy of city abbreviations.

Another approach is obviously to switch to abbreviations that retain some semantic content, i.e. to do a smart split. But we have already seen the problems with that approach. It forces us to change the primary keys of customers whose semantics require them to have a different key.

The third approach involves no splits at all. Instead, it involves a change to the syntax of the primary key. As we have seen, this does not mean adding or dropping columns in the key. In our case, it means physically expanding the primary key, to make room for more customers in the same city. This, in

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

turn, means adding one or more bytes to the cust-nbr column, and using the extra length to permit sequence numbers higher than 10,000.

This solution has its own costs, however. One of them is that *all* foreign key occurrences of these foreign keys must be changed. These costs were described in Figures 25 – 27, and the text associated with them. But in this case, the cost is, in one sense, maximal. For in this case, it is not just selected values which must be changed. It is all values. For example, primary key "ATL-1832" must be changed to "ATL-183200" if two bytes are being added to cust-nbr, and similarly for every single cust-nbr value, primary and foreign key, in all databases in which it occurs.[1]

The semantics of these primary key values resides in their recognition and use by the business community, and in their appearance outside the confines of the database in which they occur – on invoices, in emails, in telephone calls, etc. That use is what gives them their meaning.

The problem is that the semantics have been constrained by the syntax of the cust-nbr column so that changing the semantics – allowing more than 10,000 customers per city – requires changing the syntax. That change, as we have seen, will be expensive and, because it cannot be carried out as an atomic transaction, disruptive. What's more, it is completely unnecessary.

It binds semantics to a syntax – primary and foreign keys – which is costly to change. What we need is a way to separate volatile semantics – semantics whose cost-weighted likelihood of occurring is high – from being bound to syntax which is costly to change, to being bound to a syntax – that of columns which are neither primary nor foreign keys – which is inexpensive to change.

Unintelligent Keys.

For these reasons, most data modelers now realize that we need not only "unintelligent" keys, but in fact surrogate keys – although most data modelers still use surrogate keys only in the most important tables, in

---

[1] Or change "ATL-1832" to "ATL-001832". This latter change preserves the sequence number semantics of the second part of these intelligent keys. But on the assumption that the sequence in which rows were created is not of semantic value to the business user, an assumption we made earlier, neither way of changing the value is better than the other.

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

general those whose keys are referenced by foreign keys in many other tables, but which do not themselves contain foreign key references.[2]

And so we persuade our companies to fund projects to create a new key—a new customer key, in the case of this example. In large enterprises, these projects can be very expensive. But the result is a key that is "impervious to change" because it doesn't *describe* the customer in any way. Because it contains no descriptive information, the user has no reason to request a change to it. And if the systems which access customer information are reengineered to use the new surrogate key correctly, that new key will be used in joins, but never in WHERE clauses.

Of course, the user will still want his "business key". He'll say something like "I don't know anything about these fancy surrogate keys', and I don't want to know. What I do know is that I'm not going to give up my customer number! I just want to enlarge the sequence number part from four to six positions. That should give me plenty of room to grow."[3]

The Information Technology (IT) department isn't going to win this battle. Nor should it. Instead, as part of the project to create an unintelligent key for customers, IT should keep the original customer number, and just take away its role as a primary key. The Database Administrator (DBA) should then define a unique index on the non-key customer number column, because uniqueness is still a requirement for it.

---

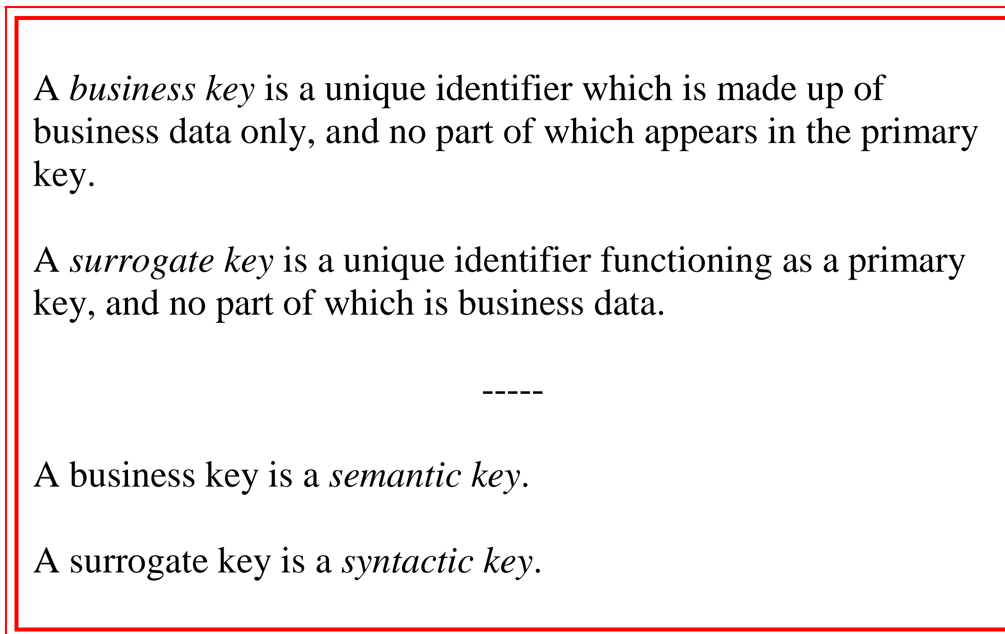
2 These turn out to be the tables called "kernel tables" in Codd's RM/T paper.

3 *{11/2007. Here I conflate two issues. Intelligent keys are a bad idea because when they reach their limits, e.g. the 10,001st customer in Atlanta, as explained earlier, the primary/foreign key changes that are required will affect all rows with those keys. The second issue is this: even if intelligent keys are not used, natural keys – keys at least one of whose component columns are business data – leave the database open to the high cost of changing primary/foreign keys. However, when non-intelligent natural keys change, it is only specific values within those keys which have to change. No change is required to the syntax of the keys, as the "intelligent split" option for resolving intelligent key changes would require. No key columns are rendered meaningless, as the "dumb split" option for resolving intelligent key changes would require. I need to rewrite these sections to separate the intelligent key issue from the natural key issue (the former being a subset of the latter).*

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

Primary Keys and Business Keys.

So now we have tables with both a *primary key* and a *business key*. The business key is how the *user* identifies a customer. The primary key is how the *DBMS* identifies the customer, and also how it implements joins to customer-related tables. The business key is not used as a foreign key. But it can continue to be used as what the business recognizes as a unique identifier.



A *business key* is a unique identifier which is made up of business data only, and no part of which appears in the primary key.

A *surrogate key* is a unique identifier functioning as a primary key, and no part of which is business data.

A business key is a *semantic key*.

A surrogate key is a *syntactic key*.

Figure 32. Definitions and Terminology Pairs: "Business Key" and "Surrogate Key".

The business key is the unique identifier which is made up of business data only. It is thus a unique identifier which describes. That's one thing which makes it so attractive to the business users. Experienced users, which know their data well, can often remember a large number of specific business keys, because the keys follow a clear pattern.

As I mentioned earlier, I am calling primary keys "syntactic keys". This is to distinguish them from keys composed entirely of business data, which I will call "semantic keys". Semantic keys contain descriptive information. Syntactic keys are meaningless, system-generated tags. Syntactic keys, then, are what we have called "surrogate keys".

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

With syntactic and semantic keys distinguished – all in the interests of not expressing semantics in a syntax which makes semantic changes costly to realize – there are up to four types of columns or groups of columns that may appear in a table. This is shown in Figure 33.

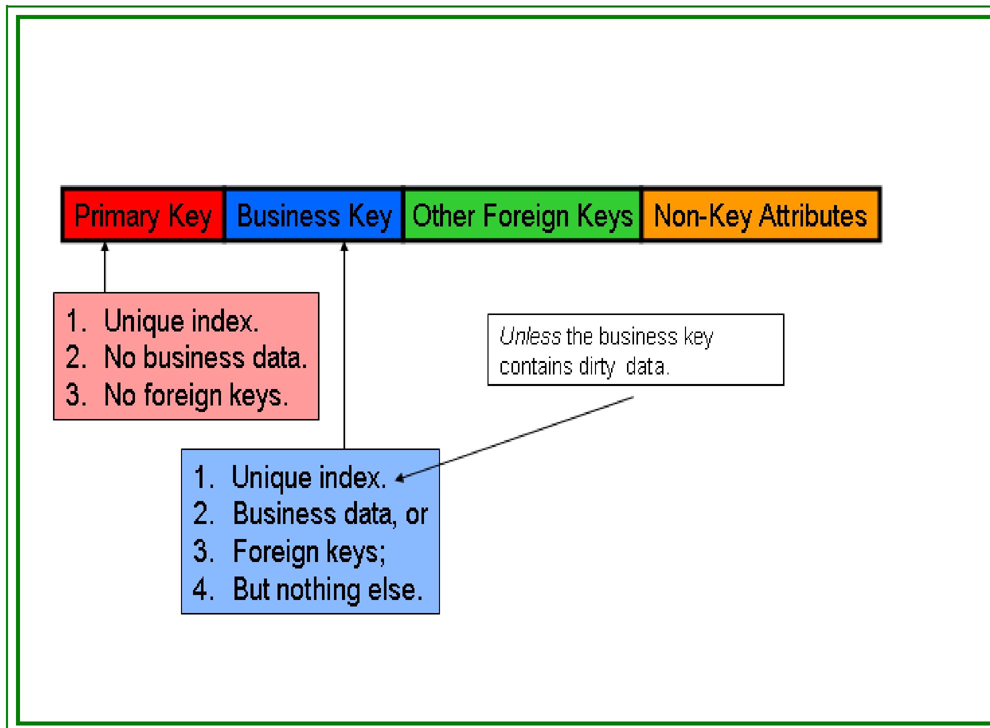


Figure 33. Types of Columns in a Table.

We will examine these types of columns in greater detail, in later essays. To pique your interest in that discussion, however, briefly consider the following aside, and the following definition of a term used in that aside:

Dirty data is data which does not conform to the constraints which define its semantics.

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

A non-key attribute which contains a value not in the domain defined for it, or which is a required attribute but is null, is dirty data. This is data which violates Codd's *domain integrity* constraint.

Any foreign key which refers to a non-existent row, or a required foreign key which is null, is dirty data. This violates Codd's *referential integrity* constraint.

Any column or group of columns which should be unique across all rows of a table, but which is not, is dirty data. In particular, non-unique business keys are dirty data. This violates Codd's *entity integrity* constraint.

Figure 34. Definition and Corollaries: "Dirty Data".

One of the things which distinguishes real world data modeling from textbook data modeling is the need to manage *dirty data*. Sometimes, the business key itself contains dirty data. When this is the case, we may have multiple rows in our table with the same business key. In the example we are considering, we may have multiple rows with the same cust-nbr in our source data, *and know that in some cases they represent different customers, while in other cases they don't.* If we knew which was which, we could assign different cust-nbrs in the former cases, and merge the rows in the latter cases. But the point is that we don't know.

In situations like this, we usually say that the data is not "de-dupped" (i.e. de-duplicated).

Some relational purists would point out that tables which are not de-dupped are not relational tables. They would encourage us to explain to our business users how serious a mistake it is to violate the requirement that every table in a relational database

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

should be a set, which entails that the table should *not* be allowed to contain duplicates.

But if loading this data is a business requirement, the purists are just whistling in the wind.

There is nothing wrong with loading such a Customer table, no matter how many duplicate cust-nbr rows there are. Indeed, if that's what the business wants, that's what we must do. We simply have to insure that when we do it, the system-generated primary key is unique, *and* that the uniqueness requirement is removed from the business key. Eventually, we hope, when we solve the data quality problems with cust-nbr, we can clean up the data and *then* apply the uniqueness constraint to the business key.

In the meantime, however, have we violated relational theory? Have we created, instead of a set, a multi-set, sometimes called a "bag"?

No, we have not. All rows in the table we have described are unique because each has a unique system-generated primary key. So *as a mathematical construct*, the table is a set, not a multi-set. Its unique primary key makes it so.

If the business key is not unique, this simply means that the semantics of this table is that each of its rows stands for a *putative* customer, a customer *as* given a unique identifier by some authorized customer-recognizing agent of the business.

An example of such an agent might be a claims-processing system. Based on dirty data, that system might create two *putative* customer records for the same customer. So as a semantic construct too, the table is a set, not a multi-set. More precisely, it is a multi-set not of customers, but rather of putative customers.

The business key of this table is not unique. It is the business key for customers, and this is not a table of customers. We

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

could create a unique business key for putative customers, if the business community thought it worthwhile to do so. For example, we could combine, with each cust-nbr, the name of the system that created it and a timestamp. This would be a valid business key, but a business key for a putative customer, not for a customer. But in general, business users are not interested in putative anythings. So instead of creating unique putative customer business keys, we simply remove the uniqueness constraint on the customer business key, i.e. we do not define a unique index over it.

In accordance with the principle that the name of a table should accurately reflect the meaning of the table, I recommend that all tables which may contain duplicates of X (e.g. of customers) be named “Putative X”. Encountering a table named “Putative Customer”, any user would be immediately aware that there may be duplicate customer rows in that table.

Nor should any reader think that “putative” is just a verbal trick. On the contrary, it is a point of verbal precision. Our company may *wish* it had a Customer table. But until it can correctly identify customers, and correctly assign one and only one cust-nbr to each one, it does *not* have a table of Customers. Call the table which it does have (and which it probably does call the “Customer table”) what you like. I call it a “Putative Customer table”.

The semantic gap between customers and putative customers is dealt with in a myriad of ways, most of them messy. We will consider this specific dirty data problem later on.

Figure 35. Aside: Dirty Data, Multi-Sets and Putative Customers.

Returning to the topic of distinguishing business keys from primary keys, our Customer Table now looks like this:

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

Customer Table

cust-nbr	**cust-nbr-bk**	other attributes of customer
200345434	ALB-000388	. . . .
974233818	ATL-001832	. . . .
010228433	NYC-002453	. . . .
464592317	ATL-009668	. . . .

Figure 36. Sample Data. Primary Keys and Business Keys.[4]

Cust-nbr is a system-generated surrogate key. Because it does not describe, it has no business meaning. Because the only role it plays is that of identification, it is no longer a homonym.

We still must change all the original customer numbers, however. Originally, cust-nbr was a Char(8) column, three alpha characters taken from a controlled list (a domain), a dash, and four numeric characters. That column is now defined as a Char(10), and the numeric part of the key is padded with a pair of zeroes on the left. Our 10,000st Atlanta customer originally had cust-nbr ATL-9999, and now has cust-nbr ATL-009999. The next customer would receive business key "ATL-010000".

This solves our problems with the intelligent customer number key. Business users get to keep their intelligent customer number, and use it to find rows in the Customer table. It is the column now named cust-nbr-bk, and it appears in one place only – on the Customer table. Business users get the sequence number part of that key enlarged, just as they requested, so the key can continue to be free of semantic anomalies. But since there are no foreign keys involved, the Customer table is the only table that must be altered, and there is only one column on that table to alter. *As experienced IT data management personnel can confirm from their own experience, the difference between altering one column on one table, and implementing the changes described in Figures 25 – 27, can easily be a couple of orders of*

[4] With the introduction of business keys to our tables, primary keys will still be the left-most columns, and will be shown underlined. Foreign keys will still be shown in italics. Business keys will be shown left-most following primary keys. They will be in bold-face, and will contain the suffix "bk".

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

magnitude in cost and elapsed time. And we achieved these dramatic reductions in cost to change by separating syntax from semantics, and we did that by removing the descriptive function from the cust-nbr primary key.

The database, and the SQL statements which access this table now have an identification mechanism (primary keys) and also a relationship mechanism (foreign keys) which are *impervious to change*. Because business users use the system-generated surrogate primary and foreign keys only for joining rows (and not for selecting rows), they have no interest in changing those keys.

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