Surrogate Keys

By Ralph
Kimball

According to the Webster's Unabridged Dictionary, a surrogate is an "artificial or synthetic product that is used as a substitute for a natural product." Thatýs a great definition for the surrogate keys we use in data warehouses. A surrogate key is an artificial or synthetic key that is used as a substitute for a natural key.

Actually, a surrogate key in a data warehouse is more than just a substitute for a natural key. In a data warehouse, a surrogate key is a necessary generalization of the natural production key and is one of the basic elements of data warehouse design. Let's be very clear: Every join between dimension tables and fact tables in a data warehouse environment should be based on surrogate keys, not natural keys. It is up to the data extract logic to systematically look up and replace every incoming natural key with a data warehouse surrogate key each time either a dimension record or a fact record is brought into the data warehouse environment.

In other words, when we have a product dimension joined to a fact table, or a customer dimension joined to a fact table, or even a time dimension joined to a fact table, the actual physical keys on either end of the joins are not natura keys directly derived from the incoming data. Rather, the keys are surrogate keys that are just anonymous integers. Each one of these keys should be a simple integer, starting with one and going up to the highest number that is needed. The product key should be a simple integer, the customer key should be a simple integer, and even the time key should be a simple integer. None of the keys should be:

- Smart, where you can tell something about the record just by looking at the key
- Composed of natural keys glued together
- Implemented as multiple parallel joins between the dimension table and the fact table; so-called double or triple barreled joins.

If you are a professional DBA, I probably have your attention. If you are new to data warehousing, you are probably horrified. Perhaps you are saying, "But if I know what my underlying key is, all my training suggests that I make my key out of the data I am given." Yes, in the production transaction processing environment, the meaning of a product key or a customer key is directly related to the record's content. In the data warehouse environment, however, a dimension key must be a generalization of what is found in the record.

As the data warehouse manager, you need to keep your keys independent from the production keys. Production has different priorities from you. Production keys such as product keys or customer keys are generated, formatted, updated, deleted, recycled, and reused according to the dictates of production. If you use production keys as your keys, you will be jerked around by changes that can be, at the very least, annoying, and at the worst, disastrous. Suppose that you need to keep a three-year history of product sales in your large sales fact table, but production decides to purge their product file every 18 months. What do you do then? Let's list some of the ways that production may step on your toes:

- Production may reuse keys that it has purged but that you are still maintaining, as I described.
- Production may make a mistake and reuse a key even when it isn't supposed to. This happens frequently in the world of UPCs in the retail world, despite everyone's best intentions.

- Production may re-compact its key space because it has a need to garbage-collect the production system. One of my customers was recently handed a data warehouse load tape with all the production customer keys reassigned!
- Production may legitimately overwrite some part of a product description or a customer description with new values but not change the product key or the customer key to a new value. You are left holding the bag and wondering what to do about the revised attribute values. This is the Slowly Changing Dimension crisis, which I will explain in a moment.
- Production may generalize its key format to handle some new situation in the transaction system. Now the production keys that used to be integers become alphanumeric. Or perhaps the 12-byte keys you are used to have become 20-byte keys.
- Your company has just made an acquisition, and you need to merge more than a million new customers into the master customer list. You will now need to extract from two production systems, but the newly acquired production system has nasty customer keys that don't look remotely like the others.

The Slowly Changing Dimension crisis I mentioned earlier is a well-known situation in data warehousing. Rather than blaming production for not handling its keys better, it is more constructive to recognize that this is an area where the interests of production and the interests of the data warehouse legitimately diverge. Usually, when the data warehouse administrator encounters a changed description in a dimension record such as product or customer, the correct response is to issue a new dimension record. But to do this, the data warehouse must have a more general key structure. Hence the need for a surrogate key.

There are still more reasons to use surrogate keys. One of the most important is the need to encode uncertain knowledge. You may need to supply a customer key to represent a transaction, but perhaps you donýt know for certain who the customer is. This would be a common occurrence in a retail situation where cash transactions are anonymous, like most grocery stores. What is the customer key for the anonymous customer? Perhaps you have introduced a special key that stands for this anonymous customer. This is politely referred to as a "hack."

If you think carefully about the "I don't know" situation, you may want more than just this one special key for the anonymous customer. You may also want to describe the situation where "the customer identification has not taken place yet." Or maybe, "there was a customer, but the data processing system failed to report it correctly." And also, "no customer is possible in this situation." All of these situations call for a data warehouse customer key that cannot be composed from the transaction production customer keys. Don't forget that in the data warehouse you must provide a customer key for every fact record in the schema shown in Figure 1. A null key automatically turns on the referential integrity alarm in your data warehouse because a foreign key (as in the fact table) can never be null.

The "I don't know" situation occurs quite frequently for dates. You are probably using date-valued keys for your joins between your fact tables and your dimension tables. Once again, if you have done this you are forced to use some kind of real date to represent the special situations where a date value is not possible. I hope you have not been using January 1, 2000 to stand for "I don't know." If you have done this, you have managed to combine the production key crisis with the Year 2000 crisis.

Maybe one of the reasons you are holding on to your smart keys built up out of real data is that you think you want to navigate the keys directly with an application, avoiding the join to the dimension table. It is time to forget this strategy. If the fifth through ninth alpha characters in the join key can be interpreted as a manufacturer's ID, then copy these characters and make them a normal field in the dimension table. Better yet, add the manufacturer's name in plain text as a field. As the final step, consider throwing away the alphanumeric manufacturer ID. The only reason the marketing end users know these IDs is that they have been forced to use them for computer requests.

Holding onto real date values as keys is also a strategic blunder. Yes, you can navigate date keys with straight SQL, thereby avoiding the join, but you have left all your special calendar information marooned in the date dimension table. If you navigate naked date keys with an application, you will inevitably begin embedding calendar logic in your

application. Calendar logic belongs in a dimension table, not in your application code.

You may be able to save substantial storage space with integer-valued surrogate keys. Suppose you have a big fact table with a billion rows of data. In such a table, every byte wasted in each row is a gigabyte of total storage. The beauty of a four-byte integer key is that it can represent more than 2 billion different values. That is enough for any dimension, even the so-called monster dimensions that represent individual human beings. So we compress all our long customer IDs and all our long product stock keeping units and all our date stamps down to four-byte keys. This saves many gigabytes of total storage.

The final reason I can think of for surrogate keys is one that I strongly suspect but have never proven. Replacing big, ugly natural keys and composite keys with beautiful, tight integer surrogate keys is bound to improve join performance. The storage requirements are reduced, and the index lookups would seem to be simpler. I would be interested in hearing from anyone who has harvested a performance boost by replacing big ugly fat keys with anonymous integer keys.

Having made the case for surrogate keys, we now are faced with creating them. Fundamentally, every time we see a natural key in the incoming data stream, we must look up the correct value of the surrogate key and replace the natural key with the surrogate key. Because this is a significant step in the daily extract and transform process within the data staging area, we need to tighten down our techniques to make this lookup simple and fast.

© Kimball Group. All rights reserved.