

What Makes Real-World Data Modeling Tough?

Dr. Tom Johnston
MindfulData.com

There is one thing that distinguishes real-world data modeling from what we see in textbook presentations of normalization, or in academic articles in such publications as those from the IEEE or the ACM. Real-world data is dirty!

When practitioners talk about dirty data, they aren't calling the data "dirty" because it isn't fully normalized. They are talking about data issues like these ones:

1. "SSN comes in as a text field. We filtered out all zeroes and all nines, but who could have imagined that we'd get thousands with a leading zero followed by spaces? So until we get it fixed, just filter out the leading zero plus spaces SSNs in your SQL."
2. "Invoice number is unique in the OLTP system, but not for data warehousing purposes. That's because the sequence number used for invoices recycles every couple of years. The OLTP system only keeps twelve months of invoices in its database, but the data warehouse needs to keep five years plus a month. That's why the data warehouse uses a surrogate key for invoices, even though it keeps the original invoice number. And that's why you're finding 'duplicate' invoices in the warehouse. Just keep that in mind as you write your queries."
3. "Jones and MacDonald are our only two salespersons who work on more than one sales team. Jones actually works on three teams, and MacDonald works on two. Our database only allows a salesperson to work on one team, however, and it's too expensive to change, especially for a couple of rare cases that may go away next year anyway. So we've put Jones into the Salesperson table three times, with salesperson-id Jones-1, Jones-2 and Jones-3. MacDonald is in there as MacDonald-1 and MacDonald-2. So if you don't want counts, averages and other statistical functions against the salesperson data to return faulty results, keep this in mind."

4. “Several years ago, the post office split out zipcode 20912 from zipcode 20012. The reason was to separate the Maryland portion of 20012 from the Washington DC portion, which retained the 20012 designation. That’s why the Sales by Zipcode data mart shows a drastic drop in total sales in 20012 starting in the year that change took place. Just keep that in mind when you’re doing your geographic sales analyses.”

The admonition at the end of each of these statements is there to make an important point. With dirty data like this, the burden of compensating for it lies with the end user. And this is the last, worst, and all too often most common, point at which problems like these are resolved!

Data purists might say that the way to handle dirty data is obvious: keep it out, and don’t allow it in until it’s cleaned up. Well, how would we apply this advice to our four examples above?

The first example seems pretty easy. The solution is to put a better filter on SSN as the transactions are loaded into their target databases. But let’s look at the situation a little further. Maybe there is some information value in the bad SSNs because of which we should let them in.

For example, if we let bad SSNs into our data warehouse, we may accumulate enough information over time to rank our external data sources by how good the SSNs are that they send us. On the other hand, a nine-for-nine SSN match has always counted as a pretty reliable indicator of related records; but it certainly is *not* a reliable indicator when the matching numbers are numbers like “9 “. So perhaps the best solution is to put *two* SSNs in the data warehouse, a valid-SSN column which is blank if and only if the source SSN is null or invalid, and a source-SSN column which contains whatever was in the source SSN when it arrived at our front door.

So let’s suppose we did that and, after loading several years’ worth of transactions, analyzed SSN data quality. This resulted in a letter being sent to the Acme Corporation, chastising them for sending us transactions in which, over the past five years, 60% of the SSNs were bad. Acme protested,

and said that they send a high percentage of transactions without any SSN, but they don't send bad SSNs!

So we went back and looked at the report that gave us the information, and indeed it showed Acme with 60% in the "Bad SSN" column. The IT department was asked to look at the SQL behind that report. They did, and said that it was fine; there was no mistake. However, after a couple of meetings between IT and the VP of vendor relations, it was discovered that the programmer who wrote the query thought that a blank in the valid-SSN column indicated a bad source SSN. Of course, as we just said, what it really indicated was *either* a bad source SSN *or* a null source SSN. To put the problem in the language of concepts, it is that the concept "bad or null SSN" may be a fine concept for some purposes, but for the purposes of analyzing SSN data quality, it is not. It is a "smushing together" of two concepts, "bad SSN" and "null SSN", and the concept we needed was the former one only. Needless to say, relations with the Acme Corporation have not improved because of this incident.

This is the simplest of the four examples listed above, and it has turned out to not be so simple after all. I encourage all readers, and not just the skeptical ones, to look at the other three examples, and decide what data model changes they would make to improve things.

Semantics.

In all of these examples, we find a single root cause of dirty data. That root cause is misunderstood or altered semantics. Either: when the data was modeled, and the code written, the meaning of the data was not properly or completely understood by the modelers and developers. Or: after the system was deployed, the meaning of the data changed, and corresponding changes to the system and its database were not made. To illustrate, with the remaining three examples:

- In the second example, the meaning of invoice number was assumed to be "a unique identifier of an invoice". The real meaning was "an identifier of an invoice that is unique until the sequence number recycles". The real meaning was good enough for the OLTP system. It wasn't good enough for the data warehouse.

- In the third example, statistical functions assumed that the meaning of a salesperson-id was “the unique representation of a salesperson”. The real meaning was “the representation of a salesperson, usually unique, but no guarantees”.
- In the last example, geographical sales analyses routines assumed that the meaning of a zipcode was “a specific geographical area”. But the Post Office uses zipcode as a tool to deliver mail efficiently, and will both add and change zipcodes in order to improve mail delivery. The meaning of “a specific geographical area” has been applied by companies doing geographical and/or demographic analyses, not by the Post Office. If you redefine concepts which someone else can change, you are doing it at your own risk.

What makes real-world data so much more difficult to deal with than one would suspect from textbook and academic article examples is that real-world data is dirty. Its semantics are obscure or otherwise compromised, and the clean-up is difficult because we can't just ignore dirty data or refuse to allow it into our databases.

Normalization doesn't address the dirty data issue. Rather, it *assumes* that the semantics of the data being normalized is clear and unambiguous. New developments in data engineering also fail to address the dirty data issue, and also assume that the semantics of the data being engineered is clear and unambiguous.

If theory – especially normalization and the rest of relational theory – is to truly be practical, we must learn how to apply it to real-world data, and not just admonish practitioners to understand theory better and apply theory more thoroughly. We must learn how to apply theory to dirty data, data which cannot be swept under the rug or even kept out of the house entirely, and data which is often very resistant to the topical or systemic application of “theory”.