How Perverted is Your Data?

By David A. Ruble, Olympic Consulting Group

How perverted is your data? As salacious as this question sounds, most major IT systems of any appreciable size or age contain some level of perverted data.

per·vert·ed [pər vúrtəd] (adj)[1]

1. deviating from what is proper: deviating greatly from what is accepted as right, normal, or proper
2. (Um … Let's skip this one!)
3. distorted: misinterpreted or distorted

What is perverted data?

Perverted data is not to be confused with dirty data. Dirty data is generally accepted to be data in a system that is inaccurate, incomplete or erroneous. Perverted data is none of these things. It may be accurate, complete and even error-free. However, it is represented in such a deviant manner that it creates havoc when you try to integrate or replace the systems in which it resides. Data that's perverted is typically undocumented and cryptic. In its worst form, it is stored in fields that contain more than one single fact.

As we look at some of the more spectacular examples of perverted data that I've encountered in my career we'll endeavor to understand how the data came to such a sorry state. Most are the result of a system's design degrading over time as it is continually patched and extended beyond its original scope. In other cases, the downright naughty nature of the data can be traced to technical constraints that were foisted upon the system's designers at the time it was created. (Names have been changed to protect the guilty.)

Next I'll explain the tremendous cost and impact of perverted data to an organization and examine some of the strategies for dealing with it.

[1] Bing Dictionary

Examples of Perverted Data

Overloading: A Code for All Reasons

Much of what I call perverted data is referred to in polite company as “overloaded” data. Overloading occurs when more than one fact is packed into a single field. Mostly, this is the natural result of ever-changing business requirements that out-pace the rate of enhancements to the business’ computer systems. A need arises for the system to do something new; yet the budget or the time does not exist to upgrade the system properly, so clever users and intrepid IT staff figure out a way to rig the system to do what is necessary by overloading the contents of existing fields to accommodate new facts.

Computer systems are said to be inherently entropic, which is a fancy way of saying they tend toward an increasing state of disorder over time. Entropy is a measure of the disorder in a system containing energy or information and is the founding concept of the 2nd law of thermodynamics.[2] The less ordered a system is, the greater is its entropy.

As a system is enhanced and data fields are repurposed or overloaded to meet short-term business objectives, the meaning of the data becomes more and more inscrutable through simple inspection. After looking at a system that had degenerated into a state of severe entropy over three decades, my colleague Meilir Page-Jones once observed: “This system is like a fractal. At every level of magnification it's complete crap.”

One of the worst cases of data perversion due to overloading that I’ve encountered involved a simple mechanism for grouping customer accounts together. The original request by the business was to group customers that shared the same contractual pricing so they could be treated as a whole, rather than individually.

No sooner had the compliant IT folks added a field to the customer file for this grouping code, the business dreamt up all sorts of other reasons why they might benefit from grouping customers together. By the time we were called on to the scene, use of this innocuous field had spiraled out of control and the concept of “pricing group” had long since lost all meaning. The field itself was impossible to define. Considerable forensic analysis uncovered no less than thirty different uses of the grouping field such as identifying legal ownership of like customers, grouping customers for the purposes of awarding sales commissions and identifying respondents to direct marketing promotions.

Perhaps the most perverse use of the customer grouping code came to pass when the firm created a new service, but found the IT systems lacking in their ability to properly identify the service, so the customer grouping code was coopted to stand in for a product identifier.

[2] In communications theory, entropy is a measure of the information that is lost in the transmission of signals; messages tend to become distorted or degraded because of random errors called noise. In computer systems, entropy can be observed by the manner in which the meaning of stored data becomes increasingly unintelligible by simple inspection.

You may be nodding your head thinking, "I've seen something like that." If so, you'll appreciate that if you tried either to replace this system with something sensible or to integrate it with another system, you'd soon get stuck – because there's no systematic or automated way to untangle such wreckage.

In the absence of any documentation, you still have a couple of assets to fall back on. The first is human memory. You may be lucky enough to find some wizened old timers who either directly witnessed the permutations of abuse, or who later learned about the perversities as tribal folklore. If so, then interviewing the user community and IT community is your best option to unraveling the meaning of the data. Unfortunately, the second option involves inspecting the source code. Nothing short of finding every place the offending field is referenced and evaluating what is actually done with it, will reveal the full scope of how it is used. In many cases, it is a safe bet that you'll find copious hard-coded logic that says, "If group-code = 'XYZ', then do ABC."

Reason Codes beyond Reason

Another common horror in systems is numeric status, reason and disposition codes. You've probably seen these and have tried to align them with codes from other systems in a data warehouse at some point in your career.

With modern relational database and relatively cheap storage, it continues to vex me as to why developers seem to delight in reducing simple status values to numeric codes. How many times have we seen "active status" fields whose values were "00" and "01" instead of simply "inactive" and "active?" Or yes/no fields with values of "01" and "00" respectively – as if the designer had left room for 98 shades of "Maybe."

Reason and explanation codes are great examples of data that has lost its original meaning. Codes 01 through 05 may have made perfect sense, but 07, 08 and 09 were wedged into the database for an entirely different purpose than explaining a transactions' "reason" – and nobody remembers what 06 was used for.

The scourge of cryptic numeric codes is bad enough, but data gets really perverted when the same code *value* is assigned different meanings. I saw this first hand recently in a system that tracked users' online behavior. The action codes for user events such as click-through, starting or pausing a video were well known. Things got out of hand with the sudden introduction of new touch screen device types that came with a plethora of gestures that had no prior representation in the system.

For reasons that were never adequately explained to me, the upstream systems were not modified to overtly differentiate these new events, but the downstream data warehouse now had to interpret an action code of "3" that meant one thing when the subject device was a garden-variety PC, and meant something completely different when the device was a touch-screen mobile device.

I am confident that the developers of this solution knew full well that they were trading expediency and time-to-market for additional complexity, and accepted begrudgingly that it was a worthy trade off. Unfortunately, single decisions such as this one occur perhaps hundreds of times over the course of

many years, and the cumulative effect is that the enterprises' systems appear to have been designed by people who had taken leave of their senses.

Perversion by Design – When systems are born that way

So far we've looked at examples where fine upstanding systems have become perverted over time. Now let's look at the case of systemic perversion – where the system is simply born that way.

Our firm was once called into do an audit of the engineering soundness a software vendor's packaged system. At issue was the database design. Our client had purchased the package without actually being privy to the structure of the underlying data. This is not an unusual situation. Most package vendors jealously guard their actual database design as highly-valuable intellectual property. One hopes that the reason is that the level of abstraction and generalization is so incredibly elegant, that the vendor has truly alighted upon a schema so flexible that they can accommodate virtually any new customer's proclivities without twisting their package in knots.

On the flip side, the worry is that the vendor's underlying data structures are such a mess, a mere glimpse at them would send potential customers fleeing from the building.

Vendors of packaged software are highly practiced at telling customers that theirs is a closed system and one shouldn't care what goes on under the covers, as long as the system produces the result the customer desires. To a point this is true. My counter-argument to this is that no system is an island unto itself in a modern enterprise, so if they insist on keeping their internal structures a secret, they better produce one heck of a detailed and elegant application interface specification so we can comprehend the structure, relationships and meaning of the data that feeds into or extrudes out of the package.

Returning to the story of our audit, our client had found a way to peep into the package's actual database and what they saw utterly horrified them. In concept, it looked something like this:

Column A	Column B
Dave123MainStreet	[code snippet]
Rat Poison	[code snippet]
Bin42OutofStockBackOrder	[code snippet]

At first glance, one will notice that the contents of Column A seemingly have nothing in common with one another from row to row. Additionally, Column A appears to sometimes contain concatenations of multiple data elements crammed into one column.

The big tip off as to what was going on here can be found in Column B, which contained little snippets of code. The source code in Column B was actually the parsing routine that told you how to read and interpret the contents of Column A. What was most egregious was that this "design" was replicated all over the very large system.

Our investigation into this seemingly bizarre system resulted in a road trip to meet with the vendor and the core design team that was responsible for it. I admit, I was expecting to stumble across a lost tribe of developers who resided on their own technological Galapagos Island and had missed several decades of software engineering evolution, data modeling and best practices.

Instead, much to my surprise and delight, I found one of the most capable software engineering groups I've ever encountered. The package's designers admitted, rather sheepishly, that the first version of the system was designed using a flat file database, but was coded in a language that was only capable of accessing 11 files concurrently. So they were horribly constrained by this limitation and had to find a way to cram more and more information into the few files they had available.

Ultimately, they became a victim of their own popularity. As the installed base of customers who purchased the system grew, so did the number of enhancements, and they lacked the capital necessary to re-engineer their database into something modern and sensible.

The moral of this story is that once you take the first step down The Road to Data Hell, you may never be able to retrace your path. You may be drawn ever further into the realm of evil data design and its resulting perversions. Even packaged software is not immune. It can be just as perverted as custom-built systems - and sometimes even more so, depending on its age and the number of modifications made to accommodate disparate customer requests.

Why perverted data is a big problem

Integration of heterogeneous computer systems costs companies millions of dollars every year and accounts for a significant chunk of any shop's total IT budget. Whether the diverse landscape of systems we face came about through mergers and acquisitions or is the cumulative result of natural growth and development is irrelevant. This is the world we live in as data professionals, where more and more of our time is spent in knitting together what already exists than in designing new transaction processing systems from scratch.

The most common approach to solving the problem of multiple, overlapping and divergent representations of the same data is to try to rationalize the data in the back-end business intelligence systems. This strategy says: "We'll clean up the mess downstream."

The upstream approach attempts to drive the various systems to adhere to a single, enterprise canonical model, which is expensive and time-consuming, but ultimately can make great headway in draining the upstream swamp.

Whether your data integration strategy involves (a) wrapping existing systems with a common set of translators, redesigning and refactoring legacy systems or (b) simply piecing it all together in data marts and data warehouses, the fact is that integration efforts come to a grinding halt when perverted data is discovered.

Perverted data, as I have defined it here, defies comprehension through simple inspection. It may be overloaded, concatenated and /or ambiguous in isolation. (That is, it uses the same values to convey completely different meanings, which can be resolved only in context with other differencing data.) It is no mere mapping exercise to make sense of it, and it often derails entire integration efforts.

So what can be done about it?

As much as we hear about automated data integration tools, PEOPLE are always your best first line of defense.

Let's take the example of runaway status, reason and disposition codes. You can use any number of tools and queries to trawl through the ocean of source code and snag each and every instance where the code is referenced – looking especially for cases where the logic refers to specific values upon which the automated systems' behavior hinges.

From this sort of bottom-up forensic analysis, you can eventually piece together a complete picture of what the data actually means, based on how it is used in the system. What you do next is critical. You MUST write a clear and cogent definition of the field at the detailed level in order to contain the madness that these fields wreak on your organization.

I can't stress enough the importance of solid data definitions. Do not leave this task to amateurs or those who view the task as some sort of punishment or violation of the Geneva Convention. How many times have we been told a system already has a data dictionary only to find definitions that look like this?

Field	Definition
Address Line 1	The first line of the address
Order Status	The status of the order

Definitions such as these are a complete waste of paper, ink, pixels and quite frankly, oxygen. The Address definition is one of particular interest to me, because I encountered just such a definition once, which omitted the fact that a major constraint in this business was that they would not deliver to a PO Box if the associated Address Type value was "Ship To."

Everybody who hung around the old mainframe knew this basic business rule by heart, but they had never bothered to write it down. So when they outsourced the development of their web-based portal, this crucial piece of business intelligence was not revealed to the vendor until the product failed basic acceptance testing. Eventually, their new system correctly fashioned an "is a PO Box" flag that allowed them to test for the rule once and save the result. This essentially untwisted the previous design that obfuscated the real policy associated with the address line.

At least in the case of the PO Box rule, one could eventually find a snippet of code in the mainframe that revealed the policy's enforcement. Some rules don't exist in any system at all – leaving their enforcement entirely in the realm of human endeavor.

I discovered such an example recently when ordering custom window blinds from a retail store that had them fashioned by a third party supplier. I told the clerk I would pick the blinds up from the store when they were ready to avoid the cost of having them shipped a few blocks to my house. He reacted by selecting "Truck Delivery" from the list of options, which inserted a $10 shipping and handling fee which he then zeroed out with the space bar.

Feeling a bit uneasy, I asked whether he understood my request. "Our system doesn't have the option of entering our store as the Ship-To instead of the customer," he explained. "So we pick 'Truck Delivery' and zero out the charges and that's what tells our supplier to send the product back here to us." I felt so relieved.

This is a perfect example of highly perverted data. If one were to simply look at the database, you would see a bunch of truck deliveries, seemingly to customers' addresses, with no shipping and handling charges. Nowhere in any line of code or layer of the system would you find the rule that states this unique combination of data really means "ship the product to the order's originating retail store for customer pick-up."

This brings me back to my earlier point that people are your greatest assets when attacking the problem of data perversion. Automated tools and source code inspections will only go so far when your data looks like the aftermath of a wild Twister party. Nothing short of face-to-face interviews between a skilled analyst and members of the business and IT staff will reveal the true meaning of what really lurks in your systems' databases.

Conclusion

Just because your data is perverted doesn't mean it's dirty. The data may well be accurate and complete. It is, however, represented in such an unorthodox way as to be unintelligible without special knowledge. In most cases, nobody set out with nefarious intent to make it this way. There were no disreputable developers sitting on a park bench, eyeing flat files with bad intent. Rather, data perversion in systems is usually the result of many short-term design decisions adding up to a twisted critical mass over time.

The irony is that even though the data may be perverted, the business may be running along just fine. In fact, in an entirely closed ecosystem, nobody might care. It is typically when we try to integrate or replace such systems that full cost of the data's anomalous proclivities hits us squarely in the pocketbook.

Vendors who claim that what's going on under the covers of their package is nobody's business need to be challenged with how their system will interact with others around it. If you are not allowed to peer inside the package, it better come with a heck of a solid application interface specification.

When it comes to integrating and making sense of your deviant data, forensic analysis of your systems' source code will only go so far. Even a massive reverse-engineering effort to discover how data is used will ultimately hinge on human interpretation of the code, which may raise as many questions as it answers.

This brings me to the final and most important point of this paper, and that is that people are your biggest assets for untangling wayward data. It requires skilled analysts meeting face-to-face with bona fide users and those who possess your company's institutional memory.

The final result must be well-written, cogent and complete definitions that clearly explain how the data is represented, what it means and how it is consumed. Only that level of detail will provide the clarity required to unravel the mess and afford people the understanding to design solutions that better adhere to generally-accepted principles.

About the author

David Ruble, a principal senior partner at Olympic Consulting Group (www.ocgworld.com), is a senior analyst, designer, author and educator. He is widely regarded as an expert in the field of business analysis, information modeling, GUI design and functional specification. He has been a principal analyst and designer of many mission-critical global corporate information systems – linking suppliers and customers worldwide. David also has significant experience designing applications in the transportation, health care and public safety sectors. His background in business, technology and art create a unique skill set that allows David to communicate with ease among business people, technologists and graphic designers.

As an educator, he has taught software engineering techniques to hundreds of students throughout the United States. He is the author of Practical Analysis & Design for Client/Server & GUI Systems, published by Prentice-Hall, 1997. His book has been used widely in colleges and universities throughout the United States and Thailand, Mexico and Argentina and is considered a timeless classic in the field of business analysis.

About Olympic Consulting Group

Olympic Consulting Group
Software Architecture and Development

Olympic Consulting Group (OCG) is a full-service system architecture and development firm serving the Puget Sound region since 1997. The firm specializes in delivering high-performance consulting in the analysis, design, development and project management for complex business systems and government agencies. www.ocgworld.com