What I’m Telling Business People About Why Relational Databases Are So Bad

The following is what I have written in my book “Avoiding IT Disasters” to explain to business people why relational databases are the cause of so many problems in enterprise systems. This is a non-technical explanation that is meant to be accessible for a general audience, but I thought it might be interesting for a technical audience to read what I feel non-specialists should be told.

In the 1970s a new type of database was proposed by Dr. Codd and Dr. Date of IBM. It was called a “relational” database. Now, I have never seen any indication that either Dr. Codd or Dr. Date ever built a real-life enterprise system. If they had, I think they would have realized that relational calculus was just not a good fit for enterprise systems. Not having to actually make a system work, all they had to do was give highly-contrived academic examples. Anyone who had worked in the trenches of enterprise systems knew that real life was far more complicated. It’s just tough to write enterprise software. It is an intersection of computer and human behavior and it is really difficult to get things that satisfy those sometimes conflicting needs.

I was working on software systems for a large union at the time, and I remember everyone was talking about relational databases. It wasn’t that any of them really knew what a relational database was, but they had heard the term and it sounded good. It was at a time when computer terms were starting to come into common business parlance. There were a lot of magazines like *Byte* and *PC Mag* starting to appear on newsstands, and some of the stories were being carried in the newspapers. The idea of a database was an appealing one; the idea of something that stored the data like a filing cabinet. It is such a physical world concept that it became

something that people who really didn't know much about it started talking about.

Then there was the name, "relational." There were stories about how it would allow you to "travel relationships" throughout your data. Get the name of the department of the project manager of the project, that kind of thing. But actually, the word relation in relational refers to the mathematical concept of a relationship, which is a collection of groups of data. If two things are in a group they are said to be related. In fact, the mathematical theory of relations eschews the concept of a connection between the related objects and defines relationship just as the set of the related things. Any such set is a relationship in the mathematical theory. This is completely removed from our business-centric concept of a relationship, where there has to be some kind of a connection. If you had a group of pairs of objects and someone told you they were related, you would look for some kind of connection. There are puzzles that are set up that way, and we have to hunt for the hidden factor. It's a very human instinct for pattern-seeking. But the theory of mathematical relationships doesn't care. They can be considered a relationship even if they are just paired by chance. When I was teaching mathematical logic at York University I taught a course in the mathematical theory of relations. Some students just couldn't get away from relationships as having some kind of rule. The abstract concept the theory was based on was something they just couldn't get. When I heard that IBM had come out with a proposal for a database based on relational calculus I was startled. Having by that time written lots of enterprise software, I couldn't see how trying to force things into such a formal and abstract structure was going to be helpful.

In my opinion it was one of the most significant contributors to complexity and system overruns. Let me go through this technology and show you some of its problems.

Have you ever seen an old movie, probably set in the 1920s or 1930s where someone is reading a telegram?

It always went something like:

HAVE ELOPED WITH DANNY STOP WILL HONEYMOON IN MADRID STOP GLORIOUSLY HAPPY STOP

They were always like that, with all those STOPs. What was that all about?

To understand, we have to go back to the Boer War, which was a nasty war the British fought at the turn of the 20th century in South Africa. As well as introducing the world to the concept of concentration camps and trench warfare, it was the first war to use wireless telegraphy, so that troops could get orders by telegraph right at the front. War fronts are dirty places, even more so at the turn of the twentieth century, with mud and horse manure being splattered all over. Worried that a mud splatter could be mistaken for a comma or a period that might change the intent of an order, the British War Department mandated that all punctuation should be spelled out, for example, COMMA. For the period they chose the short word STOP, named after “full stop” which is a more British name for a “period.”

It became a practice to write out the telegraph code for a period as the word STOP. And when people read them out loud they instinctively read out the word STOP rather than just treating it as a period at the end of the sentence.

“What has that got to do with relational databases?” you might ask. The commands to a relational database are given in actual readable text. This text is in a language called SQL (Structured Query Language). SQL uses punctuation to separate the commands, just as the telegram used the word STOP between sentences. In effect the database gets its commands in a stream of text, like a telegram, with STOP between each command[1].

Look at this example:

UPDATE CUSTOMER_TABLE SET NAME=“John Smith” WHERE CUSTOM_NO=2333 STOP UPDATE …

That is a command in SQL to update customer record 2333 setting the name to John Smith.

Now notice the text between the quotes i.e. “John Smith”. Where did that come from?

Well it was entered by someone in a browser filling in a form. The person filling in the form typed in “John Smith” and hit the submit button. The web site code put what was typed in between quotes into the SQL statement and sent it to the database for execution.

Now suppose the customer is nefarious and types in this for his name:

John” STOP DELETE CUSTOMER_TABLE STOP

If the web site just puts it into the SQL statement it ends up looking like:

UPDATE CUSTOMER_TABLE SET NAME=“John” STOP DELETE CUSTOMER_TABLE STOP” STOP UPDATE …

And perhaps you can see what it has been changed to. There is a command to set a name in the customer table to John, which will be done, but then it will do the next command inserted by the nefarious customer, which is an instruction to delete a whole table of data. Remember that this writing to the database is being done by a task that has to have sufficient rights to update the database.

That is what is called “SQL Injection.”

SQL Injection has been the single biggest technique for website hacking and company intrusions. Over 90% of all the major website penetrations were done via SQL injection. You just have to google it and see a flood of data breaches that total into hundreds of millions of credit cards being compromised, bank accounts drained, and personal information exposed.

Pay close attention to what is going on here. The database commands are in text, and the data entered from web forms by Internet users is merged into that text, which allows people filling in the form to try to fool the database into interpreting the command incorrectly.

If this seems to you like a stupid way of doing things you are exactly correct.

This is probably the dumbest technological decision ever to be used extensively by so many, and at so much cost.

This is the software equivalent of a nuclear power plant combining the control room with the visitor's gallery.

It makes no sense to have two things separated, one the commands and the other the data from the forms, then to mix them up, and then to fight a technical back-and-forth battle trying to not get fooled into thinking the data is actually part of the command.

Why mix them up in the first place?

This is just a terrible architecture and it is responsible for billions of dollars in damage to organizations around the world.

But the relational database story gets worse.

Let's say you have a timesheet that you have implemented in software. The timesheet has an employee number. Now you are displaying the time sheet and you want to get the name of the employee. That is on the Employee table, so you have to create an SQL statement like:

SELECT EMPLOYEE WHERE EMPLOYEE.NUMBER EQUALS TIMESHEET.EMPLOYEE_NUMBER

This takes the employee table and matches it up with the timesheet table, and then you pick. What you are really doing here is defining the relationship between the employee and the timesheet. You are saying that they are connected via the employee number that is on the time sheet.

Remember that the timesheet form has already been described to the software. You said the field on the timesheet was the employee number, so basically at this point that relationship was already known to the software. But now you have to go to all the trouble of constructing an SQL statement and sending it to the database for execution, and then getting back a set of table rows from which you pick out the information you need. This is all totally unnecessary! The information that the timesheet was related to an employee had already been communicated to the software. Using a

relational database made it necessary to ignore that information and redefine the relationship in a totally different language.

There is a principle in computer science called the DRY principle, which stands for Don't Repeat Yourself. Its primary thrust is to not repeat code for the same calculation. You should write the code only once and call upon it whenever the calculation is needed. However, it also extends to all kinds of ways of removing redundancy. This is a principle of reducing disorder/complexity. Software that uses the same code for the same calculation removes the possibility of those two separate implementations getting out of step.

Using SQL to express "relationships" between data that have already been expressed in a different form is a total violation of the DRY principle. Information is precious in software. When it has been captured it should be squeezed for every possible way it can be used. You should never have to re-enter information. You should never have to enter something that could have been derived from what you have previously entered. To do so is to create the possibility of the two versions of this information getting out of step.

Ever since relational databases were proposed, I have been puzzled as to why this seemingly bizarre architecture has been allowed to persist.

This is like having your filing department speaking a foreign tongue so that all instructions have to be written down in this language.

But it's worse. When you store that timesheet in a relational database you have to totally take it apart, with the header information in one table and all the detail lines that assign hours to projects as separate rows in another table. You have to take apart the form and construct the SQL that takes those bits and stores them. Oh yes, and make sure you put sequence numbers on all those detail lines in the timesheet if you want to get be able to get them back in the same order. When you want the form back, you have to write SQL instructions to join the tables together and then you have

to pick out all the timesheet information from the returned results and put it together as a form.

Some people have described this as like having to take your car apart each evening when you come home, hang up the parts on your garage wall, and in the morning reassemble your car before you drive to work.

All of this takes a lot of extra code to translate between the different worlds of the relational database and the object world of the software. Extra code means the probability of extra errors.

As if that wasn't bad enough, the data in a relational database is stored in ways more in keeping with a 1980s programming language than with a modern, object-oriented language. All of the data in today's modern, object-oriented languages have to be encoded into these primitive data types.

This is sometimes referred to as the "object-relational impedance mismatch." Seriously? An impedance mismatch between an amplifier and a speaker I can understand, as it refers to a real physical phenomenon. In this context it is just technobabble, which should be replaced with "consequences of a really stupid architecture."

If you want to know why enterprise systems fail so often this is not the whole cause but it is a major one. The necessity of having to duplicate all of this logic in different languages, and in really different ways of representing data, adds a massive amount of disorder/confusion to an ERP system.

Take an older code base with a lot of additions and changes by different people over the years, then try to customize it to a new situation, and throw in all this added complexity of a relational database, and you are putting the project seriously at risk.

That said, it is true that relational databases are ubiquitous. So much so that there are programmers who have never really seen any other kind of database and believe that all databases are relational.

Relational databases have been the worst technology to ever poison a field of endeavor. Dumping this huge amount of extra disorder into systems is a major reason why enterprise systems fail so regularly.

This article has been excerpted from my book Avoiding IT Disasters: Fallacies about enterprise systems and how you can rise above them (It reveals the real reasons why enterprise systems are failing — the ones that no one wants to talk about).

Written by:

Lance Gutteridge

Dr. Lance Gutteridge has a PhD in computability theory. Presently CTO of Formever Inc. (www.formever.com) where he architects ERP authoring software.