Trigger Hell

January 6, 2019

About a year ago, I interviewed a SQL Server DBA for a project that I was working on. Part of the interview went exactly like this:

> Me: *When do you use triggers?*
>
> DBA Candidate: *Never.*
>
> Me: *You're hired!*

Not only did this guy provide the right answer to my question, he did so in a very convincing manner. His response had a very subtle "are you an idiot?" tone, which was impressive because many candidates take a weak, politically correct line in order not to offend. In this case, this particular guy had no idea what I thought of triggers, so he was taking a chance both with his answer and his tone of voice. In any case, I appreciated his honesty and conviction, so we hired him. Needless to say, we got along very well since I shared his dim view of triggers.

One of the worst code bases I ever saw was about 5 years ago. In fact, I'd rank it second worst in the pantheon of bad code bases that I've seen. (Number 1, as you may recall, I referred to in a previous blog ("Debuggability is a Design Principle"). Interestingly, one of the reasons this code base was so bad was because it was almost impossible to debug. The database not only had hundreds of triggers, but nested triggers to boot. When there was a data problem (and there were many), it usually involved wading through a tangled web of nested triggers to figure out which trigger was causing the problem. It was painful to say the least.

I'll concede that early in my career, I was seduced by triggers. In fact, I used them in a complex, data-intensive app that I developed for the phone company. Unfortunately, as the app grew, it became clear that triggers were a difficult-to-debug albatross. The good news is that I learned my lesson and never used triggers again. In fact, I've gone over twenty years without ever using a trigger. And while it's true that a case may exist for a trigger in certain situations, there's almost always a more robust alternative. In any case, a good developer should always consider other alternatives if they are thinking about using a trigger.

As I've mentioned many times in the past, I've been hired frequently to assist with complex code bases that needed some TLC. Often, these code bases are older and often include a healthy dose of triggers written by developers who probably didn't know any better. I was on one of these projects recently and I had debug several bad/missing data issues. Before I get into the specifics of that though, allow me to digress for a moment. As I've said many times, data issues are almost always more difficult to deal with than code issues. That's why I'm such a strong advocate of prioritizing the database, rejecting ORM, etc. I'm bringing this up again here for two reasons: 1. It is relevant to this discussion of triggers because triggers are often the culprit behind bad data issues (or if not the culprit, then the obfuscator), and 2. Since

debuggability is a crucial design principle (again, refer to the previous blog), triggers are one of the worst violators of good design.

Getting back to the specifics of this project that I was working on, I can't tell you how many times I was initially flabbergasted by some of the data results I was seeing. For example, I would run a query or stored proc and I'd see results that were wrong or made no sense. Then, it would hit me. I bet the problem is an effing trigger! And often, of course, it was. At that point, the debugging exercise got a lot uglier, and my blood pressure a lot higher.

Let me give you another recent example. I was on a project where a bad data condition was discovered in the production database. It was a serious issue that required an immediate fix. I analyzed the issue and wrote the SQL that needed to be executed to fix the problem. Then, I ran the SQL against the production database (in a transaction, of course) and I got back the expected message: "x records affected". Great. The only problem was that I got the "records affected" message three times. Rollback! Of course, it was a trigger that was calling another stored procedure, updating some other tables somewhere else. I had to stop at that point and wade through the trigger to make sure that my SQL wasn't causing deleterious side effects elsewhere. The story ended well, but further stoked my disdain for triggers.

The point is this: Triggers are almost never the answer. Avoid them.