Obergefell v. Hodges: the database engineering perspective

2015-06-27 by qntm

Previously

All the computer systems that exist exist to support requirements and processes arising in the real world. In particular, this includes databases intended to store personal information about human beings, such as who is married to whom, when they got married, when (if ever) they got divorced, and so on. Accurately storing this information is important because marriage often confers real-world legal privileges and obligations to the married people. It is also often important to the married people themselves that their union be public, and well-recorded, and recognised by the organisation doing the recording. The responsibility for accurately and reliably storing this information falls in part on the database administrator (DBA).

Each jurisdiction in the world imposes its own constraints upon who is allowed to marry whom. These constraints can be arbitrarily wide or narrow, but nevertheless they always exist. Because computer systems are not just structureless masses of information, marriage databases also impose constraints upon what marriages can or cannot be stored inside them. As a result, one of the responsibilities of the DBA is to ensure that their database is structured to be capable of storing all possible legal marriages. It may also be the DBA's responsibility to ensure that the database is *incapable* of storing *illegal* marriages (or combinations of marriages)... however, given that the laws surrounding this question can be arbitrarily complicated, the extent of this responsibility may be limited to only very simple rules, such as "nobody may marry themselves", with the remaining responsibility falling on application programmers and/or real people entering the data.

Today (2015-06-26)'s SCOTUS ruling changes the constraints of marriage across a great deal of the United States. This change is long-anticipated and the first marriages under the new constraints will have already happened by the time I finish writing this. This means DBAs (and many other people!) are going to have to react quickly to adjust their systems to accommodate marriages which, previously, there was no need to accommodate. And, depending on how the old systems were set up, and how well DBAs have foreseen this requirement, this may be an easy job, or a difficult one.

In my 2008 essay Gay marriage: the database engineering perspective I went through more than a dozen database schemas for storing marriages, gradually iterating towards more and more flexibility. That essay starts with some laughably stupid schemas and ends up considering some dubiously probable futures, such as scenarios where Alice is married to Bob but Bob is not married to Alice, or where Alice and Bob are somehow *double-married*, such that after a divorce, they would merely be married.

To investigate the specific ramifications of today's ruling, however, here's the schema we're probably starting with:

```
people
- id
- birthdate
- name
- gender

marriages
- id
- husband_id
- wife_id
- marriage_date
- divorce_date
```

Already the constraints on a schema like this are quite complicated. `husband_id` and `wife_id` are both foreign keys for column `people.id`. Check constraints ensure that the value of `marriages.husband_id` always points to a `people` row with `gender` set to "male" and the value of `marriages.wife_id` always points to a row with `gender` set to "female". (Exactly how the `gender` column should be structured is outside the scope of this essay, but the values "male" and "female", at least, should be available. Structuring the `name` column is even further out of scope, because yikes.) `divorce_date` is nullable. Probably there ought to be another check constraint which ensures that `divorce_date` doesn't come before `marriage_date`.

It might be required to incorporate some sort of check for duplicate combinations of `husband_id` and `wife_id`... but then again, this could make it impossible for a couple to e.g. marry in 1994, separate in 2009 and then remarry in 2015.

That's what we've got. Naturally, any of these constraints could be implemented at the application level instead, or even manually by the person entering the marriage data in the first place, and some of them probably *should* be implemented in this way.

Now the ruling comes through and we need to support same-sex marriage. Naively this would be a relatively simple matter. First, rename a few columns:

```
people
- id
- birthdate
- name
- gender

marriages
- id
- partner_1_id
- partner_2_id
- marriage_date
- divorce_date
```

And then, drop the check constraints for `gender`.

All done? Well, maybe. You almost certainly broke some applications by renaming those columns, but leaving them with their old names would be regrettable — and possibly deeply insulting to many couples, as well as highly confusing to newcomer DBAs. Those applications are going to need rewriting anyway.

A few edge cases are suddenly freed up. Married and divorced individuals may now change gender at will; previously, the database would forbid this due to the violated check constraint. In fact, it's no longer necessary to store gender information for the purposes of this now-missing constraint, and the column `people.gender`, and all the data in it, could in theory be dropped entirely.

But the more interesting thing is that you just incidentally let in a whole bunch of edge cases. Up until now, it wasn't possible for an individual to marry themself. Now it is, and you need a new check constraint to ensure that `partner_1_id` and `partner_2_id` are different. Regardless of concerns about duplicate rows/couples remarrying, you also now have to contend with swapped partners: Alice marries Eve, and also *Eve marries Alice*, resulting in two rows recording the same marriage. This can typically be prevented by ensuring that `partner_2_id` is greater than `partner_1_id`, which would incidentally also prevent self-marriage as described above. Note that this could in turn invalidate previously-existing heterosexual marriages where the `husband_id` was lower than the `wife_id`. This constraint would have to be applied for future inserts only, or the disordered rows would need to be swapped.

Overall it looks like a relatively straightforward day's work for a well-prepared DBA with cooperative application developers. Of course, these are big assumptions! The difficulty of making any given alteration to a system is unbounded.

MongoDB, NoSQL and structureless data

NoSQL wasn't a thing when I wrote the original essay, but now it is.

This kind of change is apparently trivial in NoSQL databases because these are explicitly *not* relational databases, and lack structure, foreign keys, check constraints and so on. In theory there is nothing stopping a NoSQL DBA from creating a "marriages" collection and then just adding any kind of data to it. This includes self-marriages, convoluted weighted, directed, polygamous graphs, things not involving humans, structures not recognisable as marriages at all, and so on.

Of course in practice, it is well-understood that for NoSQL databases, the structure is still there but it is *implicit*, imposed by the way applications interact with the database. The analogy is like liquid filling a complicatedly-shaped vessel. The shape taken by the fluid (the data in the database) is determined by the shape of vessel (structure and behaviour of the applications).

In other words, in this situation, the DBA has nearly nothing to do. It's the application developers who need to pick up the slack.

Lessons learned?

This change was seemingly simple but had a little subtlety. Like any change in requirements, the difficulty of handling this particular new requirement is inversely correlated with one's preparedness. In the case of same-sex marriage in US states where it was previously illegal, it should have been blindingly obvious for years that this moment was coming.

This isn't about being closed-minded or homophobic or anything like that, it's just a matter of domain knowledge: if you manage a database full of *X* then you should probably keep your ear to the ground regarding new developments in the field of *X*, particularly if those developments are likely to hit you and require immediate action within hours of developing. I don't want to cast being a bad/ill-prepared database administrator as some kind of moral failing. (I say this largely to cover my own back because I, myself, am a terrible database administrator. In fact, full disclosure: I am not now, and have never been, a database administrator of any kind. All of the above advice could well be complete gibberish.)

Looking into the realistic future, it seems as if the next major change in this area is going to relate to polygamy. Some marriage DBAs reading this (if there actually are any?) may already be familiar with the challenges and constraints associated with storing polygamous marriages. Are there Mormons in your state? Boom.

These changes will be more complex than the ones we've seen here, and I'm not going to go over them for several reasons. Firstly, I kind of already did. Secondly, it doesn't strike me as being so imminent at the time of writing... although, as I mentioned, keep your ear to the ground. Thirdly, supporting a highly complex database at a time when none of that complexity is needed or used can be tiring and unrewarding, particularly if it turns out that that complexity is never going to be needed.

But fourthly, it's always possible that *something else* could happen instead, some new legislative development which requires breaking your formerly-perfect marriage database down and building it up again into a new shape.

You can't fully prepare for that kind of thing because you don't know what it's going to be. But there are decisions you can make now which can make future changes easier (or harder) to make. Get used to making schema changes, botching them, rolling them back, losing no data, and trying again. Make backups.

Because marriage is a big deal. By definition it is a far bigger deal to the people getting married than it is to you, the person who takes care of the computer which logs the event. **After all the formidable legislative barriers have fallen, you don't want to be the jerk who raises a trivial technological barrier to replace them.**

And in case it isn't obvious: all of this generalises massively. All of us who create and maintain computer systems are simultaneously creating implicit and explicit restrictions on how those systems are used, and what kind of data they can store, and what users can do with them. Careless design can make the lives of users difficult or even nightmarish, in ways which are genuinely important.

And while carelessness can be forgivable, and a failure to prepare for all possible futures is unavoidable, not fixing the problem in a timely fashion — or even not being *able* to fix it — is a very unhappy place for both you and the user to be.