Postgres schema changes are still a PITA

Table locking, downtime risks, long running migrations, manual steps, difficult rollbacks. Is there a better way?

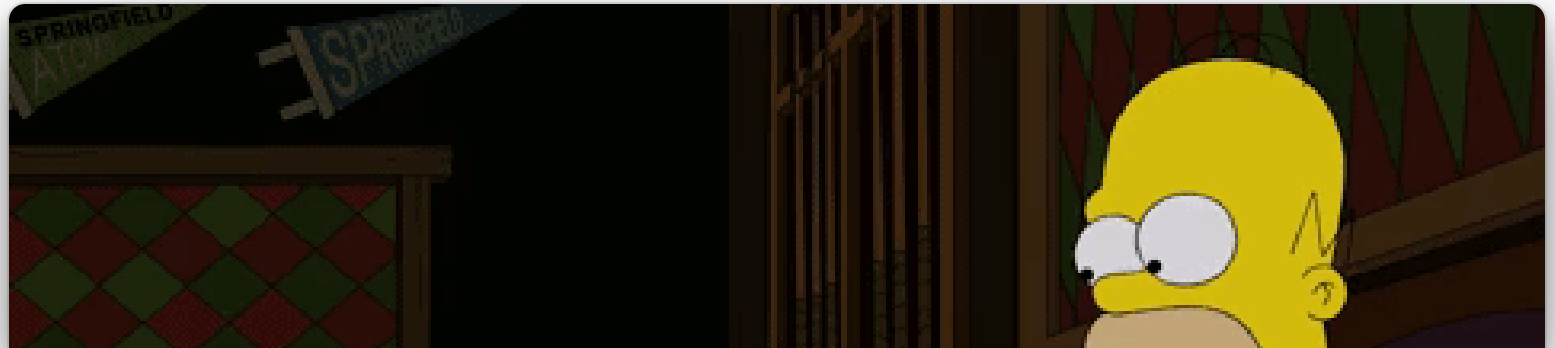
Written by
Tudor Golubenco

Published on
June 29, 2023

We software engineers don't agree on much, but we agree on this one: database schema changes are a pain in the a**.

Part of my job at Xata is to talk with as many developers as possible - from fresh bootcamp graduates, to indie developers, to principal engineers working in large teams. We talk about databases in general, what issues they face, the tools they use, and so on.

From the people that we've talked with, almost everyone said that schema changes and schema management are one of their *least favorite* parts of working with databases. While this sentiment is pretty universal, the reasons that they bring up are not always the same. Small companies or developers working on hobby projects, for example, have to make lots of changes as their applications grow and they discover new requirements. They'd like their schema changes workflow to be as simple and straight-forward as pull requests on GitHub.



For larger companies, with more data and high-traffic tables, schema changes may happen less often, but they still need to worry about things like downtime caused by locking. They require long internal guides on performing schema changes correctly (e.g. GitLab, PayPal), custom tools (e.g. Meta, Square), and they often document incidents or near-incidents caused by schema migrations (e.g. GitHub, Doctolib, GoCardless).

Across the board, developers complain about schema changes affecting their **velocity**: they require more communication, more steps, come with backwards compatibility concerns. As a result, some developers never modify and never remove columns, they only add new ones. This creates “schema-debt”- which creates bugs, confuses new team members, and keeps compatibility code around way past its use-by date.

Issues with schema changes

The following are specific to PostgreSQL, because this is the database system we use at Xata, but some of them apply to other database systems as well.

Locking gotchas

PostgreSQL has many different types of locks and most `ALTER TABLE` statements (but not all) take the table `ACCESS EXCLUSIVE` lock, which conflicts with all other lock types. This means that the table is essentially inaccessible and it’s important to keep this lock for the minimum possible duration.

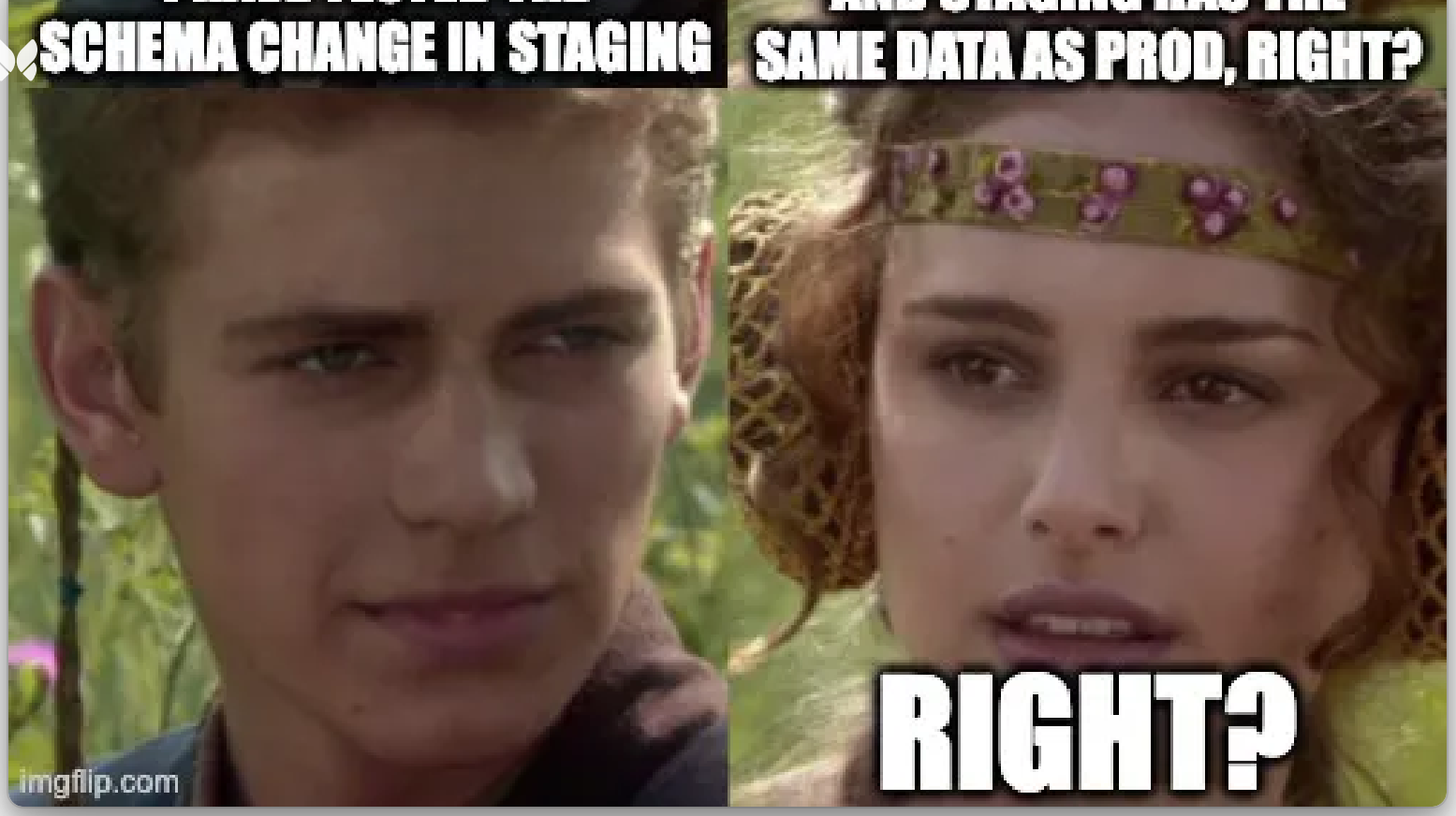
Even when the lock is taken for a short time, it can still make the table seem unavailable. During normal operations, reads and writes run on your tables concurrently. However, when an

`ALTER TABLE` query requiring `ACCESS EXCLUSIVE` runs, Postgres needs to create an opening where no other queries or transactions are running. To achieve this, all queries issued after the `ALTER TABLE` are put in a queue to run after the `ALTER TABLE` completes. Here's the issue: if your table has a running query or a running transaction, your schema change cannot begin. While Postgres is waiting to run your schema change, all of the queries that you would've assumed would run instantly are now being queued up. This gives your table the appearance of being unavailable.

The trick to avoid the above is to explicitly take the lock before running the `ALTER TABLE` and set a `lock_timeout` to avoid queueing queries for a long period of time. If the timeout hits, you keep retrying until the lock succeeds, and only then perform the `ALTER TABLE`.

There are other gotchas as well, like using the magic keyword `CONCURRENTLY` when adding indexes, but that doesn't work inside transactions. When adding a constraint like `NOT NULL`, Postgres needs to first check that there are no NULLs in the table, and it has to do that while holding the lock. You can work around it by adding a `CHECK CONSTRAINT` with `NOT VALID`, which means that the constraint is applied to new rows but not to old ones. Then you can run `VALIDATE` later, which doesn't need the `ACCESS EXCLUSIVE` lock because it assumes new rows are respecting the constraint.

In short, PostgreSQL offers good ways to control locking and avoid keeping locks for a long time, but it's fair to say that it is a minefield; it's easy to make mistakes, and every mistake can be costly. For this reason, teams need to have internal docs on how to do it correctly, and be extra careful during reviews. Staging systems are helpful to catch some of the gotchas, as long as they have the exact same data as the prod database and similar levels of traffic, which is typically not the case.



Application deploys and the 6 stages of rename

The previous section mostly applies to teams that have large tables, but this one applies to any team that cares about things not breaking. It's also not Postgres specific.

Schema changes don't happen in isolation, they are part of a new feature or fix that includes application code changes. If we could deploy the schema change and the application code to all application servers at the exact same moment, this wouldn't be an issue. However, in practice, there's a window of time where the old code needs to work with the new schema, or the other way around.

The strategy to do things correctly depends on the change you'd like to make. For example, if you add a column: you would first perform the schema change, backfill the data, then deploy the application code. After, you would apply constraints, which itself can be a multiple-step process.

If you want to remove a column, it's the other way around. You first make sure no code is accessing that column, then you drop it from the schema.

What about renames? You would follow these steps (hat tip to the PlanetScale docs):

1. Create a new column with the new name.
2. Update and deploy the application to write data to both columns.
3. Backfill missing data from the old column to the new column.
4. Optionally, add constraints like `NOT NULL` to the new column once all the data is backfilled.
5. Update the application to only use the new column, and remove any references to the old column name.
6. Drop the old column.

In summary, in order to make schema changes correctly, even ignoring locking issues, you often need a multi-step process. This is both slowing you down and has a large surface area for expensive mistakes to be made.

Rollback? Rollback your expectations

If there's one thing scarier than performing schema changes, it's the thought that you might have to undo them under time constraints.

If schema changes require careful consideration and a multi-step process, rolling them back is not any simpler. You typically have to carefully apply the same steps in the reverse order.

Because this is complicated and takes a long time, most of us tend to not test rollbacks before starting a schema migration. This makes it doubly scary; you're operating an incident, with your database in an unknown state, and now you need to run through a set of untested steps in production.

For the disciplined developers out there who did test their rollbacks, you may still be stuck waiting a long time for your rollback to complete. This can leave you staring at a terminal for minutes or hours, waiting for your application to come back online for your users.

What if we weren’t so afraid of schema changes?

One of my favorite parts about working on Xata is that we can take this type of workflow issues and reconsider them from first principles.

Let’s imagine that we have a magic wand. How would your ideal schema management system look like? This is mine:

- There is a **standard workflow** that is followed every time, regardless of the schema type.
- Schema changes are **lock-free** or take table locks for minimal amounts of time.
- Schema changes **don’t cause downtime** by breaking application code.
- You can apply all types of changes in a **single-step**, or at most a couple of steps.
- Schema changes are **undoable**, and the undo is quick.

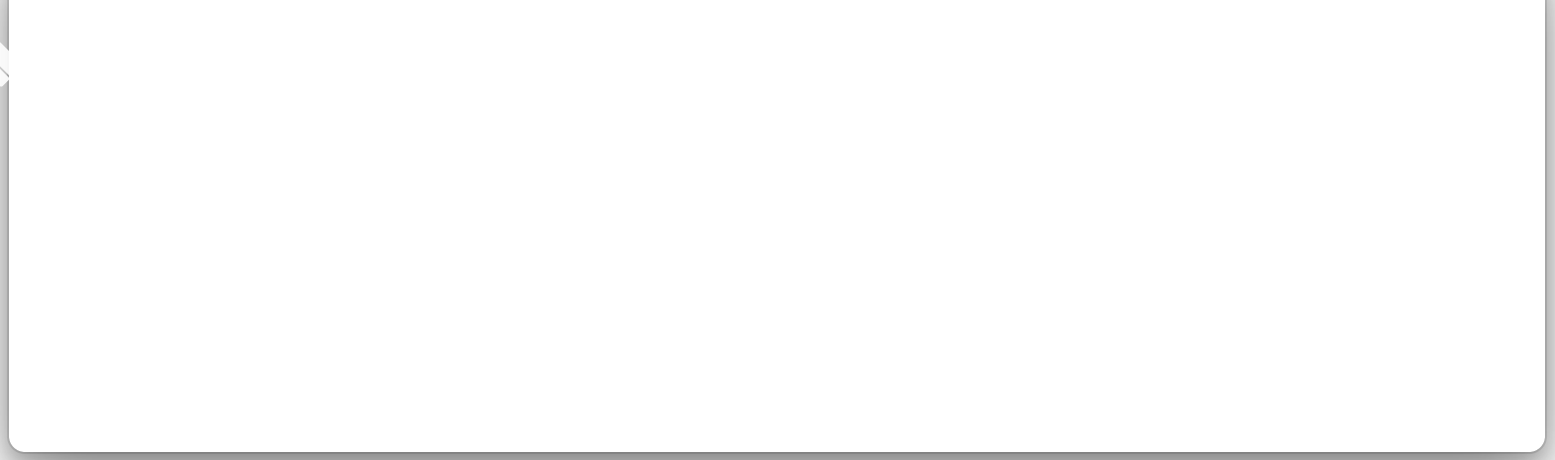
In short, a system that makes schema changes **standardized**, **zero-downtime, lock-free, one-step, and undoable.**

Is this possible?

The first insight is that when it comes to schema changes, we put a lot of onus on the application code in order to keep the database simple. If we move the complexity on the database side, we implement it once and all applications benefit from it.

Instead of having to make the application code forwards/backwards compatible, we make the database schema backwards compatible. The database could serve both the old schema (before the change) and the new schema (after the change) simultaneously. You can apply the schema change, and both old code and new code can work in parallel until the rolling upgrade is complete.

Putting it on a timeline, it would look like this:



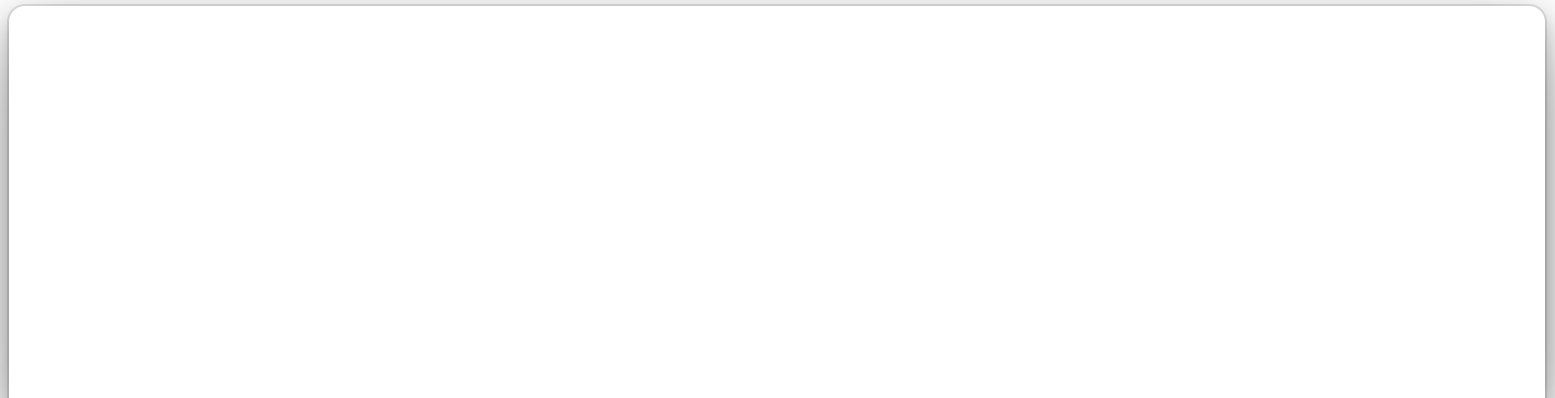
The timeline of a schema change + application roll-out.

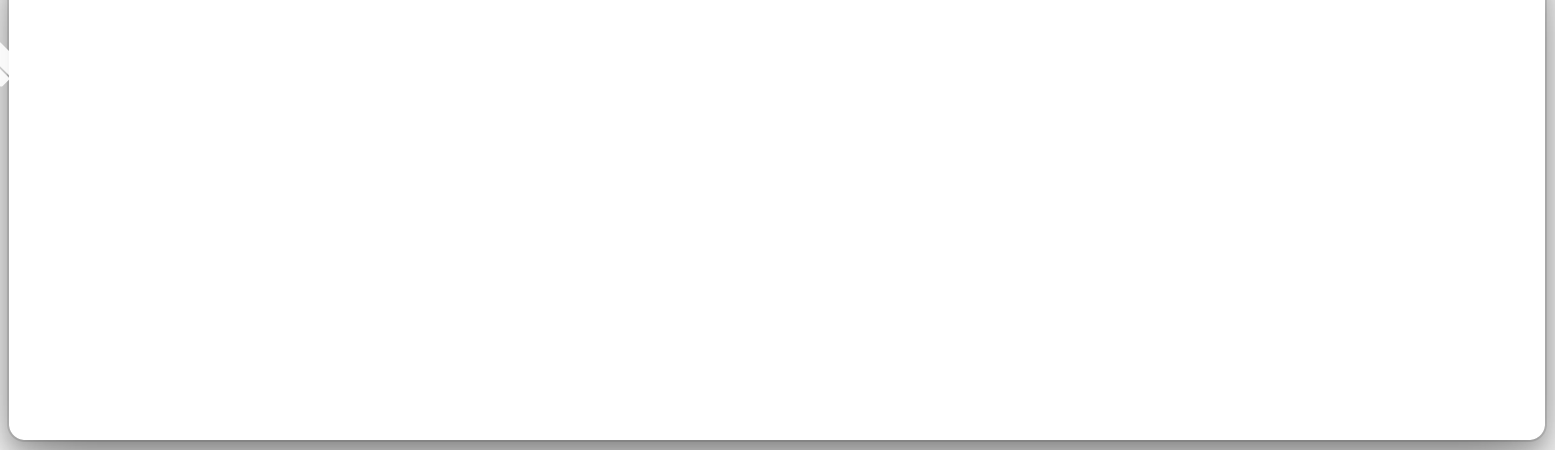
The timeline of a schema change + application roll-out.

1. The schema change is started, for example when the PR is merged. It can take some time until the new schema is available, so the application deployment doesn't yet start.
2. Once the new schema is ready, the application deployment starts. It can be a rolling restart, so the old and the new version of the application might coexist for a period of time. That is ok because the old schema is still available.
3. Once the application deployment is completed, the old schema can be deleted. However, you might want to leave it live for a while longer, in case a rollback is needed.

If a rollback is needed while the old schema is still available, you can safely roll-back the application code. From its point of view, the schema was never modified.

During the time period in which both the old and the new schema are valid, the system keeps temporary hidden columns, and uses views to represent the old and the new schema. New inserts and updates are automatically "upgraded" or "downgraded" between the two schemas, so both the old and the new versions of the application can work normally.





Views are used to represent the old and the new schema to the application.

With the above, the basic column operations (add/remove/rename) all become standardized and safe. No more having to schedule schema changes, no more avoiding renames, or postponing deleting unused columns for weeks “just to be sure”.

Adding constraints is more challenging to implement, because the old and the new schema can be in conflict. For example, let’s say you are adding a `NOT NULL` constraint on a column. If a new `INSERT` comes over the old schema with a `NULL` value, it needs to be accepted, because it respects the old schema. However, the resulting row cannot be exposed in the new schema, because it wouldn’t respect the constraint. In this case, it’s best to hide the new row in the new schema, and block the migration from completing until this issue is solved.

From what I know, there is only one project that tries something close to this: the relatively recent Reshape. It uses Postgres views to expose the two versions of the schema and triggers to upgrade/downgrade the new data. It doesn’t do the constraints part as described above, but shows that this approach is possible. Combined with the Xata pull request based workflow, I think the ideal system described above is possible!

Next Steps

If you’re feeling the pain of schema changes and think that there must be a better way, we’d love to talk to you! We plan to work on this in the form of an open-source project for PostgreSQL, and if you want to be notified when we release it, you can follow us on Twitter, join us on Discord, or simply sign up for Xata.

Comment on Hacker News.

PRODUCT

Roadmap

Feature requests

Pricing

Status

AI solutions

COMPANY

About

Careers

Blog

RSS

CONTACT

Email

Support

Discord

Twitter

GitHub

LEGAL

Cookies

Security

Terms

Privacy policy