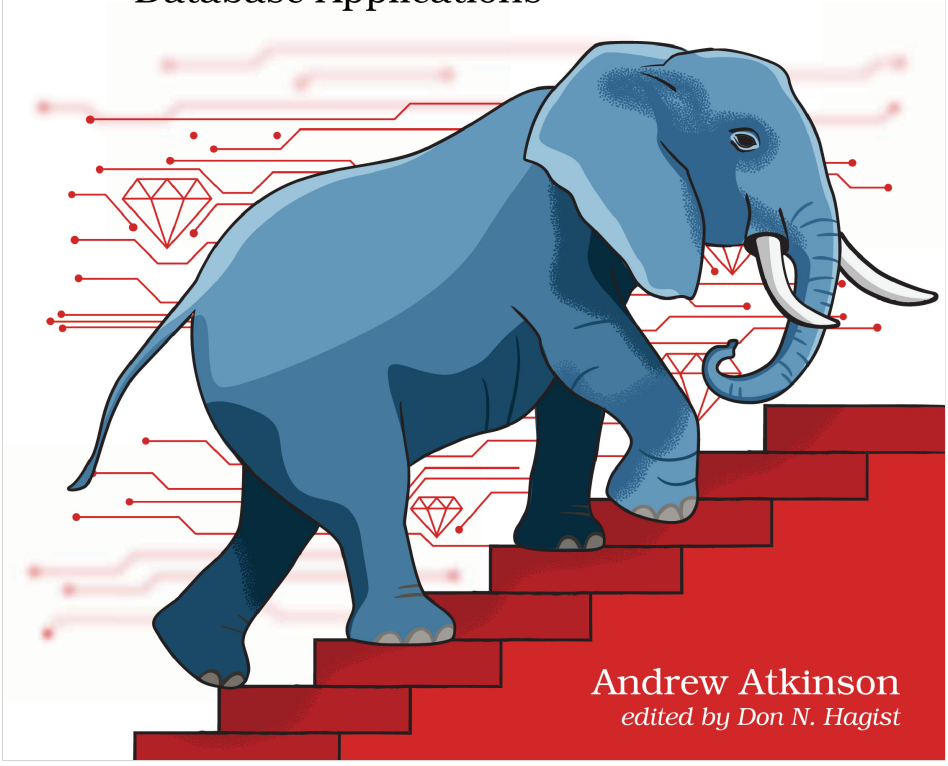
The Pragmatic Programmers

High Performance PostgreSQL for Rails

Reliable, Scalable, Maintainable Database Applications



Andrew Atkinson
edited by Don N. Hagist

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

CHAPTER 5

Modifying Busy Databases Without Downtime

Over time, application codebases and database structures evolve. Your development team's velocity depends on being able to deliver new code and database changes quickly. Rails developers use Active Record migrations to evolve the database structure, creating DDL changes at a steady rate alongside application code changes.

When your application is new and query volume is low, modifying database structures has little to no risk. Since tables are locked for changes for a brief period, with low row counts and low numbers of concurrent users, this brief lock period is no big deal.

However, for applications that are successful and have grown in data volume and query volume, with increased numbers of concurrent sessions, making on-the-fly database structural changes adds more risk for downtime to concurrent user sessions contending for the same data.

One of the challenges you'll face in this phase is continuing to evolve your database structure at the same rate while avoiding risk and downtime.

How do you identify problematic database changes and avoid those pitfalls? You'll learn how to do that in this chapter.

Besides structural changes, you may also perform large-scale *data backfills* or *data migrations*. Backfills populate one or more empty columns, using queries and performing updates during a short period, where they run in high volume. Large backfill operations *also* carry risk to the running operations for concurrent sessions. For that reason, we'll include big backfills in the scope of riskier operations to design safeguards for.

With those goals in mind, let's look at some of the terminology you'll work with:

Busy Databases Terminology

- Strong Migrations—Library that adds safety to migrations
- Multiversion Concurrency Control (MVCC)—Mechanism for managing row changes and concurrent access
- ACID—Set of Guarantees that PostgreSQL makes including Atomicity, Consistency, Isolation, Durability
- Isolation levels—Configurable access level for transactions
- Denormalization—Duplicating some data for improved access speed
- Backfilling—Populating new empty columns for a new table design
- Table rewrites—Internal changes from schema modifications that cause a significant availability delay

You've learned some changes are dangerous at higher levels of scale. What are those and how do we find them?

Identifying Dangerous Migrations

As an experienced Rails developer and PostgreSQL user, you've likely had migrations that didn't run correctly. These might have resulted in failed deployments that blocked releases. You'd like to detect these earlier and avoid failed migrations.

One solution would be to take your database offline to perform changes. Structural changes would be perfectly safe this way!

Unfortunately, that strategy is usually impractical. Modern development teams don't take their databases down for structural changes. Modern teams continually ship code changes and schema changes every day. Fortunately, PostgreSQL can keep pace with these needs, but you'll need to learn some tricks and add processes around detecting riskier scenarios to achieve this.

Safe Migrations

Active Record migrations have no built-in concept of "safety" or "danger." All migrations are treated as being equally safe or unsafe.

Fortunately, third-party gems like Strong Migrations[1] can be added to introduce a concept of safety to your migrations process.

Strong Migrations identifies potentially unsafe migrations, by comparing the migration code to well known problematic patterns. Since it's hooked into the normal migrations process, it prevents unsafe migrations by default. To help developers out, Strong Migrations provides safer alternatives when an unsafe migration is detected.

Let's explore that further.

Learning from Unsafe Migrations

One of the goals in preventing unsafe migrations is to prevent blocking concurrent queries while a structure change is made, which causes application errors.

To learn more about how queries get blocked, you'll simulate the scenario. The unsafe migration you'll work with adds a "Volatile default" value.

To simulate the effect, you'll create a long-running modification and then run a query in that same time period. You'll use Rideshare and the temp schema that was set up earlier.

Launch psql, connecting to the Rideshare database. To reset the temp schema, use the CASCADE option, which drops any tables in that schema. Then create it again.

```
DROP SCHEMA IF EXISTS temp CASCADE;

CREATE SCHEMA temp;
```

Within the temp schema, create a slimmed down users table with an id and name column.

Populate the table with ten million rows. Use CREATE TABLE AS to create the table and populate it in a single statement.

Use GENERATE_SERIES() to get integer values from one to ten million for the id and to create a unique name.

Toggle timing on using \timing to see how long the operation takes.

This will take a bit of time to populate. Run the following statements:

1. https://github.com/ankane/strong_migrations

sql/migration_dangerous_defaults_setup.sql

```
-- Enable timing
\timing

-- create users table (id, name)
-- populate it with 10,000,000 rows
CREATE TABLE temp.users AS
SELECT
  seq AS id,
  'Name-' || seq::TEXT AS name
FROM GENERATE_SERIES(1, 10000000) AS t(seq);
```

Verify the temp.users table structure by running \d temp.users. Confirm it has ten million rows.

Imagine you've made an application change and want to associate users to cities. You'll track a City for the User in a new city_id column. You want both new and existing users to have a city assigned.

In the application, you might prompt users to add their city after adding the column. But you'd like the column to have a default value to start.

Add a new city_id column and give it a default value.

You'll perform this two times to illustrate a difference between how a Volatile and Non-volatile Default works when being added. The first one is relatively safe since it uses a non-volatile, or "static" value. The second version is unsafe on a large table because of the Volatile Default.

Run the following statement to set the Non-volatile value of 1 for all rows.

sql/migration_safe_modification.sql

```
-- add column with a constant default value
-- This is safe to do and runs fairly quick
ALTER TABLE temp.users ADD COLUMN city_id INTEGER
  DEFAULT 1;
```

After adding the column, run \d temp.users and make sure city_id has the Default defined.

Drop the column you just added to prepare to add it a second time.

```
ALTER TABLE temp.users DROP COLUMN city_id;
```

For the next run, it will help to have two different sessions. The following examples have psql1 and psql2 in their name, referring to where you should run the statements.

In the unsafe version, you'll run the ALTER TABLE statement from the psql1 session.

Run that now.

sql/migration_dangerous_modification_psql1.sql

```
-- !!! Dangerous Version !!!
-- Adds a "non-constant" or "volatile" DEFAULT
-- This takes a LOT longer
-- Table is locked in `ACCESS EXCLUSIVE` mode for duration
ALTER TABLE temp.users ADD COLUMN city_id INTEGER
  DEFAULT 1 + FLOOR(RANDOM() * 25);
```

While that's running, from the psql2 session, run the following SQL query, which is normally very quick:

sql/migration_dangerous_modification_psql2.sql

```
SELECT * FROM temp.users LIMIT 1;
```

The DDL in psql1 can take ten seconds or more to run. In that time, the query in psql2 will appear "hung" until psql1 finishes. Once psql1 finishes, psql2 instantly finishes.

psql2 appeared hung because psql1 had locked the table with exclusive access while the Default value was added to all rows. Since there are ten million rows, performing this change took a long time, and the table was locked, even blocking SELECT queries until it finished.

Imagine instead of the single query from psql2, there are hundreds of queries running during that lock period, all getting blocked. This could quickly cause thousands of errors in your application, something you'd like to avoid!

What steps can be put in place to help detect these issues?