Extracted from:

High Performance PostgreSQL for Rails

Reliable, Scalable, Maintainable Database Applications

This PDF file contains pages extracted from *High Performance PostgreSQL for Rails*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2023 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

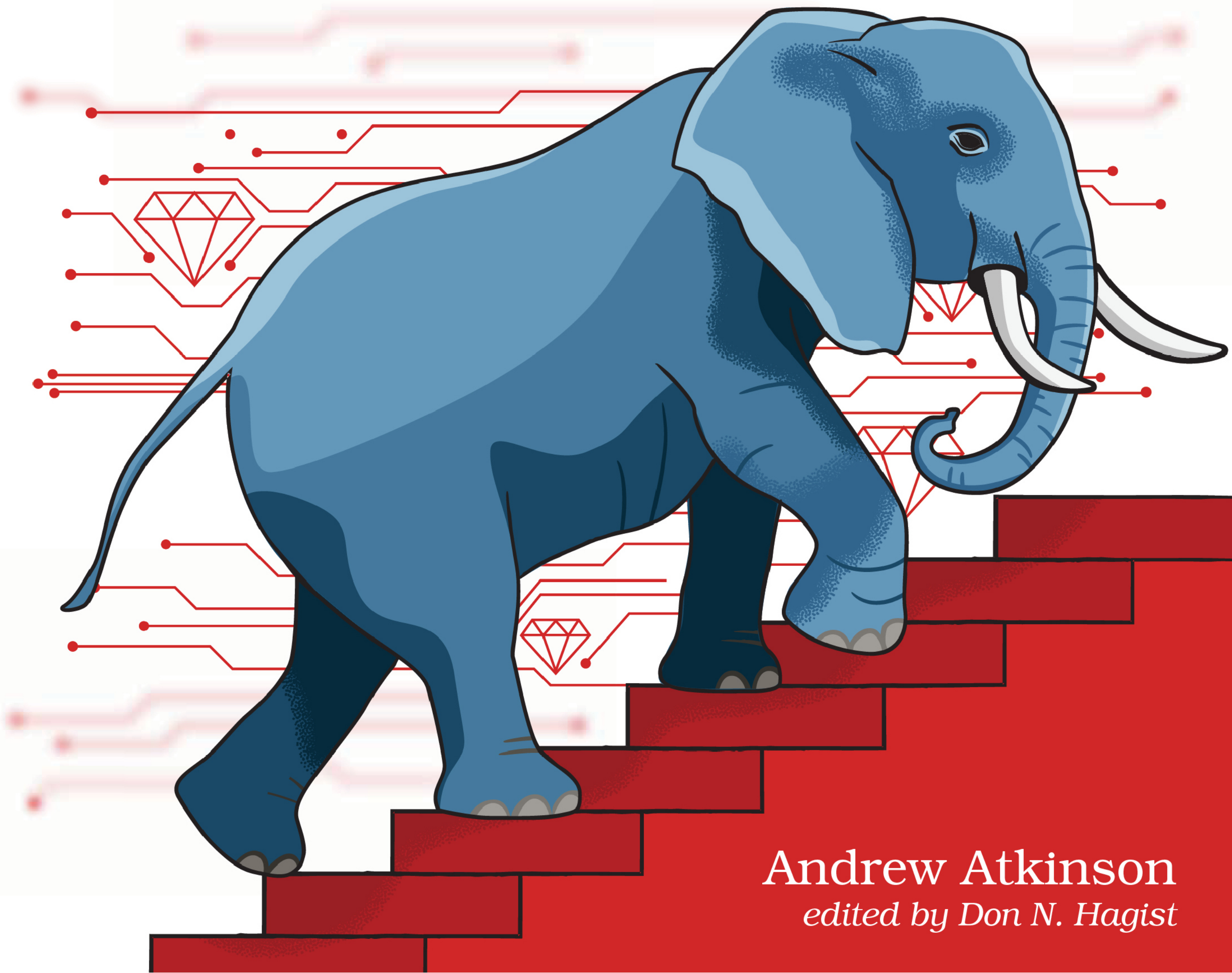
The Pragmatic Bookshelf

Dallas, Texas

The Pragmatic Programmers

High Performance PostgreSQL for Rails

Reliable, Scalable, Maintainable Database Applications



Andrew Atkinson

edited by Don N. Hagist

High Performance PostgreSQL for Rails

Reliable, Scalable, Maintainable Database Applications

Andrew Atkinson

The Pragmatic Bookshelf

Dallas, Texas

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Copyright © 2023 The Pragmatic Programmers, LLC.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

ISBN-13: 979-8-88865-038-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—August 30, 2023

CHAPTER 5

Modifying Busy Databases Without Downtime

Over time your application codebase and database structure will evolve. Part of your development team's velocity hinges on being able to have a short time to delivery for both new code features and incremental database modifications. Rails developers use Active Record Migrations to evolve the database structure, constantly creating DDL changes as needed for features and as a change management mechanism to keep all databases in sync.

When your application is new and usage is low, modifying the database structure incurs almost no risk. Tables that are locked have few rows and lock times are short. There may not be much concurrent activity from customers or internal users. As an application grows in popularity and usage, data growth increases, query volume increases, and your team may begin to feel more pressure from database changes that become riskier and reduce the team development velocity.

One of the main challenges you'll face in evolving your database structure is performing operations that cause table locks, and how to do those safely and quickly.

How do you identify these riskier types of changes and how can you prevent them?

Structural changes from Active Record Migration methods that create ALTER TABLE[1] SQL statements can be riskier for tables with high row counts and high query volume. Most structural changes lock the table in Exclusive Access

1. https://www.postgresql.org/docs/current/sql-altertable.html

mode which means no other transactions can access the resource while the lock is in effect. How can you make any change to a table then for a busy database?

Don't worry. Most of the time the lock durations are short and clients can wait a bit if needed.

Some changes are less safe though, particularly for tables with very high row counts and queried heavily. In this chapter you'll learn to identify which changes are the most risky, and how to work around those risks using safer tactics.

Besides changes to the database structure, you may be conducting large scale *backfill* operations. Backfills populate one or more columns that didn't exist before and are otherwise null, to have useful values for the application.

Backfilling on tables with millions or billions of rows without stopping the server is another type of technical challenge that will be covered in this chapter.

Take a look at some of the terminology you'll be learning.

Busy Databases Terminology

- Strong Migrations — Library that adds more safety to Migrations
- Multiversion Concurrency Control (MVCC) — PostgreSQL mechanism for managing row modifications and concurrent data access
- ACID — Design properties of Atomicity, Consistency, Isolation, Durability
- Isolation Levels — Configurable access level among transactions accessing the same data
- Denormalization — Duplicating some column data
- Backfilling — Populating a recently added empty column on existing table rows
- Table Rewrites — Internal changes from some schema modifications that can cause a significant availability delay

Earlier you learned that some types of Migrations are riskier than others. How can you find the riskier ones?

Identifying Dangerous Migrations

As an experienced Rails developer and PostgreSQL user, you've likely had Migrations that didn't run correctly and possibly blocked your release process. Blocking releases slows down your development team velocity. Besides slowing the team down, Migrations could even cause application errors.

One solution would be to take your database down when you need to make schema changes. With a disconnected database there would be no concurrent modifications to worry about from application queries, so changes would be perfectly safe!

Unfortunately that strategy is impractical. Modern development teams don't take their databases offline to make changes. Modern teams are shipping code changes and evolving SQL database schemas every day, and fortunately PostgreSQL can keep pace with the needs of your team, allowing you to change the structure as often as needed.

How do you detect those scenarios?

When a developer creates an Active Record Migration for a DDL change, there is no built-in concept of safety or danger with Migrations themselves. All migrations are treated equally. This leaves the possibility open of deploying unsafe migrations that can harm team velocity and cause application errors.

Fortunately third party gems like Strong Migrations[2] fill this need. Strong Migrations has been added to Rideshare so you can experiment with it there. The main purpose of Strong Migrations is to identify potentially unsafe migrations during normal development, using well known patterns from DDL changes that can cause trouble. Strong Migrations even presents safer alternatives.

Read on to learn more.

Learning From Unsafe Migrations

To get some hands-on experience with how queries can get blocked, you'll create a long running modification and run a query during that time that gets blocked. You'll use the Rideshare database and the `temp` schema from earlier.

Launch psql from your terminal, connecting to the Rideshare database. To reset the `temp` schema, use the `CASCADE` option which drops any tables in that Schema. Create it again.

```
DROP SCHEMA IF EXISTS temp CASCADE;
```

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

```
CREATE SCHEMA temp;
```

Within the recreated temp schema, create a slimmed down users table with an id and name column. This is a simplified form of a users table.

Next, populate the table with 10 million rows. Use CREATE TABLE AS to create the table and populate it in one statement.

Use GENERATE_SERIES() to get Integer values from 1 to 10 million for the id and to create a unique name.

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

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

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 users table structure with \d temp.users and confirm it has 10 million rows.

Imagine you've made an application change and want to associate Users to Cities using a city_id column. You want all new and existing Users to have a City assigned by default.

In the application you might prompt users to change their default Cities, but you'd like to start with a default value.

To accomplish this, you'll add a new city_id column and give it a default value. This value will then be used for all new rows being inserted, but will also be applied retroactively to all 10 million existing rows.

For long time PostgreSQL users, adding a Default column value carries some baggage. Column defaults have been made mostly safe to perform on newer versions of PostgreSQL but there are still some gotchas.

Open up a second psql session connected to the Rideshare database. The example below uses "psql1" and "psql2" to refer to the first and second sessions.

• Click HERE to purchase this book now. discuss

In the first session add the default value and while that's being added, you'll run a query in the second session. Running a query simulates an application query running during a structural change.

You'll add the default value two times, a different way each time, demonstrating an important difference between them. The first one is relatively safe and the second one would be likely unsafe for a table with a lot of rows and in a high activity period.

The main difference is that in the first version, the column Default has a static value. This is very fast to add. Run the statement below which adds a static value of "1" to 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 observe the `city_id` column and Default value.

Drop the column you just added to prepare for adding it a second time. You can run this from the "psql1" session:

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

Next is the unsafe version. In this version you'll run the ALTER TABLE statement, adding the column in the "psql1" session again.

Run that now.

sql/migration_dangerous_modification.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, try selecting a single row from the same `temp.users` table in the "psql2" session. Run this SQL query:

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

The DDL query in "psql1" might take ten seconds or more to run. The very simple query in "psql2", trying to select one row, will appear "hung" for the whole time until the modification in "psql1" completes, and then the query in "psql2" will finish instantly.

Now imagine this scenario in production for a table with hundreds of millions of rows, and thousands of queries per second trying to read from this table. This would be a disaster and cause huge amounts of errors, so it's something you'll want to avoid!

What steps can be put in place to help detect and avoid this type of scenario?