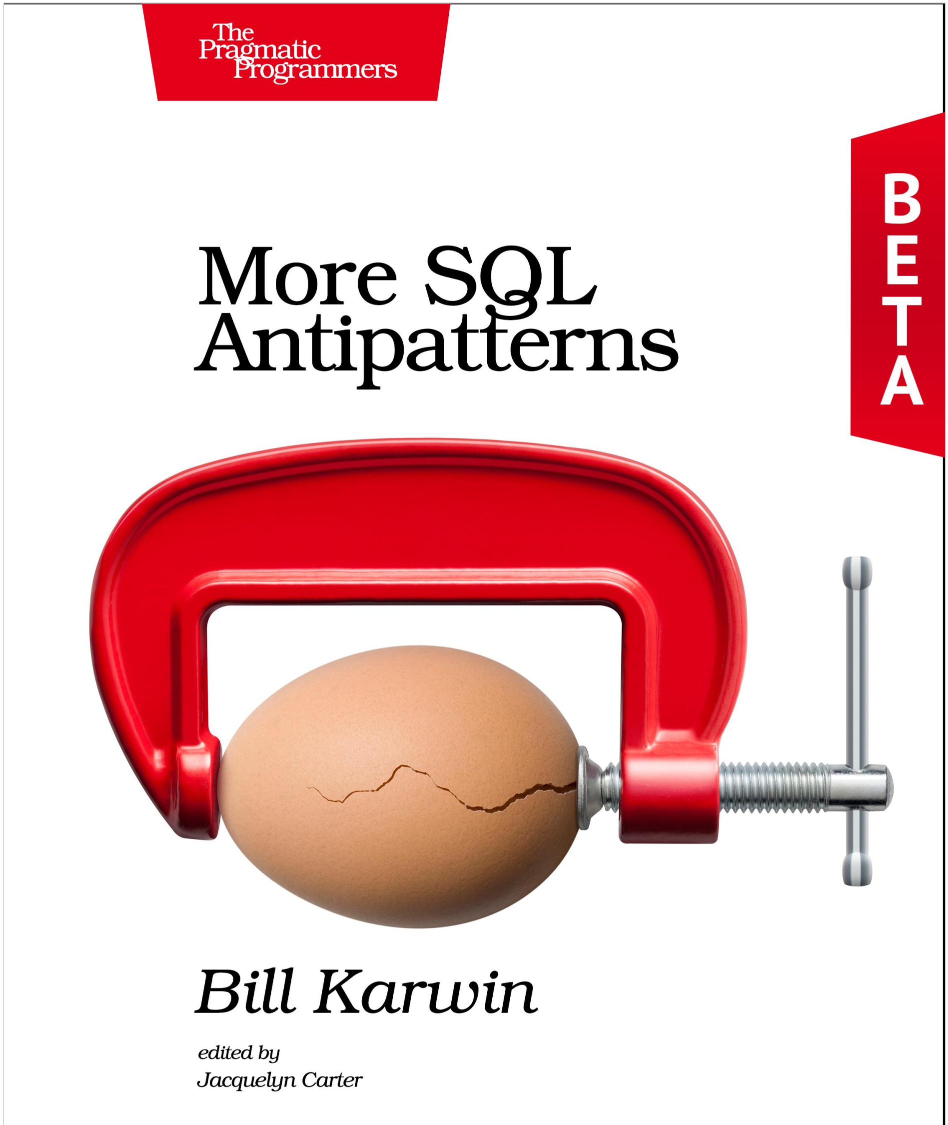

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

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

Siobhán's face appears over your monitor. She's the manager of the group that engineers reports and monitoring. "Got a moment? We apparently have a problem in the database," she says. Next to her is J.P., the Director of Finance. You suddenly sense that your answer to Siobhán's question better be yes.

"How can I help?" you ask, showing that you are giving her your full attention.

Siobhán begins, "I recall you stored the quarterly total revenue in the database."

"Right, that was to optimize the sales dashboard," you explain. "As sales are booked, the total is recalculated, and the dashboard reads the total. It was too slow when we queried the whole table every time the dashboard refreshed. Is the dashboard too slow again?"

"No, the dashboard is quick enough," she continues. "But the total doesn't match the sales data." J.P. is looking at you with a stern expression on their face.

You gradually remember that the dashboard code was recently refactored by an engineer who left the company suddenly last week. He changed a lot of code, and his changes slipped through the normal code review process. Since it was only the dashboard, no one felt it was worth fighting over enforcing that process.

"Okay, we can add a backlog item to revisit the dashboard code," you suggest, trying to avoid blame.

"You don't understand," the J.P. says. "My team used the number on the dashboard for our revenue reporting, which has already gone out. Now that we discovered that it's way off from the true total, we have to refile that report. The finance team will have to work through the weekend, but that's not the worst part. The company is being scrutinized by stock market analysts, and this unexpected change makes it look like we're deliberately reporting inaccurate numbers."

Siobhán explains, "It doesn't matter whether the change goes up or down, it will probably affect our stock price."

The Finance Director nods, and they add, "And it may trigger a full audit."

Objective: Breaking the Rules to Improve Performance

As always, it's about performance. Developers seek any method they can use to optimize their database, even if it means breaking the rules in creative ways.

In a relational database, the rules include the rules of normalization, or designing tables and columns in a way that minimizes redundancy and reduces the chance of data anomalies. Read more about normalization in the appendix "Rules of Normalization" in *SQL Antipatterns Volume 1 [Kar22]*.

Rules can be broken, and if doing so means even a slight improvement to performance, it's easy to justify. This is where *denormalization* can provide value, if used wisely.

If not used wisely, any method of optimization—including denormalization—can cause new problems. That's what this chapter is about.

Antipattern: Breaking the Rules Recklessly

"Normalize until it hurts, denormalize until it works," has been written many times, for example in Jeff Atwood's 2008 blog "Maybe Normalizing Isn't Normal."[1] A well-chosen exception to the rules might give big rewards. The problem is that when a trick works well once, you're tempted to use it more and more. In the case of denormalization, some software developers start to prefer it over the standard practice, and they use it too liberally.

Let's look at examples of forms of denormalization to show the problems you can cause.

Avoiding Joins, Part 1

Chapter 2, Fear of JOINs, on page ? showed that joins can be optimized, and it's a good way to return attributes of related tables. But reading two or more rows from different tables is more costly than reading one row from a single table (this depends on the SQL implementation, but it's true in most OLTP databases products). If you've tried to improve a join query with indexes and it's still too inefficient, you might be tempted to store an attribute in the table you mean to join to, instead of the table where rules of normalization would guide you to put that attribute.

1. https://blog.codinghorror.com/maybe-normalizing-isnt-normal/

For example, suppose you need to write an SQL query to show a post with its author's name. The user name is normally stored in the user table, but that makes you join the tables.

Denormalization/anti/normal-join.sql
```
SELECT p.post_title, u.display_name
FROM post AS p JOIN "user" AS u USING (user_id)
WHERE p.post_id = 1234;
```

So you try to move the user name into the post table. You can visualize what you want to do: the name that exists for a given user_id in the user table, copy it to a new column in the respective row of the post table that references that user.

"post"

post_id	user_id	author_display_name
1234	3	groucho

"user"

user_id	display_name
3	groucho
4	chico
5	harpo

In SQL, that operation looks like the following:

Denormalization/anti/avoid-join.sql
```
ALTER TABLE post ADD COLUMN author_display_name VARCHAR(64);

UPDATE post SET author_display_name = 'Groucho'
WHERE post_id = 1234;
```

Now you can query the author name from the post table, without doing a join.

Denormalization/anti/avoid-join.sql
```
SELECT p.post_title, p.author_display_name
FROM post AS p
WHERE post_id = 1234;
```

But if the user wants to change their display name, it becomes more work to update it. You'd have to update the user table, as you normally would, and potentially many rows in the post table.

You could create a trigger to update copies of the display name in the other table when the user changes it. Then you might wonder if there are other tables in this database that have their own copy of the user display name. Your trigger would have to be coded to update all those other tables. Everything is possible, but the point is that the more you denormalize, the more extra work you have to do to maintain those copies.

An alternative would be to skip changing the display name for old posts. Only new posts written after the name change would get the new name. That would make the same user appear using different display names in different posts, which would be confusing for users of Visage.

Avoiding Joins, Part 2

A post in Visage can be assigned one or more tags to help readers search for posts they're interested in. So you have a requirement to display the tags for a given post. Posts and tags have a many-to-many relationship. That is, a post may have multiple tags, and a given tag may apply to multiple posts. In a normalized database, you'd create a third table, and save each pairing as a row. Now you have to make *two* joins to show a post with its tags.

Denormalization/anti/normal-join-many.sql
```sql
SELECT p.post_title, t.tag
FROM post AS p LEFT JOIN post_tag AS pt USING (post_id)
LEFT JOIN tag AS t USING (tag_id)
WHERE p.post_id = 1234;
```

If you could denormalize by storing the list of tags in the post table, you'd skip those joins, and get the list directly when you display the post. Suppose you copied a set of tag id's from multiple rows into a single row, as illustated by the following diagram:

In SQL, you write it like the following:

Denormalization/anti/avoid-join-many.sql
```sql
-- Add the column once
ALTER TABLE post ADD COLUMN tag_list VARCHAR(64);

-- Set the list each time you add or remove a tag
UPDATE post SET tag_list = 'tag1,tag2,tag5'
WHERE post_id = 1234;

-- Then subsequently get the list any time you query
SELECT p.post_title, p.tag_list
FROM post AS p
WHERE post_id = 1234;
```

This leads to new problems. The first is that the items in the comma-separated string don't necessarily match the values in the original post_tag table. If you were to add a new tag or remove a tag for a given post, you'd have to update both tables, and if you forget, then the lists will be out of sync. The problem is you can't tell which is correct—the comma-separated string, or the rows in the post_tag table. Which one failed to be updated? Was a new tag added to post.tag_list, or was a row deleted from post_tag?

You could solve this by eliminating the post_tag table. Storing the tags for a post in two places risks them getting out of sync, so make sure the tags for a given post are stored in one place. This causes a new problem: if you need to search for posts by tag instead of by primary key, you can't easily optimize searching for substrings in a comma-separated list. PostgreSQL has support for arrays, and an operator && that tests if two arrays overlap. But this is not supported in other brands of SQL, and it also can't be optimized with an index. The following example shows how you could write the query in PostgreSQL:

Denormalization/anti/avoid-join-many-by-tag.sql

```
SELECT p.post_title, p.tag_list
FROM post AS p
WHERE regexp_split_to_array(p.tag_list, ',') && '{"tag5"}';
```

If you still had the normalized table, you would write the following simpler query that's easier to optimize:

Denormalization/anti/normal-join-many-by-tag.sql

```
SELECT p.post_title, t.tag
FROM post AS p JOIN post_tag AS pt USING (post_id)
JOIN tag AS t USING (tag_id)
WHERE t.tag_id = 5;
```

In addition, there are many problems with storing values as comma-separated lists. For example, the length of the VARCHAR creates an unnecessary limit to the number of tags. The items in the string aren't constrained to be valid tags. The string itself isn't even constrained to be a comma-separated list. See the chapter "Jaywalking" in *SQL Antipatterns Volume 1 [Kar22]* for the full story.