Extracted from:

SQL Antipatterns, Volume 1

Avoiding the Pitfalls of Database Programming

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

SQL Antipatterns, Volume 1

Avoiding the Pitfalls of Database Programming

Bill Karwin

edited by Jacquelyn Carter

SQL Antipatterns, Volume 1

Avoiding the Pitfalls of Database Programming

Bill Karwin

The Pragmatic Bookshelf
Raleigh, North Carolina

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Jacquelyn Carter
Copy Editor: Karen Galle
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

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

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

ISBN-13: 978-1-68050-898-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—November 2022

To my wife Jan, my best supporter.

A Netscape engineer who shan't be named once passed a pointer to JavaScript, stored it as a string, and later passed it back to C, killing 30.

➤ *Blake Ross*

CHAPTER 2

Jaywalking

You're developing a feature in the bug-tracking application to designate a user as the primary contact for a product. Your original design allowed only one user to be the contact for each product. However, it was no surprise when you were requested to support assigning multiple users as contacts for a given product.

At the time, it seemed simple to change the database to store a list of user account identifiers separated by commas, instead of the single identifier it used before.

Soon your boss approaches you with a problem. "The engineering department has been adding associate staff to their projects. They tell me they can add five people only. If they try to add more, they get an error. What's going on?"

You nod, "Yeah, you can only list so many people on a project," as though this is completely ordinary.

Sensing that your boss needs a more precise explanation, "Well, five to ten—maybe a few more. It depends on how old each person's account is." Now your boss raises his eyebrows. You continue, "I store the account IDs for a project in a comma-separated list. The list of IDs has to fit in a string with a maximum length. If the account IDs are short, I can fit more in the list. So, people who created the earlier accounts have an ID of 99 or less, and those are shorter."

Your boss frowns. You have a feeling you're going to be staying late.

Programmers commonly use comma-separated lists to avoid creating an intersection table for a many-to-many relationship. This antipattern is called *Jaywalking*, because jaywalking is also an act of avoiding an intersection.

Objective: Store Multivalue Attributes

When a column in a table has a single value, the design is straightforward: you can choose an SQL data type to represent a single instance of that value, for example an integer, date, or string. It's not clear how you store a collection of related values in a column.

In the example bug-tracking database, you might associate a product with a contact using an integer column in the Products table. Each account may have many products, and each product references one contact, so there's a *many-to-one* relationship between products and accounts.

Jaywalking/obj/create.sql

```
CREATE TABLE Products (
  product_id   SERIAL PRIMARY KEY,
  product_name VARCHAR(1000),
  account_id   BIGINT UNSIGNED,
  -- . . .
  FOREIGN KEY (account_id) REFERENCES Accounts(account_id)
);

INSERT INTO Products (product_id, product_name, account_id)
VALUES (DEFAULT, 'Visual TurboBuilder', 12);
```

As your project matures, you realize that a product might have multiple contacts. In addition to the many-to-one relationship, you also need to support a one-to-many relationship from products to accounts. One row in the Products table must be able to have more than one contact.

Antipattern: Format Comma-Separated Lists

To minimize changes to the database structure, you decide to redefine the account_id column as a VARCHAR so you can list multiple account IDs in that column, separated by commas.

Jaywalking/anti/create.sql

```
CREATE TABLE Products (
  product_id   SERIAL PRIMARY KEY,
  product_name VARCHAR(1000),
  account_id   VARCHAR(100), -- comma-separated list
  -- . . .
);

INSERT INTO Products (product_id, product_name, account_id)
VALUES (DEFAULT, 'Visual TurboBuilder', '12,34');
```

This seems like a win, because you've created no additional tables or columns; you've changed the data type of only one column. However, let's look at the performance and data integrity problems this table design suffers from.

Querying Products for a Specific Account

Queries are difficult if all the foreign keys are combined into a single field. You can no longer use equality; instead, you have to use a test against some kind of pattern. For example, MySQL lets you write something like the following to find all the products for account 12:

Jaywalking/anti/regexp.sql
```
SELECT * FROM Products WHERE account_id REGEXP '\\b12\\b';
```

Pattern-matching expressions may return false matches. Performance is poor because the matching can't benefit from indexes. Since pattern-matching syntax is different in each database brand, your SQL code isn't vendor neutral.

Querying Accounts for a Given Product

Likewise, it's awkward and slow to join a comma-separated list to matching rows in the referenced table.

Jaywalking/anti/regexp.sql
```
SELECT * FROM Products AS p JOIN Accounts AS a
    ON p.account_id REGEXP '\\b' || a.account_id || '\\b'
WHERE p.product_id = 123;
```

Joining two tables using an expression like this one spoils any chance of using indexes, so again the performance will suffer. The query must scan through both tables, generate a cross product, and evaluate the regular expression for every combination of rows.

Making Aggregate Queries

Aggregate queries use functions like COUNT(), SUM(), and AVG(). However, these functions are designed to be used over groups of rows, not comma-separated lists. You have to resort to tricks, like calculating the length of the string of comma-separated values minus the length of that string with the commas removed. This can be used to count the elements in the list.

Jaywalking/anti/count.sql
```
SELECT product_id,
    LENGTH(account_id) - LENGTH(REPLACE(account_id, ',', '')) + 1
      AS contacts_per_product
FROM Products;
```

Tricks like this can be clever but never clear. These kinds of solutions are time consuming to develop and hard to debug. Some aggregate queries can't be accomplished with tricks at all.

Updating Accounts for a Specific Product

You can add a new ID to the end of the list with string concatenation, but this might not leave the list in sorted order.

Jaywalking/anti/update.sql
```
UPDATE Products
SET account_id = account_id || ',' || 56
WHERE product_id = 123;
```

To remove an item from the list, you have to run two SQL queries: one to fetch the old list and a second to save the updated list.

Jaywalking/anti/remove.py
```
import mysql.connector

cnx = mysql.connector.connect(user='scott', database='test')
cursor = cnx.cursor()

product_id_to_search = 2
value_to_remove = '34'

query = "SELECT product_id, account_id FROM Products WHERE product_id = %s"
cursor.execute(query, (product_id_to_search,))
for (row) in cursor:
    (product_id, account_ids) = row
    account_id_list = account_ids.split(",")
    account_id_list.remove(value_to_remove)
    account_ids = ",".join(account_id_list)
    query = "UPDATE Products SET account_id = %s WHERE product_id = %s"
    cursor.execute(query, (account_ids, product_id,))

cnx.commit()
```

That’s quite a lot of code just to remove an entry from a list.

Validating Product IDs

What prevents a user from entering invalid entries like *banana*?

Jaywalking/anti/banana.sql
```
INSERT INTO Products (product_id, product_name, account_id)
VALUES (DEFAULT, 'Visual TurboBuilder', '12,34,banana');
```

Users will find a way to enter any and all variations, and your database will turn to mush. There won’t necessarily be database errors, but the data will be nonsense.

Even if the values are at least integers, you can’t be sure they are integers that occur in the Accounts table. The standard way to ensure this is to use a foreign key constraint, but foreign keys can only validate the whole column, not individual elements in a list.

Choosing a Separator Character

If you store a list of string values instead of integers, some list entries may contain your separator character. Using a comma as the separator between entries may become ambiguous. You can choose a different character as the separator, but you can't guarantee that this new separator will never appear in an entry.

List Length Limitations

How many list entries can you store in a VARCHAR(30) column? It depends on the length of each entry. If each entry is two characters long, then you can store ten (including the commas). If each entry is six characters, then you can store only four entries:

Jaywalking/anti/length.sql

```sql
UPDATE Products SET account_id = '10,14,18,22,26,30,34,38,42,46'
WHERE product_id = 123;

UPDATE Products SET account_id = '101418,222630,343842,467790'
WHERE product_id = 123;
```

How can you know that VARCHAR(30) supports the longest list you will need in the future? How long is long enough? Try explaining the reason for this length limit to your boss or to your customers.

How to Recognize the Antipattern

If you hear phrases like the following spoken by your project team, treat it as a clue that the Jaywalking antipattern is being employed:

- "What is the greatest number of entries this list must support?"

  This question comes up when you're trying to choose the maximum length of the VARCHAR column.

- "Do you know how to match a word boundary in SQL?"

  If you use regular expressions to pick out parts of a string, this could be a clue that you should store those parts separately.

- "What character will never appear in any list entry?"

  You want to use an unambiguous separator character, but you should expect that any character might someday appear in a value in the list.