Extracted from:

SQL Antipatterns

Avoiding the Pitfalls of Database Programming

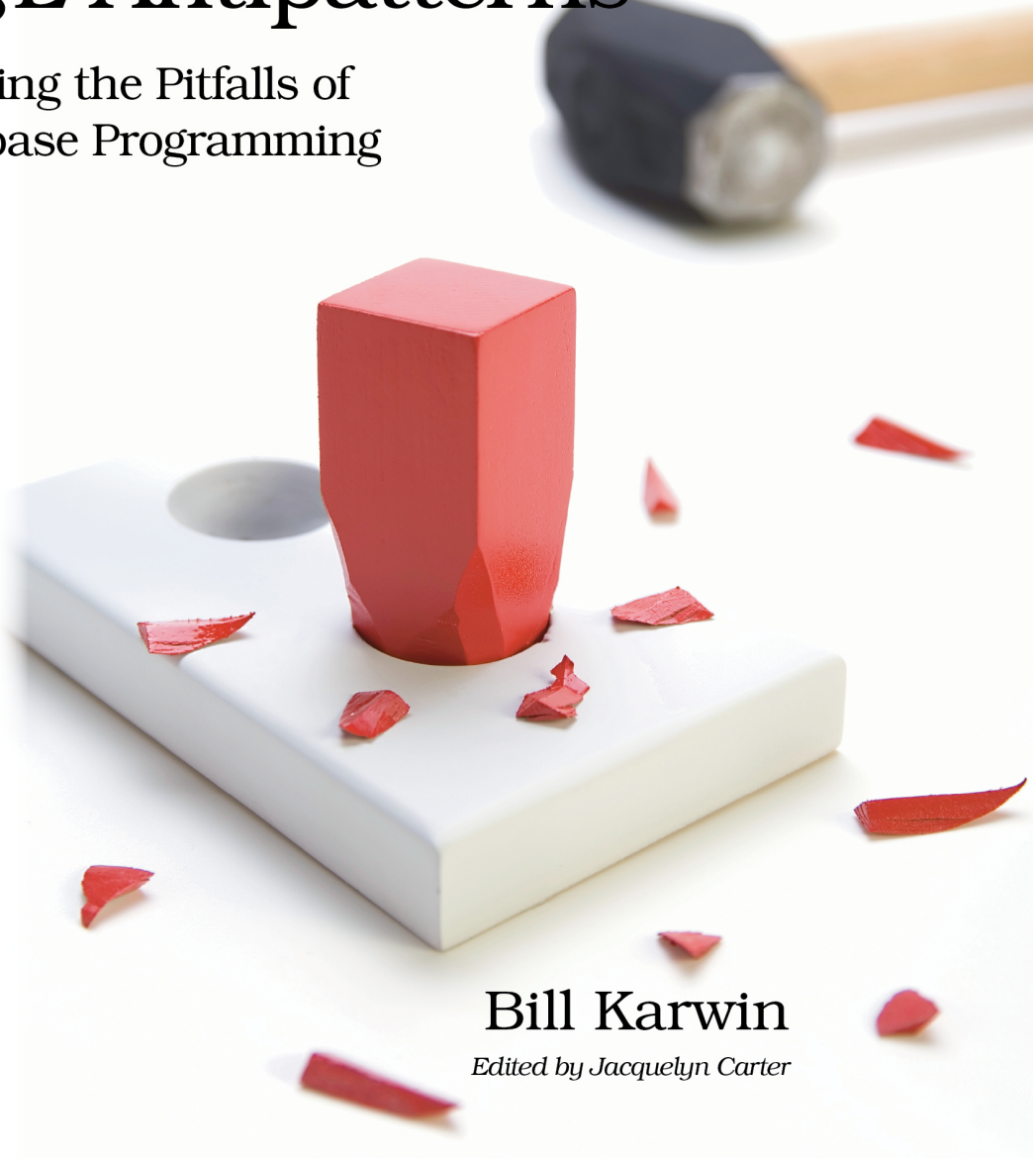
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

SQL Antipatterns

Avoiding the Pitfalls of Database Programming



Bill Karwin

Edited by Jacquelyn Carter

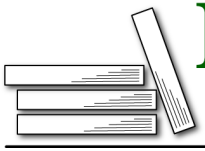
SQL Antipatterns

Avoiding the Pitfalls of Database Programming

Bill Karwin

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-934356-55-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P3.0—March 2012

An expert is a person who has made all the mistakes that can be made in a very narrow field.

➤ *Niels Bohr*

CHAPTER 1

Introduction

I turned down my first SQL job.

Shortly after I finished my college degree in computer and information science at the University of California, I was approached by a manager who worked at the university and knew me through campus activities. He had his own software startup company on the side that was developing a database management system portable between various UNIX platforms using shell scripts and related tools such as `awk` (at this time, modern dynamic languages like Ruby, Python, PHP, and even Perl weren't popular yet). The manager approached me because he needed a programmer to write the code to recognize and execute a limited version of the SQL language.

He said, "I don't need to support the full language—that would be too much work. I need only one SQL statement: `SELECT`."

I hadn't been taught SQL in school. Databases weren't as ubiquitous as they are today, and open source brands like MySQL and PostgreSQL didn't exist yet. But I had developed complete applications in shell, and I knew something about parsers, having done projects in classes like compiler design and computational linguistics. So, I thought about taking the job. How hard could it be to parse a single statement of a specialized language like SQL?

I found a reference for SQL and noticed immediately that this was a different sort of language from those that support statements like `if()` and `while()`, variable assignments and expressions, and perhaps functions. To call `SELECT` only one statement in that language is like calling an engine only one part of an automobile. Both sentences are literally true, but they certainly belie the complexity and depth of their subjects. To support execution of that single SQL statement, I realized I would have to develop all the code for a fully functional relational database management system and query engine.

I declined this opportunity to code an SQL parser and RDBMS engine in shell script. The manager underrepresented the scope of his project, perhaps because he didn't understand what an RDBMS does.

My early experience with SQL seems to be a common one for software developers, even those who have a college degree in computer science. Most people are self-taught in SQL, learning it out of self-defense when they find themselves working on a project that requires it, instead of studying it explicitly as they would most programming languages. Regardless of whether the person is a hobbyist or a professional programmer or an accomplished researcher with a PhD, SQL seems to be a software skill that programmers learn without training.

Once I learned something about SQL, I was surprised how different it is from procedural programming languages such as C, Pascal, and shell, or object-oriented languages like C++, Java, Ruby, or Python. SQL is a *declarative programming language* like LISP, Haskell, or XSLT. SQL uses *sets* as a fundamental data structure, while object-oriented languages use objects. Traditionally trained software developers are turned off by this so-called *impedance mismatch*, so many programmers are drawn to object-oriented libraries to avoid learning how to use SQL effectively.

Since 1992, I've worked with SQL a lot. I've used it when developing applications, I've developed libraries for SQL programming in Perl and PHP, and I've provided technical support and developed training and documentation for the InterBase RDBMS product. I've answered thousands of questions on Internet mailing lists and newsgroups. I see a lot of repeat business—frequently asked questions that show that software developers make the same mistakes over and over again.

1.1 Who This Book Is For

I'm writing *SQL Antipatterns* for software developers who need to use SQL so I can help you use the language more effectively. It doesn't matter whether you're a beginner or a seasoned professional. I've talked to people of all levels of experience who would benefit from the subjects in this book.

You may have read a reference on SQL syntax. Now you know all the clauses of a `SELECT` statement, and you can get some work done. Gradually, you may increase your SQL skills by inspecting other applications and reading articles. But how can you tell good examples from bad examples? How can you be sure you're learning best practices, instead of yet another way to paint yourself into a corner?

You may find some topics in *SQL Antipatterns* that are well-known to you. You'll see new ways of looking at the problems, even if you're already aware of the solutions. It's good to confirm and reinforce your good practices by reviewing widespread programmer misconceptions. Other topics may be new to you. I hope you can improve your SQL programming habits by reading them.

If you are a trained database administrator, you may already know the best ways to avoid the SQL pitfalls described in this book. This book can help you by introducing you to the perspective of software developers. It's not uncommon for the relationship between developers and DBAs to be contentious, but mutual respect and teamwork can help us to work together more effectively. Use *SQL Antipatterns* to help explain good practices to the software developers you work with and the consequences of straying from that path.

1.2 What's in This Book

What is an antipattern? An *antipattern* is a technique that is intended to solve a problem but that often leads to other problems. An antipattern is practiced widely in different ways, but with a thread of commonality. People may come up with an idea that fits an antipattern independently or with help from a colleague, a book, or an article. Many antipatterns of object-oriented software design and project management are documented at the Portland Pattern Repository,[1] as well as in the 1998 book *AntiPatterns* [BMMM98] by William J. Brown et al.

SQL Antipatterns describes the most frequently made missteps I've seen people naively make while using SQL as I've talked to them in technical support and training sessions, worked alongside them developing software, and answered their questions on Internet forums. Many of these blunders I've made myself; there's no better teacher than spending many hours late at night making up for one's own errors.

Parts of This Book

This book has four parts for the following categories of antipatterns:

Logical Database Design Antipatterns
: Before you start coding, you should decide what information you need to keep in your database and the best way to organize and interconnect your data. This includes planning database tables, columns, and relationships.

1. Portland Pattern Repository: http://c2.com/cgi-bin/wiki?AntiPattern

Physical Database Design Antipatterns
: After you know what data you need to store, you implement the data management as efficiently as you can using the features of your RDBMS technology. This includes defining tables and indexes and choosing data types. You use SQL's *data definition language*—statements such as `CREATE TABLE`.

Query Antipatterns
: You need to add data to your database and then retrieve data. SQL queries are made with *data manipulation language*—statements such as `SELECT`, `UPDATE`, and `DELETE`.

Application Development Antipatterns
: SQL is supposed to be used in the context of applications written in another language, such as C++, Java, PHP, Python, or Ruby. There are right ways and wrong ways to employ SQL in an application, and this part of the book describes some common blunders.

Many of the antipattern chapters have humorous or evocative titles, such as *Golden Hammer*, *Reinventing the Wheel*, or *Design by Committee*. It's traditional to give both positive design patterns and antipatterns names that serve as a metaphor or mnemonic.

The appendix provides practical descriptions of some relational database theory. Many of the antipatterns this book covers are the result of misunderstanding database theory.

Anatomy of an Antipattern

Each antipattern chapter contains the following subheadings:

Objective
: This is the task that you may be trying to solve. Antipatterns are used with an intention to provide that solution but end up causing more problems than they solve.

The Antipattern
: This section describes the nature of the common solution and illustrates the unforeseen consequences that make it an anti-pattern.

How to Recognize the Antipattern
: There may be certain clues that help you identify when an antipattern is being used in your project. Certain types of barriers you encounter, or quotes you may hear yourself or others saying, can tip you off to the presence of an antipattern.

Legitimate Uses of the Antipattern
: Rules usually have exceptions. There may be circumstances in which an approach normally considered an antipattern is nevertheless appropriate, or at least the lesser of all evils.

Solution
: This section describes the preferred solutions, which solve the original objective without running into the problems caused by the antipattern.

1.3 What's Not in This Book

I'm not going to give lessons on SQL syntax or terminology. There are plenty of books and Internet references for the basics. I assume you have already learned enough SQL syntax to use the language and get some work done.

Performance, scalability, and optimization are important for many people who develop database-driven applications, especially on the Web. There are books specifically about performance issues related to database programming. I recommend *SQL Performance Tuning* [GP03] and *High Performance MySQL, Second Edition* [SZTZ08]. Some of the topics in *SQL Antipatterns* are relevant to performance, but it's not the main focus of the book.

I try to present issues that apply to all database brands and also solutions that should work with all brands. The SQL language is specified as an ANSI and ISO standard. All brands of databases support these standards, so I describe vendor-neutral use of SQL whenever possible, and I try to be clear when describing vendor extensions to SQL.

Data access frameworks and object-relational mapping libraries are helpful tools, but these aren't the focus of this book. I've written most code examples in PHP, in the plainest way I can. The examples are simple enough that they're equally relevant to most programming languages.

Database administration and operation tasks such as server sizing, installation and configuration, monitoring, backups, log analysis, and security are important and deserve a book of their own, but I'm targeting this book to developers using the SQL language more than database administrators.

This book is about SQL and relational databases, not alternative technology such as object-oriented databases, key/value stores, column-oriented databases, document-oriented databases, hierarchical databases, network databases, map/reduce frameworks, or semantic data stores. Comparing the

strengths and weaknesses and appropriate uses of these alternative solutions for data management would be interesting but is a matter for other books.

1.4 Conventions

The following sections describe some conventions I use in this book.

Typography

SQL keywords are formatted in all-capitals and in a monospaced font to make them stand out from the text, as in `SELECT`.

SQL tables, also in a monospaced font, are spelled with a capital for the initial letter of each word in the table name, as in `Accounts` or `BugsProducts`. SQL columns, also in a monospaced font, are spelled in lowercase, and words are separated by underscores, as in `account_name`.

Literal strings are formatted in italics, as in *bill@example.com*.

Terminology

SQL is correctly pronounced "ess-cue-ell," not "see-quell." Though I have no objection to the latter being used colloquially, I try to use the former, so in this book you will read phrases like "*an* SQL query," not "*a* SQL query."

In the context of database-related usage, the word *index* refers to an ordered collection of information. The preferred plural of this word is *indexes*. In other contexts, an index may mean an *indicator* and is typically pluralized as *indices*. Both are correct according to most dictionaries, and this causes some confusion among writers. In this book, I spell the plural as *indexes*.

In SQL, the terms *query* and *statement* are somewhat interchangeable, being any complete SQL command that you can execute. For the sake of clarity, I use *query* to refer to `SELECT` statements and *statement* for all others, including `INSERT`, `UPDATE`, and `DELETE` statements, as well as data definition statements.

Entity-Relationship Diagrams

The most common way to diagram relational databases is with *entity-relationship diagrams*. Tables are shown as boxes, and relationships are shown as lines connecting the boxes, with symbols at either end of the lines describing the cardinality of the relationship. For examples, see Figure 1, *Examples of entity-relationship diagrams*, on page 7.

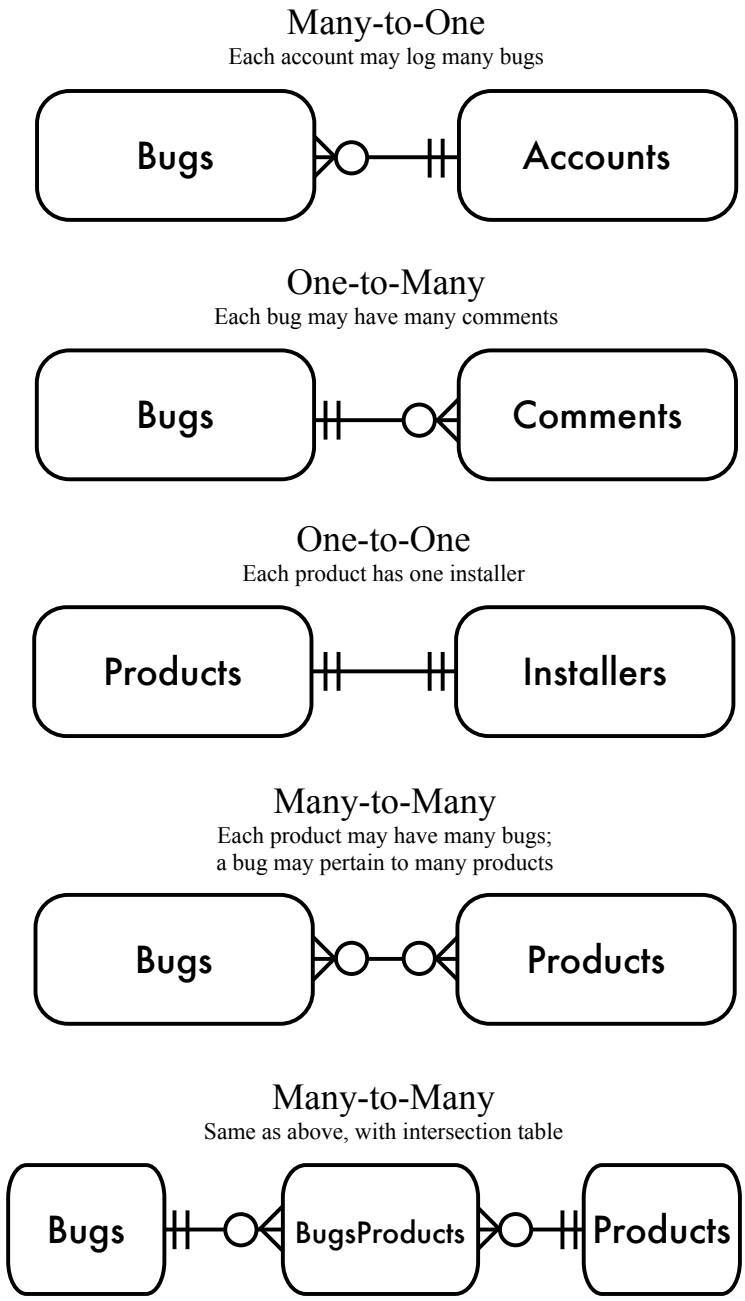


Figure 1—Examples of entity-relationship diagrams

1.5 Example Database

I illustrate most of the topics in *SQL Antipatterns* using a database for a hypothetical bug-tracking application. The entity-relationship diagram for this database is shown in Figure 2, *Diagram for example bug database*, on page 10. Notice the three connections between the Bugs table and the Accounts table, representing three separate foreign keys.

The following data definition language shows how I define the tables. In some cases, choices are made for the sake of examples later in the book, so they might not always be the choices one would make in a real-world application. I try to use only standard SQL so the example is applicable to any brand of database, but some MySQL data types also appear, such as SERIAL and BIGINT.

Introduction/setup.sql

```
CREATE TABLE Accounts (
  account_id        SERIAL PRIMARY KEY,
  account_name      VARCHAR(20),
  first_name        VARCHAR(20),
  last_name         VARCHAR(20),
  email             VARCHAR(100),
  password_hash     CHAR(64),
  portrait_image    BLOB,
  hourly_rate       NUMERIC(9,2)
);

CREATE TABLE BugStatus (
  status            VARCHAR(20) PRIMARY KEY
);

CREATE TABLE Bugs (
  bug_id            SERIAL PRIMARY KEY,
  date_reported     DATE NOT NULL,
  summary           VARCHAR(80),
  description       VARCHAR(1000),
  resolution        VARCHAR(1000),
  reported_by       BIGINT UNSIGNED NOT NULL,
  assigned_to       BIGINT UNSIGNED,
  verified_by       BIGINT UNSIGNED,
  status            VARCHAR(20) NOT NULL DEFAULT 'NEW',
  priority          VARCHAR(20),
  hours             NUMERIC(9,2),
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id),
  FOREIGN KEY (verified_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (status) REFERENCES BugStatus(status)
);
```

```
CREATE TABLE Comments (
  comment_id        SERIAL PRIMARY KEY,
  bug_id            BIGINT UNSIGNED NOT NULL,
  author            BIGINT UNSIGNED NOT NULL,
  comment_date      DATETIME NOT NULL,
  comment           TEXT NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);

CREATE TABLE Screenshots (
  bug_id            BIGINT UNSIGNED NOT NULL,
  image_id          BIGINT UNSIGNED NOT NULL,
  screenshot_image  BLOB,
  caption           VARCHAR(100),
  PRIMARY KEY      (bug_id, image_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);

CREATE TABLE Tags (
  bug_id            BIGINT UNSIGNED NOT NULL,
  tag               VARCHAR(20) NOT NULL,
  PRIMARY KEY      (bug_id, tag),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);

CREATE TABLE Products (
  product_id        SERIAL PRIMARY KEY,
  product_name      VARCHAR(50)
);

CREATE TABLE BugsProducts(
  bug_id            BIGINT UNSIGNED NOT NULL,
  product_id        BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY      (bug_id, product_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

In some chapters, especially those in Logical Database Design Antipatterns, I show different database definitions, either to exhibit the antipattern or to show an alternative solution that avoids the antipattern.

1.6 Acknowledgments

First and foremost, I owe my gratitude to my wife Jan. I could not have written this book without the inspiration, love, and support you give me, not to mention the occasional kick in the pants.

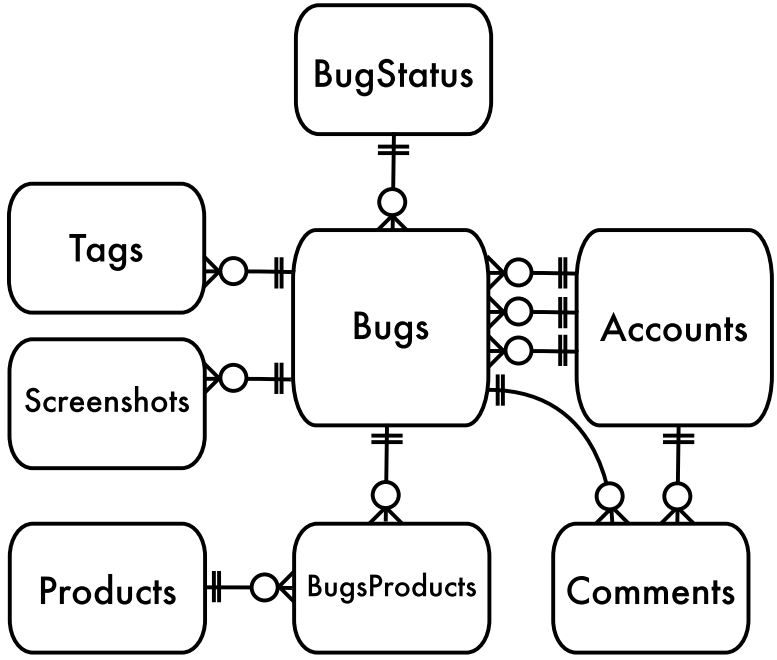


Figure 2—Diagram for example bug database

I also want to express thanks to my reviewers for giving me a lot of their time. Their suggestions improved the book greatly. Marcus Adams, Jeff Bean, Frederic Daoud, Darby Felton, Arjen Lentz, Andy Lester, Chris Levesque, Mike Naberezny, Liz Nealy, Daev Roehr, Marco Romanini, Maik Schmidt, Gale Straney, and Danny Thorpe.

Thanks to my editor Jacquelyn Carter and the publishers of Pragmatic Bookshelf, who believed in the mission of this book.