SQL Antipatterns

Avoiding the Pitfalls of Database Programming

Bill Karwin

What Readers Are Saying About *SQL Antipatterns*

I am a strong advocate of best practices. I prefer to learn from other people's mistakes. This book is a comprehensive collection of those other people's mistakes and, quite surprisingly, some of my own. I wish I had read this book sooner.

▶ **Marcus Adams**
Senior Software Engineer

Bill has written an engaging, useful, important, and unique book. Software developers will certainly benefit from reading the anti-patterns and solutions described here. I immediately applied techniques from this book and improved my applications. Fantastic work!

▶ **Frederic Daoud**
Author of *Stripes: ...And Java Web Development Is Fun Again* and *Getting Started with Apache Click*

SQL Antipatterns is a must-read for software developers, who will frequently encounter the database design choices presented in this book. It helps development teams to understand the consequences of their database designs and to make the best decisions possible based on requirements, expectations, measurements, and reality.

▶ **Darby Felton**
Cofounder, DevBots Software Development

I really like how Bill has approached this book; it shows his unique style and sense of humor. Those things are really important when discussing potentially dry topics. Bill has succeeded in making the teachings accessible for developers in a good descriptive form, as well as being easy to reference later. In short, this is an excellent new resource for your pragmatic bookshelf!

▶ **Arjen Lentz**
Executive Director of Open Query (http://openquery.com);
Coauthor of *High Performance MySQL, Second Edition*

This book is obviously the product of many years of practical experience with SQL databases. Each topic is covered in great depth, and the attention to detail in the book was beyond my expectations. Although it's not a beginner's book, any developer with a reasonable amount of SQL experience should find it to be a valuable reference and would be hard-pressed not to learn something new.

> ► **Mike Naberezny**
> Partner at Maintainable Software; Coauthor of *Rails for PHP Developers*

This is an excellent book for the software engineer who knows basic SQL but finds herself needing to design SQL databases for projects that go a little beyond the basics.

> ► **Liz Neely**
> Senior Database Programmer

Karwin's book is full of good and practical advice, and it was published at the right time. While many people are focusing on the new and seemingly fancy stuff, professionals now have the chance and the perfect book to sharpen their SQL knowledge.

> ► **Maik Schmidt**
> Author of *Enterprise Recipes with Ruby and Rails* and *Enterprise Integration with Ruby*

Bill has captured the essence of a slew of traps that we've probably all dug for ourselves at one point or another when working with SQL — without even realizing we're in trouble. Bill's antipatterns range from "I can't believe I did that (again!)" hindsight gotchas to tricky scenarios where the best solution may run counter to the SQL dogma you grew up with. A good read for SQL diehards, novices, and everyone in between.

> ► **Danny Thorpe**
> Microsoft Principal Engineer; Author of *Delphi Component Design*

SQL Antipatterns

Avoiding the Pitfalls of Database Programming

Bill Karwin

Pragmatic Bookshelf

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 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.

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://www.pragprog.com

Contents

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 provided technical support and developed training and documentation for the InterBase RDBMS product, and I've developed libraries for SQL programming in Perl and PHP. 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.

---

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

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 your database tables, columns, and relationships.

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 antipattern.

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* [SZT$^+$08]. 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 1.1, on the following page.

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 1.2, on page 21. 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.

Download 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)
);
```

Many-to-One
Each account may log many bugs

One-to-Many
Each bug may have many comments

One-to-One
Each product has one installer

Many-to-Many
Each product may have many bugs;
a bug may pertain to many products

Many-to-Many
Same as above, with intersection table
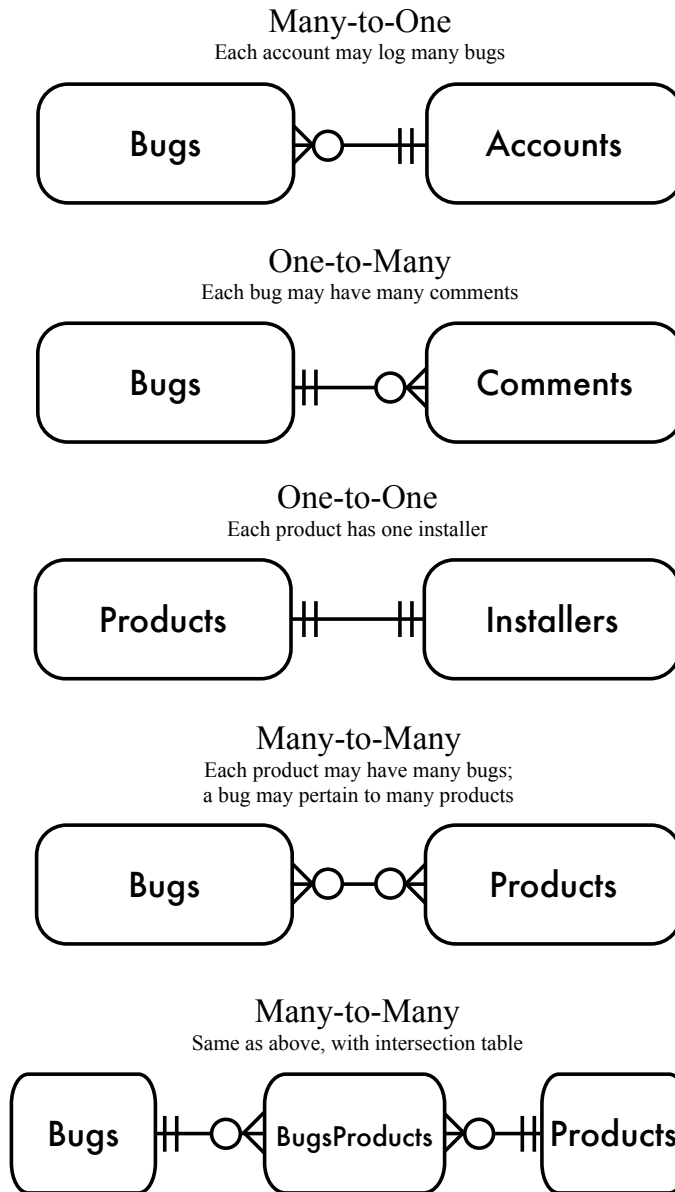
Figure 1.1: Examples of entity-relationship diagrams
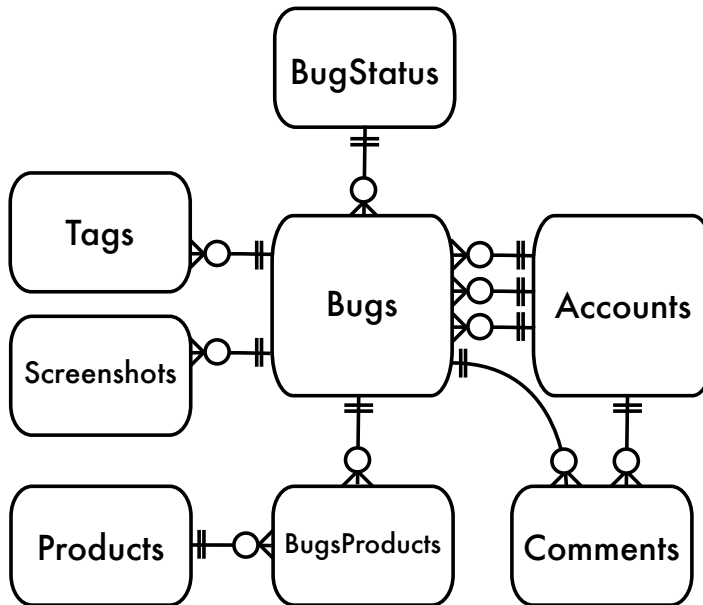
Figure 1.2: Diagram for example bug database

```sql
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)
);
```

```sql
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 Anti-patterns, I show different database definitions, either to exhibit the antipattern or to show an alternative solution that avoids the anti-pattern.

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.

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.

Part I

Logical Database Design Antipatterns

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 de-
partment 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 ac-
count IDs for a project in a comma-separated list. But 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. I call this anti-
pattern *Jaywalking*, because jaywalking is also an act of avoiding an
intersection.

2.1 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. But how do you store a collection of related values in a column?

In the example bug-tracking database, we 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 we have a *many-to-one* relationship between products and accounts.

Download 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, we 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.

2.2 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.

Download 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 problems 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:

Download Jaywalking/anti/regexp.sql

```sql
SELECT * FROM Products WHERE account_id REGEXP '[[:<:]]12[[:>:]]';
```

Pattern-matching expressions may return false matches and 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 costly to join a comma-separated list to matching rows in the referenced table.

Download Jaywalking/anti/regexp.sql

```sql
SELECT * FROM Products AS p JOIN Accounts AS a
    ON p.account_id REGEXP '[[:<:]]' || a.account_id || '[[:>:]]'
WHERE p.product_id = 123;
```

Joining two tables using an expression like this one spoils any chance of using indexes. 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 the following:

Download Jaywalking/anti/count.sql

```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.

Download 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.

Download Jaywalking/anti/remove.php

```php
<?php

$stmt = $pdo->query(
  "SELECT account_id FROM Products WHERE product_id = 123");
$row = $stmt->fetch();
$contact_list = $row['account_id'];

// change list in PHP code
$value_to_remove = "34";
$contact_list = split(",", $contact_list);
$key_to_remove = array_search($value_to_remove, $contact_list);
unset($contact_list[$key_to_remove]);
$contact_list = join(",", $contact_list);

$stmt = $pdo->prepare(
  "UPDATE Products SET account_id = ?
   WHERE product_id = 123");
$stmt->execute(array($contact_list));
```

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*?

Download 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.

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 can you 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). But if each entry is six characters, then you can store only four entries:

Download Jaywalking/anti/length.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.

2.3 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.

2.4 Legitimate Uses of the Antipattern

You might improve performance for some kinds of queries by apply-ing *denormalization* to your database organization. Storing lists as a comma-separated string is an example of denormalization.

Your application may need the data in a comma-separated format and have no need to access individual items in the list. Likewise, if your application receives a comma-separated format from another source and you simply need to store the full list in a database and retrieve it later in exactly the same format, there's no need to separate the values.

Be conservative if you decide to employ denormalization. Start by using a normalized database organization, because it permits your applica-tion code to be more flexible, and it allows your database to help pre-serve data integrity.

2.5 Solution: Create an Intersection Table

Instead of storing the account_id in the Products table, store it in a sepa-rate table, so each individual value of that attribute occupies a separate row. This new table Contacts implements a *many-to-many* relationship between Products and Accounts:

Download Jaywalking/soln/create.sql

```
CREATE TABLE Contacts (
  product_id  BIGINT UNSIGNED NOT NULL,
  account_id  BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (product_id, account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id),
  FOREIGN KEY (account_id) REFERENCES Accounts(account_id)
);

INSERT INTO Contacts (product_id, accont_id)
VALUES (123, 12), (123, 34), (345, 23), (567, 12), (567, 34);
```

When the table has foreign keys referencing two tables, it's called an *intersection table.*[1] This implements a *many-to-many* relationship be-tween the two referenced tables. That is, each product may be associ-ated through the intersection table to multiple accounts, and likewise each account may be associated to multiple products. See the entity-relationship diagram in Figure 2.1, on the following page.

---

1. Some people use a join table, a many-to-many table, a mapping table, or other terms to describe this table. The name doesn't matter; the concept is the same.

Figure 2.1: Intersection table entity-relationship diagram

Let's see how using an intersection table resolves all the problems we saw in the "Antipattern" section.

Querying Products by Account and the Other Way Around

To query the attributes of all products for a given account, it's more straightforward to join the Products table with the Contacts table:

Download Jaywalking/soln/join.sql

```
SELECT p.*
FROM  Products AS p JOIN Contacts AS c ON (p.account_id = c.account_id)
WHERE c.account_id = 34;
```

Some people resist queries that contain a join, thinking that they perform poorly. However, this query uses indexes much better than the solution shown earlier in the "Antipattern" section.

Querying account details is likewise easy to read and easy to optimize. It uses indexes for the join efficiently, instead of an esoteric use of regular expressions:

Download Jaywalking/soln/join.sql

```
SELECT a.*
FROM   Accounts AS a JOIN Contacts AS c ON (a.account_id = c.account_id)
WHERE c.product_id = 123;
```

Making Aggregate Queries

The following example returns the number of accounts per product:

Download Jaywalking/soln/group.sql

```
SELECT product_id, COUNT(*) AS accounts_per_product
FROM Contacts
GROUP BY product_id;
```

The number of products per account is just as simple:

Download Jaywalking/soln/group.sql

```sql
SELECT account_id, COUNT(*) AS products_per_account
FROM Contacts
GROUP BY account_id;
```

Other more sophisticated reports are possible too, such as the product with the greatest number of accounts:

Download Jaywalking/soln/group.sql

```sql
SELECT c.product_id, c.accounts_per_product
FROM (
  SELECT product_id, COUNT(*) AS accounts_per_product
  FROM Contacts
  GROUP BY product_id
) AS c
HAVING c.accounts_per_product = MAX(c.accounts_per_product)
```

Updating Contacts for a Specific Product

You can add or remove entries in the list by inserting or deleting rows in the intersection table. Each product reference is stored in a separate row in the Contacts table, so you can add or remove them one at a time.

Download Jaywalking/soln/remove.sql

```sql
INSERT INTO Contacts (product_id, account_id) VALUES (456, 34);

DELETE FROM Contacts WHERE product_id = 456 AND account_id = 34;
```

Validating Product IDs

You can use a foreign key to validate the entries against a set of legitimate values in another table. You declare that Contacts.account_id references Accounts.account_id, and therefore you rely on the database to enforce referential integrity. Now you can be sure that the intersection table contains only account IDs that exist.

You can also use SQL data types to restrict entries. For example, if the entries in the list should be valid INTEGER or DATE values and you declare the column using those data types, you can be sure all entries are legal values of that type (not nonsense entries like *banana*).

Choosing a Separator Character

You use no separator character, since you store each entry on a separate row. There's no ambiguity if the entries contain commas or other characters you might have used as a separator.

List Length Limitations

Since each entry is in a separate row in the intersection table, the list is limited only by the number of rows that can physically exist in one table. If it's appropriate to limit the number of entries, you should enforce the policy in your application using the count of entries rather than the collective length of the list.

Other Advantages of the Intersection Table

An index on Contacts.account_id makes performance better than matching a substring in a comma-separated list. Declaring a foreign key on a column implicitly creates an index on that column in many database brands (but check your documentation).

You can also create additional attributes for each entry by adding columns to the intersection table. For example, you could record the date a contact was added for a given product or an attribute noting who is the primary contact vs. the secondary contacts. You can't do this in a comma-separated list.

_____

Store each value in its own column and row.

_____

A tree is a tree—how many more do you need to look at?
   ► Ronald Reagan

<div align="right">Chapter 3</div>

Naive Trees

Suppose you work as a software developer for a famous website for science and technology news.

This is a modern website, so readers can contribute comments and even reply to each other, forming threads of discussion that branch and extend deeply. You choose a simple solution to track these reply chains: each comment references the comment to which it replies.

Download Trees/intro/parent.sql

```sql
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  parent_id    BIGINT UNSIGNED,
  comment      TEXT NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Comments(comment_id)
);
```

It soon becomes clear, however, that it's hard to retrieve a long chain of replies in a single SQL query. You can get only the immediate children or perhaps join with the grandchildren, to a fixed depth. But the threads can have an *unlimited* depth. You would need to run many SQL queries to get all the comments in a given thread.

The other idea you have is to retrieve *all* the comments and assemble them into tree data structures in application memory, using traditional tree algorithms you learned in school. But the publishers of the website have told you that they publish dozens of articles every day, and each article can have hundreds of comments. Sorting through millions of comments every time someone views the website is impractical.

There must be a better way to store the threads of comments so you can retrieve a whole discussion thread simply and efficiently.

3.1 Objective: Store and Query Hierarchies

It's common for data to have recursive relationships. Data may be orga-
nized in a treelike or hierarchical way. In a tree data structure, each
entry is called a *node*. A node may have a number of children and one
parent. The top node, which has no parent, is called the *root*. The nodes
at the bottom, which have no children, are called *leaves*. The nodes in
the middle are simply *nonleaf nodes*.

In the previous hierarchical data, you may need to query individual
items, related subsets of the collection, or the whole collection. Exam-
ples of tree-oriented data structures include the following:

Organization chart: The relationship of employees to managers is the
    textbook example of tree-structured data. It appears in count-
    less books and articles on SQL. In an organizational chart, each
    employee has a manager, who represents the employee's *parent* in
    a tree structure. The manager is also an employee.

Threaded discussion: As seen in the introduction, a tree structure may
    be used for the chain of comments in reply to other comments. In
    the tree, the children of a comment node are its replies.

In this chapter, we'll use the threaded discussion example to show the
antipattern and its solutions.

3.2 Antipattern: Always Depend on One's Parent

The naive solution commonly shown in books and articles is to add
a column parent_id. This column references another comment in the
same table, and you can create a foreign key constraint to enforce this
relationship. The SQL to define this table is shown next, and the entity-
relationship diagram is shown in Figure 3.1, on the next page.

<div style="background:#eee;padding:4px;display:inline-block">Download Trees/anti/adjacency-list.sql</div>

```sql
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  parent_id    BIGINT UNSIGNED,
  bug_id       BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL,
  comment      TEXT NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Comments(comment_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);
```
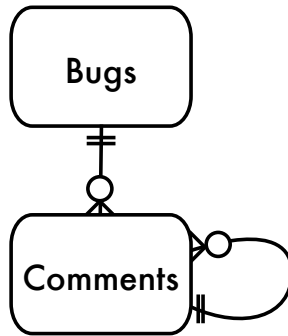
Figure 3.1: Adjacency list entity-relationship diagram

This design is called *Adjacency List.* It's probably the most common design software developers use to store hierarchical data. The following is some sample data to show a hierarchy of comments, and an illustration of the tree is shown in Figure 3.2, on the following page.

comment_id	parent_id	author	comment
1	NULL	Fran	What's the cause of this bug?
2	1	Ollie	I think it's a null pointer.
3	2	Fran	No, I checked for that.
4	1	Kukla	We need to check for invalid input.
5	4	Ollie	Yes, that's a bug.
6	4	Fran	Yes, please add a check.
7	6	Kukla	That fixed it.

Querying a Tree with Adjacency List

Adjacency List can be an antipattern when it's the default choice of so many developers yet it fails to be a solution for one of the most common tasks you need to do with a tree: query all descendants.

You can retrieve a comment and its immediate children using a relatively simple query:

Download **Trees/anti/parent.sql**

```
SELECT c1.*, c2.*
FROM Comments c1 LEFT OUTER JOIN Comments c2
  ON c2.parent_id = c1.comment_id;
```
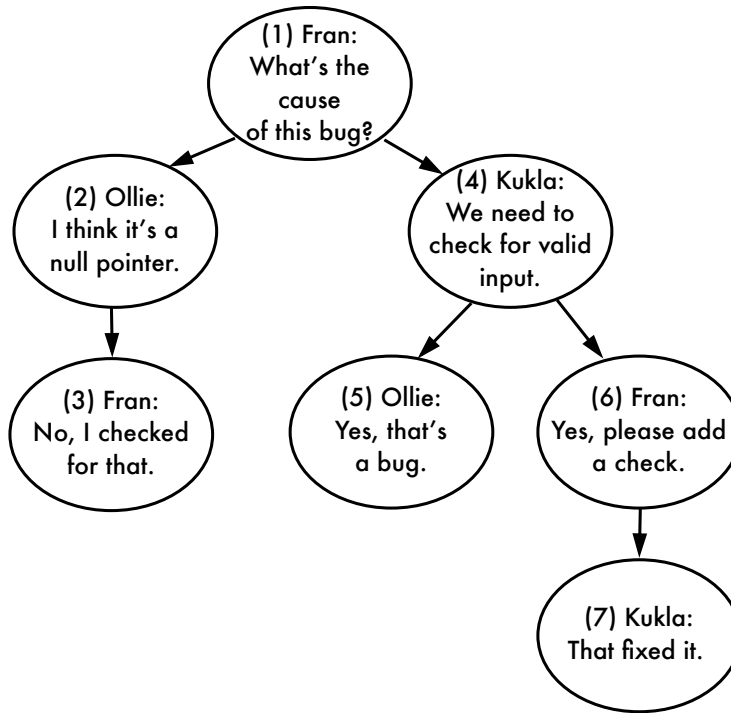
Figure 3.2: Threaded comments illustration

However, this queries only two levels of the tree. One characteristic of a tree is that it can extend to any depth, so you need to be able to query the descendents without regard to the number of levels. For example, you may need to compute the COUNT() of comments in the thread or the SUM() of the cost of parts in a mechanical assembly.

This kind of query is awkward when you use Adjacency List, because each level of the tree corresponds to another join, and the number of joins in an SQL query must be fixed. The following query retrieves a tree of depth up to four but cannot retrieve the tree beyond that depth:

Download **Trees/anti/ancestors.sql**

```
SELECT c1.*, c2.*, c3.*, c4.*
FROM Comments c1                        -- 1st level
  LEFT OUTER JOIN Comments c2
    ON c2.parent_id = c1.comment_id   -- 2nd level
```

```
  LEFT OUTER JOIN Comments c3
    ON c3.parent_id = c2.comment_id  -- 3rd level
  LEFT OUTER JOIN Comments c4
    ON c4.parent_id = c3.comment_id; -- 4th level
```

This query is also awkward because it includes descendants from progressively deeper levels by adding more columns. This makes it hard to compute an aggregate such as COUNT().

Another way to query a tree structure from Adjacency List is to retrieve all the rows in the collection and instead reconstruct the hierarchy in the application before you can use it like a tree.

Download **Trees/anti/all-comments.sql**

```
SELECT * FROM Comments WHERE bug_id = 1234;
```

Copying a large volume of data from the database to the application before you can analyze it is grossly inefficient. You might need only a subtree, not the whole tree from its top. You might require only aggregate information about the data, such as the COUNT() of comments.

Maintaining a Tree with Adjacency List

Admittedly, some operations are simple to accomplish with Adjacency List, such as adding a new leaf node:

Download **Trees/anti/insert.sql**

```
INSERT INTO Comments (bug_id, parent_id, author, comment)
  VALUES (1234, 7, 'Kukla', 'Thanks!');
```

Relocating a single node or a subtree is also easy:

Download **Trees/anti/update.sql**

```
UPDATE Comments SET parent_id = 3 WHERE comment_id = 6;
```

However, deleting a node from a tree is more complex. If you want to delete an entire subtree, you have to issue multiple queries to find all descendants. Then remove the descendants from the lowest level up to satisfy the foreign key integrity.

Download **Trees/anti/delete-subtree.sql**

```
SELECT comment_id FROM Comments WHERE parent_id = 4; -- returns 5 and 6
SELECT comment_id FROM Comments WHERE parent_id = 5; -- returns none
SELECT comment_id FROM Comments WHERE parent_id = 6; -- returns 7
SELECT comment_id FROM Comments WHERE parent_id = 7; -- returns none

DELETE FROM Comments WHERE comment_id IN ( 7 );
DELETE FROM Comments WHERE comment_id IN ( 5, 6 );
DELETE FROM Comments WHERE comment_id = 4;
```

You can use a foreign key with the ON DELETE CASCADE modifier to automate this, as long as you know you always want to delete the descendants instead of promoting or relocating them.

If you instead want to delete a nonleaf node and promote its children or move them to another place in the tree, you first need to change the parent_id of children and then delete the desired node.

Download Trees/anti/delete-non-leaf.sql

```sql
SELECT parent_id FROM Comments WHERE comment_id = 6; -- returns 4

UPDATE Comments SET parent_id = 4 WHERE parent_id = 6;

DELETE FROM Comments WHERE comment_id = 6;
```

These are examples of operations that require multiple steps when you use the Adjacency List design. That's a lot of code you have to write for tasks that a database should make simpler and more efficient.

3.3 How to Recognize the Antipattern

If you hear a question like the following, it's a clue that the Naive Trees antipattern is being employed:

- "How many levels do we need to support in trees?"

  You're struggling to get all descendants or all ancestors of a node, without using a recursive query. You could compromise by supporting only trees of a limited depth, but the next natural question is, how deep is deep enough?

- "I dread ever having to touch the code that manages the tree data structures."

  You've adopted one of the more sophisticated solutions of managing hierarchies, but you're using the wrong one. Each technique makes some tasks easier, but usually at the cost of other tasks that become harder. You may have chosen a solution that isn't the best choice for the way you need to use hierarchies in your application.

- "I need to run a script periodically to clean up the orphaned rows in the trees."

  Your application creates disconnected nodes in the tree as it deletes nonleaf nodes. When you store complex data structures in

a database, you need to keep the structure in a consistent, valid state after any change. You can use one of the solutions presented later in this chapter, along with triggers and cascading foreign key constraints, to store data structures that are resilient instead of fragile.

3.4 Legitimate Uses of the Antipattern

The Adjacency List design might be just fine to support the work you need to do in your application. The strength of the Adjacency List design is retrieving the direct parent or child of a given node. It's also easy to insert rows. If those operations are all you need to do with your hierarchical data, then Adjacency List can work well for you.

Don't Over-Engineer

I wrote an inventory-tracking application for a computer data center. Some equipment was installed inside computers; for example, a caching disk controller was installed in a rackmount server, and extra memory modules were installed on the disk controller.

I needed an SQL solution to track the usage of hierarchical collections easily. But I also needed to track each individual piece of equipment to produce accounting reports of equipment utilization, amortization, and return on investment.

The manager said the collections could have subcollections, and thus the tree could in theory descend to any depth. It took quite a few weeks to perfect the code for manipulating trees in the database storage, user interface, administration, and reporting.

In practice, however, the inventory application never needed to create a grouping of equipment with a tree deeper than a single parent-child relationship. If my client had acknowledged that this would be enough to model his inventory requirements, we could have saved a lot of work.

Some brands of RDBMS support extensions to SQL to support hierarchies stored in the Adjacency List format. The SQL-99 standard defines recursive query syntax using the WITH keyword followed by a *common table expression*.

Download **Trees/legit/cte.sql**

```
WITH CommentTree
    (comment_id, bug_id, parent_id, author, comment, depth)
AS (
    SELECT *, 0 AS depth FROM Comments
    WHERE parent_id IS NULL
```

```
    UNION ALL
      SELECT c.*, ct.depth+1 AS depth FROM CommentTree ct
      JOIN Comments c ON (ct.comment_id = c.parent_id)
)
SELECT * FROM CommentTree WHERE bug_id = 1234;
```

Microsoft SQL Server 2005, Oracle 11g, IBM DB2, and PostgreSQL 8.4 support recursive queries using common table expressions, as shown earlier.

MySQL, SQLite, and Informix are examples of database brands that don't support this syntax yet. It's the same for Oracle 10g, which is still widely used. In the future, we might assume recursive query syntax will become available across all popular brands, and then using Adjacency List won't be so limiting.

Oracle 9i and 10g support the WITH clause, but not for recursive queries. Instead, there is proprietary syntax: START WITH and CONNECT BY PRIOR. You can use this syntax to perform recursive queries:

Download **Trees/legit/connect-by.sql**

```
SELECT * FROM Comments
START WITH comment_id = 9876
CONNECT BY PRIOR parent_id = comment_id;
```

3.5   Solution: Use Alternative Tree Models

There are several alternatives to the Adjacency List model of storing hierarchical data, including *Path Enumeration*, *Nested Sets*, and *Closure Table*. The following three sections show examples using these designs to solve the scenario in the "Antipattern" section, storing and querying a tree-like collection of comments.

These solutions take some getting used to. They may seem more complex than Adjacency List at first, but they make some tree operations easier that were very difficult or inefficient using the Adjacency List design. If your application needs to perform those operations, then these designs are a better choice than the simple Adjacency List.

Path Enumeration

One weakness of Adjacency List is that it's expensive to retrieve ancestors of a given node in the tree. In Path Enumeration, this is solved by storing the string of ancestors as an attribute of each node.

You can see a form of Path Enumeration in directory hierarchies. A UNIX path like /usr/local/lib/ is a Path Enumeration of the filesystem, where usr is the parent of local, which in turn is the parent of lib.

In the Comments table, instead of the parent_id column, define a column called path as a long VARCHAR. The string stored in this column is the sequence of ancestors of the current row in order from the top of the tree down, just like a UNIX path. You can even choose / as a separator character.

Download Trees/soln/path-enum/create-table.sql

```sql
CREATE TABLE Comments (
  comment_id    SERIAL PRIMARY KEY,
  path          VARCHAR(1000),
  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)
);
```

comment_id	path	author	comment
1	1/	Fran	What's the cause of this bug?
2	1/2/	Ollie	I think it's a null pointer.
3	1/2/3/	Fran	No, I checked for that.
4	1/4/	Kukla	We need to check for invalid input.
5	1/4/5/	Ollie	Yes, that's a bug.
6	1/4/6/	Fran	Yes, please add a check.
7	1/4/6/7/	Kukla	That fixed it.

You can query ancestors by comparing the current row's path to a pattern formed from the path of another row. For example, to find ancestors of comment #7, whose path is *1/4/6/7/*, do this:

Download Trees/soln/path-enum/ancestors.sql

```sql
SELECT *
FROM Comments AS c
WHERE '1/4/6/7/' LIKE c.path || '%';
```

This matches the patterns formed from paths of ancestors *1/4/6/%*, *1/4/%*, and *1/%*.

You can query descendants by reversing the arguments of the LIKE predicate. To find the descendants of comment #4 whose path is *1/4/*, use this:

Download **Trees/soln/path-enum/descendants.sql**

```sql
SELECT *
FROM Comments AS c
WHERE c.path LIKE '1/4/' || '%';
```

The pattern *1/4/%* matches the paths of descendants *1/4/5/*, and *1/4/6/*, and *1/4/6/7/*.

Once you can easily select a subset of the tree or the chain of ancestors to the top of the tree, you can perform many other queries easily, such as computing the SUM() of costs of nodes in a subtree or simply counting the number of nodes. For example, to count the comments per author in the subtree starting at comment #4, do this:

Download **Trees/soln/path-enum/count.sql**

```sql
SELECT COUNT(*)
FROM Comments AS c
WHERE c.path LIKE '1/4/' || '%'
GROUP BY c.author;
```

Inserting a node is similar to inserting in the Adjacency List model. You can insert a nonleaf node without needing to modify any other row. Copy the path from the new node's parent, and append the ID of the new node to this string. If your primary key generates its value automatically during the insert, you may need to insert the row and then update the path once you know the ID value for the new row. For example, if you use MySQL, the built-in function LAST_INSERT_ID() returns the most recent ID value generated for an inserted row in the current session. Get the rest of the path from the parent of your new node.

Download **Trees/soln/path-enum/insert.sql**

```sql
INSERT INTO Comments (author, comment) VALUES ('Ollie', 'Good job!');

UPDATE Comments
  SET path = (SELECT path FROM Comments WHERE comment_id = 7)
    || LAST_INSERT_ID() || '/'
WHERE comment_id = LAST_INSERT_ID();
```

Path Enumeration has some drawbacks similar to those shown in Chapter 2, *Jaywalking*, on page 25. The database can't enforce that the path is formed correctly or that values in the path correspond to exist-

ing nodes. Maintaining the path string depends on application code, and verifying it is costly. No matter how long you make theVARCHAR column, it still has a length limit, so it doesn't strictly support trees of unlimited depth.

Path Enumeration allows you to sort a set of rows easily by their hierarchy, as long as the elements between the separator are of consistent length.[1]

Nested Sets

The Nested Sets solution stores information with each node that pertains to the set of its descendants, rather than the node's immediate parent. This information can be represented by encoding each node in the tree with two numbers, which you can call nsleft and nsright.

Download Trees/soln/nested-sets/create-table.sql

```sql
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  nsleft       INTEGER NOT NULL,
  nsright      INTEGER NOT NULL,
  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)
);
```

Each node is given nsleft and nsright numbers in the following way: the nsleft number is less than the numbers of all the node's children, whereas the nsright number is greater than the numbers of all the node's children. These numbers have no relation to the comment_id values.

An easy way to assign these values is by following a depth-first traversal of the tree, assigning nsleft numbers incrementally as you descend a branch of the tree and assigning nsright numbers as you ascend back up the branch.

It may be easier to visualize the pattern from Figure 3.3, on the following page than from this description.

---

1. On the other hand, this may smell too much like the Jaywalking antipattern.
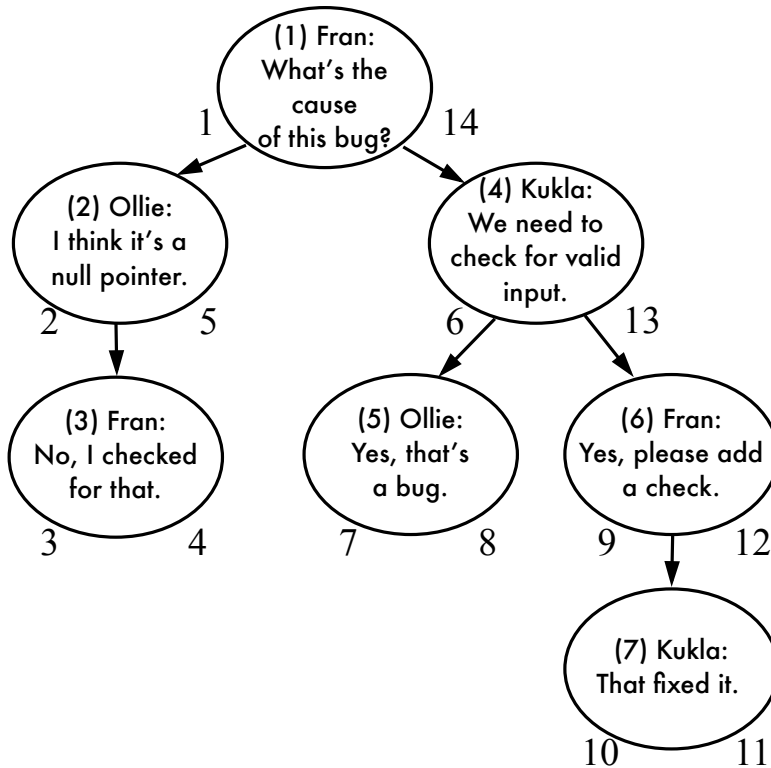
Figure 3.3: Nested Sets illustration

comment_id	nsleft	nsright	author	comment
1	1	14	Fran	What's the cause of this bug?
2	2	5	Ollie	I think it's a null pointer.
3	3	4	Fran	No, I checked for that.
4	6	13	Kukla	We need to check for invalid input.
5	7	8	Ollie	Yes, that's a bug.
6	9	12	Fran	Yes, please add a check.
7	10	11	Kukla	That fixed it.

Once you have assigned each node with these numbers, you can use them to find ancestors and descendants of any given node. For example, you can retrieve comment #4 and its descendants by searching for nodes whose numbers are between the current node's nsleft and nsright.

Download **Trees/soln/nested-sets/descendants.sql**

```sql
SELECT c2.*
FROM Comments AS c1
  JOIN Comments as c2
    ON c2.nsleft BETWEEN c1.nsleft AND c1.nsright
WHERE c1.comment_id = 4;
```

You can retrieve comment #6 and its ancestors by searching for nodes whose numbers span the current node's numbers. For example:

Download **Trees/soln/nested-sets/ancestors.sql**

```sql
SELECT c2.*
FROM Comments AS c1
  JOIN Comment AS c2
    ON c1.nsleft BETWEEN c2.nsleft AND c2.nsright
WHERE c1.comment_id = 6;
```

One chief strength of the Nested Sets design is that when you delete a nonleaf node, its descendants are automatically considered direct children of the deleted node's parents. Although the right and left numbers of each node shown in the illustration have values forming a continuous series and the difference is always one compared to adjacent siblings and parents, this is not necessary for the Nested Sets design to preserve the hierarchy. So when gaps in the values result from deleting a node, there is no interruption to the tree structure.

For example, you can count the depth of a given node and delete its parent, and then when you count the depth of the node again, it seems to have decreased depth by one level.

Download **Trees/soln/nested-sets/depth.sql**

```sql
-- Reports depth = 3
SELECT c1.comment_id, COUNT(c2.comment_id) AS depth
FROM Comment AS c1
  JOIN Comment AS c2
    ON c1.nsleft BETWEEN c2.nsleft AND c2.nsright
WHERE c1.comment_id = 7
GROUP BY c1.comment_id;

DELETE FROM Comment WHERE comment_id = 6;

-- Reports depth = 2
SELECT c1.comment_id, COUNT(c2.comment_id) AS depth
FROM Comment AS c1
  JOIN Comment AS c2
    ON c1.nsleft BETWEEN c2.nsleft AND c2.nsright
WHERE c1.comment_id = 7
GROUP BY c1.comment_id;
```

However, some queries that are simple in the Adjacency List design, such as retrieving the immediate child or immediate parent, are more complex in the Nested Sets design. The direct parent of a given node c1 is an ancestor of that node, but no other node can exist in between them. So, you can use an additional outer join to search for a node that is both an ancestor of c1 and a descendant of the parent. Only if no such node is found (that is, the result of the outer join is null) is the ancestor truly the direct parent of c1.

For example, to find the immediate parent of comment #6, do this:

Download Trees/soln/nested-sets/parent.sql

```
SELECT parent.*
FROM Comment AS c
  JOIN Comment AS parent
    ON c.nsleft BETWEEN parent.nsleft AND parent.nsright
  LEFT OUTER JOIN Comment AS in_between
    ON c.nsleft BETWEEN in_between.nsleft AND in_between.nsright
    AND in_between.nsleft BETWEEN parent.nsleft AND parent.nsright
WHERE c.comment_id = 6
  AND in_between.comment_id IS NULL;
```

Manipulations of the tree, inserting and moving nodes, are generally more complex in the Nested Sets design than they are in other models. When you insert a new node, you need to recalculate all the left and right values greater than the left value of the new node.

This includes the new node's right siblings, its ancestors, and the right siblings of its ancestors. It also includes descendants, if the new node is inserted as a nonleaf node. Assuming the new node is a leaf node, the following statement should update everything necessary:

Download Trees/soln/nested-sets/insert.sql

```
-- make space for NS values 8 and 9
UPDATE Comment
  SET nsleft = CASE WHEN nsleft >= 8 THEN nsleft+2 ELSE nsleft END,
      nsright = nsright+2
WHERE nsright >= 7;

-- create new child of comment #5, occupying NS values 8 and 9
INSERT INTO Comment (nsleft, nsright, author, comment)
  VALUES (8, 9, 'Fran', 'Me too!');
```

The Nested Sets model is best when it's more important to perform queries for subtrees quickly and easily, rather than operations on individual nodes. Inserting and moving nodes is complex, because of the

requirement to renumber the left and right values. If your usage of the tree involves frequent insertions, Nested Sets isn't the best choice.

Closure Table

The Closure Table solution is a simple and elegant way of storing hierarchies. It involves storing all paths through the tree, not just those with a direct parent-child relationship.

In addition to a plain Comments table, create another table TreePaths, with two columns, each of which is a foreign key to the Comments table.

Download **Trees/soln/closure-table/create-table.sql**

```sql
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 TreePaths (
  ancestor     BIGINT UNSIGNED NOT NULL,
  descendant   BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY(ancestor, descendant),
  FOREIGN KEY (ancestor) REFERENCES Comments(comment_id),
  FOREIGN KEY (descendant) REFERENCES Comments(comment_id)
);
```

Instead of using the Comments table to store information about the tree structure, use the TreePaths table. Store one row in this table for each pair of nodes in the tree that shares an ancestor/descendant relationship, even if they are separated by multiple levels in the tree. Also add a row for each node to reference itself. For an illustration of how the nodes are paired, see Figure 3.4, on the next page.

ancestor	descendant	ancestor	descendant	ancestor	descendant
1	1	1	7	4	6
1	2	2	2	4	7
1	3	2	3	5	5
1	4	3	3	6	6
1	5	4	4	6	7
1	6	4	5	7	7

Figure 3.4: Closure Table illustration
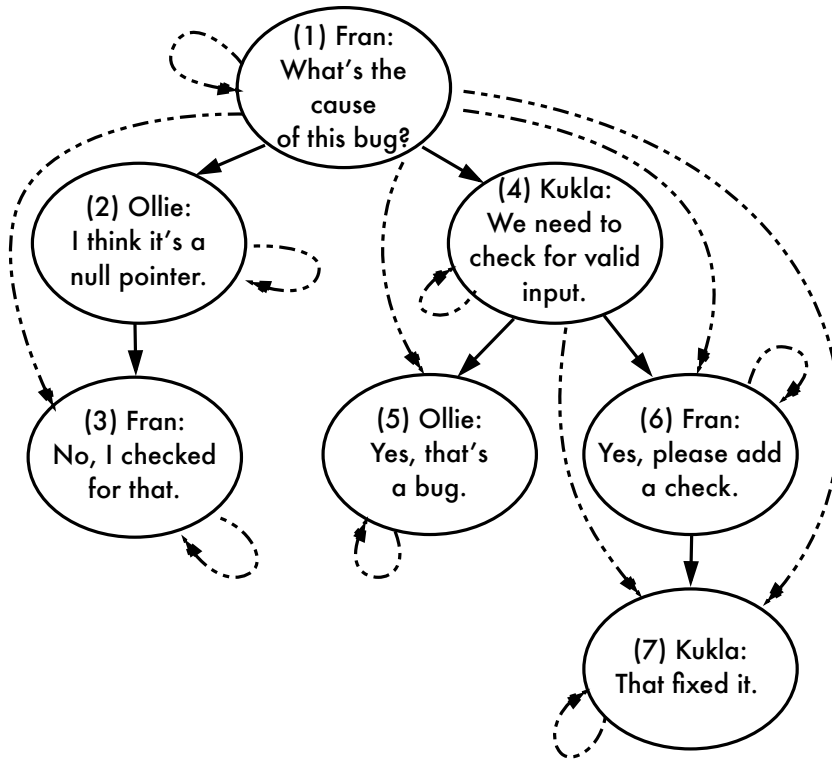
The queries to retrieve ancestors and descendants from this table are even more straightforward than those in the Nested Sets solution. To retrieve descendants of comment #4, match rows in TreePaths where the ancestor is 4:

Download **Trees/soln/closure-table/descendants.sql**

```
SELECT c.*
FROM Comments AS c
  JOIN TreePaths AS t ON c.comment_id = t.descendant
WHERE t.ancestor = 4;
```

To retrieve ancestors of comment #6, match rows in TreePaths where the descendant is 6:

Download Trees/soln/closure-table/ancestors.sql

```sql
SELECT c.*
FROM Comments AS c
  JOIN TreePaths AS t ON c.comment_id = t.ancestor
WHERE t.descendant = 6;
```

To insert a new leaf node, for instance a new child of comment #5, first insert the self-referencing row. Then add a copy of the set of rows in TreePaths that reference comment #5 as a descendant (including the row in which comment #5 references itself), replacing the descendant with the number of the new comment:

Download Trees/soln/closure-table/insert.sql

```sql
INSERT INTO TreePaths (ancestor, descendant)
  SELECT t.ancestor, 8
  FROM TreePaths AS t
  WHERE t.descendant = 5
 UNION ALL
  SELECT 8, 8;
```

To delete a leaf node, for instance comment #7, delete all rows in TreePaths that reference comment #7 as a descendant:

Download Trees/soln/closure-table/delete-leaf.sql

```sql
DELETE FROM TreePaths WHERE descendant = 7;
```

To delete a complete subtree, for instance comment #4 and its descendants, delete all rows in TreePaths that reference comment #4 as a descendant, as well as all rows that reference any of comment #4's descendants as descendants:

Download Trees/soln/closure-table/delete-subtree.sql

```sql
DELETE FROM TreePaths
WHERE descendant IN (SELECT descendant
                     FROM TreePaths
                     WHERE ancestor = 4);
```

Notice that if you delete rows in TreePaths, this doesn't delete the comments themselves. This seems odd for this example of Comments, but it makes more sense if you're working with other kinds of trees, for instance categories in a product catalog or employees in an org chart. You don't necessarily want to delete a node when you change its relationship to other nodes. When you store paths in a separate table, it helps make this more flexible.

To move a subtree from one location in the tree to another, first discon-
nect the subtree from its ancestors by deleting rows that reference the
ancestors of the top node in the subtree and the descendants of that
node. For instance, to move comment #6 from its position as a child of
comment #4 to a child of comment #3, start with the following deletion.
Make sure not to delete comment #6's self-reference.

Download Trees/soln/closure-table/move-subtree.sql

```sql
DELETE FROM TreePaths
WHERE descendant IN (SELECT descendant
                     FROM TreePaths
                     WHERE ancestor = 6)
  AND ancestor IN (SELECT ancestor
                   FROM TreePaths
                   WHERE descendant = 6
                     AND ancestor != descendant);
```

By selecting ancestors of #6, but not #6 itself, and descendants of #6,
including #6, this correctly removes all the paths from #6's ancestors
to #6 and its descendants. In other words, this deletes the paths (1, 6),
(1,7), (4, 6), and (4, 7). It does not delete (6, 6) or (6, 7).

Then add the orphaned subtree by inserting rows matching the ances-
tors of the new location and the descendants of the subtree. You can
use the CROSS JOIN syntax to create a Cartesian product, generating the
rows needed to match ancestors of the new location to all the nodes in
the subtree you need to move.

Download Trees/soln/closure-table/move-subtree.sql

```sql
INSERT INTO TreePaths (ancestor, descendant)
  SELECT supertree.ancestor, subtree.descendant
  FROM TreePaths AS supertree
    CROSS JOIN TreePaths AS subtree
  WHERE supertree.descendant = 3
    AND subtree.ancestor = 6;
```

This creates new paths using the ancestors of #3, including #3, and
the descendants of #6, including #6. So, the new paths are (1, 6), (2,
6), (3, 6), (1, 7), (2, 7), (3, 7). The result is that the subtree starting
with comment #6 is relocated as a child of comment #3. The cross join
creates all the needed paths, even if the subtree is moved to a higher or
lower level in the tree.

The Closure Table design is more straightforward than the Nested Sets
design. Both have quick and easy methods for querying ancestors and
descendants, but the Closure Table is easier to maintain the hierarchy

Design	Tables	Query Child	Query Tree	Insert	Delete	Ref. Integ.
Adjacency List	1	Easy	Hard	Easy	Easy	Yes
Recursive Query	1	Easy	Easy	Easy	Easy	Yes
Path Enumeration	1	Easy	Easy	Easy	Easy	No
Nested Sets	1	Hard	Easy	Hard	Hard	No
Closure Table	2	Easy	Easy	Easy	Easy	Yes

Figure 3.5: Comparing hierarchical data designs

information. In both designs, it's more convenient to query immediate child or parent nodes than in the Adjacency List or Path Enumeration designs.

However, you can improve the Closure Table to make queries for immediate parent or child nodes easier. Add a TreePaths.path_length attribute to the Closure Table design. The path_length of a node's self-reference is zero, the path_length of its immediate child is 1, the path_length of its grandchild is 2, and so on. Finding the children of comment #4 is now straightforward:

Download Trees/soln/closure-table/child.sql

```sql
SELECT *
FROM TreePaths
WHERE ancestor = 4 AND path_length = 1;
```

Which Design Should You Use?

Each of the designs has its own strengths and weaknesses. Choose the design depending on which operations you need to be most efficient. In Figure 3.5, some operations are marked as easy or hard with each respective tree design. You can also consider the following strengths and weaknesses of each design:

- *Adjacency List* is the most conventional design, and many software developers recognize it.

- *Recursive Queries* using WITH or CONNECT BY PRIOR make it more efficient to use the Adjacency List design, provided you use one of the database brands that supports the syntax.

- *Path Enumeration* is good for breadcrumbs in user interfaces, but it's fragile because it fails to enforce referential integrity and stores information redundantly.

- *Nested Sets* is a clever solution—maybe too clever. It also fails to support referential integrity. It's best used when you need to query a tree more frequently than you need to modify the tree.

- *Closure Table* is the most versatile design and the only design in this chapter that could allow a node to belong to multiple trees. It requires an additional table to store the relationships. This design also uses a lot of rows when encoding deep hierarchies, increasing space consumption as a trade-off for reducing computing.

There's more to learn about storing and manipulating hierarchical data in SQL. A good book that covers hierarchical queries is *Joe Celko's Trees and Hierarchies in SQL for Smarties* [Cel04]. Another book that covers trees and even graphs is *SQL Design Patterns* [Tro06] by Vadim Tropashko. The latter book has a more formal, academic style.

A hierarchy consists of entries and relationships.
    Model both of these to suit your work.

Chapter 4

ID Required

Recently I answered a question that I see frequently, from a software developer trying to prevent duplicate rows. At first I thought he must lack a primary key. But that wasn't his problem.

In his content management database, he stored articles for publishing on a website. He used an intersection table for a many-to-many association between a table of articles and a table of tags.

Download ID-Required/intro/articletags.sql

```sql
CREATE TABLE ArticleTags (
  id          SERIAL PRIMARY KEY,
  article_id  BIGINT UNSIGNED NOT NULL,
  tag_id      BIGINT UNSIGNED NOT NULL,
  FOREIGN KEY (article_id) REFERENCES Articles (id),
  FOREIGN KEY (tag_id)     REFERENCES Tags (id)
);
```

He was getting incorrect results from queries when counting the number of articles with a given tag. He knew that there were only five articles with the "economy" tag, but the query was telling him there were seven.

Download ID-Required/intro/articletags.sql

```sql
SELECT tag_id, COUNT(*) AS articles_per_tag FROM ArticleTags WHERE tag_id = 327;
```

When he queried all the rows matching that tag_id, he saw that the tag was associated with one particular article in triplicate; three rows showed the same association, although they had different values for id.

id	tag_id	article_id
22	327	1234
23	327	1234
24	327	1234

This table had a primary key, but that primary key wasn't preventing duplicates in the columns that mattered. One remedy might be to create a UNIQUE constraint over the other two columns, but given that, why is the id column needed at all?

4.1 Objective: Establish Primary Key Conventions

The objective is to make sure every table has a primary key, but confusion about the nature of a primary key has resulted in an antipattern.

Everyone who has been introduced to database design knows that a primary key is an important, even mandatory, part of a table. This is true; primary keys are integral to good database design. A primary key is guaranteed to be unique over all rows in the table, so this is the logical mechanism to address individual rows and to prevent duplicate rows from being stored. A primary key is also referenced by foreign keys to create table associations.

The tricky part is choosing a column to serve as the primary key. The value of any attribute in most tables has the potential to belong on more than one row. Textbook examples such as a person's first name and last name are clearly subject to having duplication. Even an email address or administrative identification numbers such as a United States Social Security number or taxpayer ID number aren't strictly unique.

A new column is needed in such tables to store an artificial value that has no meaning in the domain modeled by the table. This column is used as the primary key, so you can address rows uniquely while allowing any other attribute column to contain duplicates, if that's appropriate. This type of primary key column is sometimes called a *pseudokey* or a *surrogate key*.

To ensure rows can be given unique pseudokey values even when concurrent clients are inserting new rows, most databases provide a mechanism to generate unique integer values serially, outside the scope of transaction isolation.

Do I Really Need a Primary Key?

I've heard some software developers claim that their table doesn't need a primary key.

Sometimes these programmers want to avoid the imagined overhead of maintaining a unique index, or else they have tables with no columns they can use for this purpose.

A primary key constraint is important when you need to do the following:

- Prevent a table from containing duplicate rows
- Reference individual rows in queries
- Support foreign key references

If you don't use primary key constraints, you create a chore for yourself: checking for duplicate rows.

```
SELECT bug_id FROM Bugs GROUP BY bug_id HAVING COUNT(*) > 1;
```

How frequently should you run this check? What should you do with a duplicate when you find one?

A table without a primary key is like organizing your MP3 collection with no song titles. You can still listen to the music, but you can't find the one you want or keep duplicates out of your collection.

Pseudokeys weren't standardized until SQL:2003, so each database uses its own extension to SQL to implement them. Even the terminology for pseudokeys is vendor-dependent, as shown by the following table:

Feature	Supported by Database Brands
AUTO_INCREMENT	MySQL
GENERATOR	Firebird, InterBase
IDENTITY	DB2, Derby, Microsoft SQL Server, Sybase
ROWID	SQLite
SEQUENCE	DB2, Firebird, Informix, Ingres, Oracle, PostgreSQL
SERIAL	MySQL, PostgreSQL

Pseudokeys are a useful feature, but they aren't the only solution for declaring a primary key.

4.2 Antipattern: One Size Fits All

Books, articles, and programming frameworks have established a cultural convention that every database table must have a primary key column with the following characteristics:

- The primary key's column name is id.

- Its data type is a 32-bit or 64-bit integer.

- Unique values are generated automatically.

The presence of a column named id in every table is so common that this has become synonymous with a primary key. Programmers learning SQL get the false idea that a primary key always means a column defined in this manner.

Download ID-Required/anti/id-ubiquitous.sql

```sql
CREATE TABLE Bugs (
  id          SERIAL PRIMARY KEY,
  description VARCHAR(1000),
  -- . . .
);
```

Adding an id column to every table causes several effects that make its use seem arbitrary.

Making a Redundant Key

You might see an id column defined as the primary key simply for the sake of tradition, even when another column in the same table could be used as the natural primary key. The other column may even be defined with a UNIQUE constraint. For example, in the Bugs table, the application might label bugs using a string with a mnemonic for the project the bug belongs to, or other identifying information.

Download ID-Required/anti/id-redundant.sql

```sql
CREATE TABLE Bugs (
  id          SERIAL PRIMARY KEY,
  bug_id      VARCHAR(10) UNIQUE,
  description VARCHAR(1000),
  -- . . .
);

INSERT INTO Bugs (bug_id, description, ...)
  VALUES ('VIS-078', 'crashes on save', ...);
```

The bug_id column in the previous example has similar usage to the id, in that it serves to identify each row uniquely.

Allowing Duplicate Rows

A compound key consists of multiple columns. One typical use for a compound key is in an intersection table like BugsProducts. The primary key should ensure that a given combination of values for bug_id and product_id appears only once in the table, even though each value may appear many times in different pairings.

However, when you use the mandatory id column as the primary key, the constraint no longer applies to two columns that should be unique.

Download ID-Required/anti/superfluous.sql

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

INSERT INTO BugsProducts (bug_id, product_id)
  VALUES (1234, 1), (1234, 1), (1234, 1); -- duplicates are permitted
```

Duplicates in this intersection table cause unintended results when you use the table to match Bugs to Products. To prevent duplicates, you could declare a UNIQUE constraint over the two columns besides id:

Download ID-Required/anti/superfluous.sql

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

But if you need a unique constraint over those two columns anyway, the id column is superfluous.

Obscuring the Meaning of the Key

The word *code* has a number of definitions, one of which is a way to communicate a message with brevity or secrecy. In programming, we should have the opposite goal—to make meaning clearer.

The name id is so generic that it holds no meaning. This is especially important when you join two tables and they have the same primary key column name.

```
SELECT b.id, a.id
FROM Bugs b
JOIN Accounts a ON (b.assigned_to = a.id)
WHERE b.status = 'OPEN';
```

How do you tell the bug id from the account id in your application code if you reference columns by name instead of by ordinal position? This is a problem especially in dynamic languages like PHP, when a query result is an associative array: one column overwrites the other unless you specify column aliases in your query.

The name of the id column doesn't help make the query any clearer. But if the columns were named bug_id and account_id, the reader would have a much easier time reading the query results. We use a primary key to address individual rows of a table, so the column's name should give a clue about the type of entity in that table.

Using USING

You're probably familiar with the SQL syntax for a join, using the key-words JOIN and ON preceding an expression to evaluate matching rows in the two tables.

```
SELECT * FROM Bugs AS b JOIN BugsProducts AS bp ON (b.bug_id = bp.bug_id);
```

SQL also supports a more concise syntax for expressing a join between two tables. You can rewrite the previous query in the following way if the columns have the same name in both tables:

```
SELECT * FROM Bugs JOIN BugsProducts USING (bug_id);
```

However, if all tables are required to define a pseudokey primary key named id, then a foreign key column in a dependent table can never use the same name as the primary key it references. Instead, you must always use the more verbose ON syntax:

```
SELECT * FROM Bugs AS b JOIN BugsProducts AS bp ON (b.id = bp.bug_id);
```

Compound Keys Are Hard

Some developers refuse to use compound keys because they say these keys are too hard to use. Any expression that compares a key to another must compare all columns. A foreign key that references a compound

Special Scope for Sequences

Some people allocate a value for a new row by taking the greatest value currently in use and adding one.

```
SELECT MAX(bug_id) + 1 AS next_bug_id FROM Bugs;
```

This isn't reliable when you have concurrent clients both querying for the next value to use. The same value could be used by both clients. This is called a *race condition*.

To avoid the race condition, you have to block concurrent inserts while you read the current maximum value and then use it in a new row. To do this, you have to lock the whole table—row-level locking isn't enough. Table locks create a bottleneck because they cause concurrent clients to queue up for access.

Sequences solve this by operating outside of transaction scope. They never allocate the same value to multiple clients and therefore never roll back allocation of a value, whether or not you commit that value in a row. Because sequences work this way, multiple clients can generate unique values concurrently and be assured they won't try to use the same value.

Most databases support some function to return the last value a sequence generated. For example, MySQL calls this function LAST_INSERT_ID( ), Microsoft SQL Server uses SCOPE_IDENTITY( ), and Oracle uses *SequenceName*.CURRVAL( ).

These functions return the value generated during the current session, even if other clients generate their own values concurrently. No race condition exists.

primary key must itself be a compound foreign key. It requires more typing to use compound keys.

This refusal is like a mathematician refusing to use two-dimensional or three-dimensional coordinates, instead performing all calculations as though objects exist within a one-dimensional, linear space. It's true that this would make a lot of geometry and trigonometry much simpler, but it fails to describe real-world objects that we need to work with.

4.3 How to Recognize the Antipattern

The symptom of this antipattern is easy to recognize: tables use the overly generic name id for the primary key. There's virtually no reason to prefer this column name over one that is more descriptive.

The following can also be evidence of the antipattern:

- "I don't think I need a primary key in this table."

  The developer who says this is confusing the term *primary key* with *pseudokey*. Every table must have a primary key *constraint* to prevent duplicate rows and identify individual rows. They might want to use a natural key or a compound key instead.

- "How did I get duplicate many-to-many associations?"

  An intersection table for a many-to-many relationship should declare a primary key constraint, or at least a *unique key* constraint, over the set of foreign key columns.

- "I read that database theory says I should move values to a lookup table and refer to them by ID. But I don't want to do that because I have to do a join every time I want the actual values."

  This is a common misunderstanding of database design theory called *normalization*, which has nothing to do with pseudokeys in reality. For more on this, see Appendix A, on page 294.

4.4 Legitimate Uses of the Antipattern

Some object-relational frameworks simplify development by assuming *convention over configuration*. They expect every table to define its primary key in the same way: as an integer pseudokey column named id. If you use such a framework, you may want to conform to its conventions, because this gives you access to other desirable features of the framework.

There's also nothing wrong with using a pseudokey, or assigning values from an auto-incrementing integer mechanism. But not every table needs a pseudokey, and it's not necessary to name every pseudokey id.

A pseudokey is a good choice as a surrogate for a natural key that's too long to be practical. For example, for a table that records attributes of a file on the filesystem, the path of the file might be a good natural key, but it would be costly to index a string column that long.

4.5 Solution: Tailored to Fit

A primary key is a constraint, not a data type. You can declare a primary key on any column or set of columns, as long as the data types support indexing. You should also be able to define a column as an auto-incrementing integer without making it the primary key of the table. The two concepts are independent.

Don't let inflexible conventions get in the way of good design.

Tell It Like It Is

Choose sensible names for your primary key. The name should convey the type of entity that the primary key identifies. For example, the primary key of the Bugs table should be bug_id.

Use the same column name in foreign keys where possible. This often means that the name of a primary key should be unique within your schema; no two tables should use the same name for their primary key, unless one is also a foreign key referencing the other. However, there are exceptions: sometimes it is appropriate for a foreign key to be named differently from the primary key it references, for instance to be descriptive of the nature of the association.

Download ID-Required/soln/foreignkey-name.sql

```
CREATE TABLE Bugs (
  -- . . .
  reported_by  BIGINT UNSIGNED NOT NULL,
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id)
);
```

An industry standard exists to describe naming conventions for metadata. The standard, called ISO/IEC 11179,[1] is a guideline for "managing classification schemes" in information technology systems. In other words, this is how you should name your tables and columns sensibly. Like most ISO standards, this document is nearly impenetrable, but Joe Celko applies it practically to SQL in his book *SQL Programming Style* [Cel05].

---

1. http://metadata-standards.org/11179/

Be Unconventional

Object-relational frameworks expect you to use a pseudokey named id, but they also allow you to override this and declare a different name instead. The following example uses Ruby on Rails:[2]

Download ID-Required/soln/custom-primarykey.rb

```ruby
class Bug < ActiveRecord::Base
  set_primary_key "bug_id"
end
```

Some developers think that specifying the primary key column is necessary only when supporting *legacy* databases where they can't use their preferred conventions. In fact, supporting sensible column names is also important in new projects.

Embrace Natural Keys and Compound Keys

If your table contains an attribute that's guaranteed to be unique, is non-null, and can serve to identify the row, don't feel obligated to add a pseudokey solely for the sake of tradition.

Practically speaking, it's not uncommon for every attribute in a table to be subject to change or to be nonunique. Databases tend to evolve during the lifetime of a project, and decision makers may not respect the sanctity of a natural key. Sometimes a column that at first seemed like it would be a good natural key turns out to have legitimate duplicates. In those cases, a pseudokey is the only solution.

Use compound keys when they're appropriate. When a row is best identified by the combination of multiple attribute columns, as in the BugsProducts table, use those columns in a compound primary key.

Download ID-Required/soln/compound.sql

```sql
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)
);

INSERT INTO BugsProducts (bug_id, product_id)
  VALUES (1234, 1), (1234, 2), (1234, 3);
```

---

2. "Rails" and "Ruby on Rails" are trademarks of David Heinemeier Hansson.

```
INSERT INTO BugsProducts (bug_id, product_id)
  VALUES (1234, 1); -- error: duplicate entry
```

Note that foreign keys that reference a compound primary key also need to be compound. This may seem clumsy to duplicate these columns in dependent tables, but they can have advantages too: you might simplify a query that would have required a join to fetch attributes of the referenced row.

Conventions are good only if they are helpful.

Victorious warriors win first and then go to war, while
defeated warriors go to war first and then seek to win.
► Sun Tzu

Chapter 5

Keyless Entry

"Bill, it looks like two managers have reserved the same server in our lab for the same days—how can this happen?" the testing lab manager burst into my cube. "Can you take a look into this and get it fixed? They're screaming at me that they both need the equipment and that I'm holding up their project schedule."

I designed an equipment-tracking application some years ago using MySQL. The default storage engine for MySQL was MyISAM, which doesn't support foreign key constraints. The database had many logical relationships but could not enforce referential integrity.

As the project evolved and the application manipulated data in new ways, we developed a problem: when referential integrity wasn't satisfied, discrepancies showed up in reports, subtotals didn't add up, and schedules became double-booked.

The project manager asked me to write quality control scripts that we could run periodically to let us know when discrepancies occurred. These scripts examined the state of the database, found mistakes such as orphaned rows in child tables, and sent an email to report them.

Every table relationship had to be checked by these scripts. As the volume of data grew larger and the number of tables increased, the number of quality control queries also grew, and the scripts took longer to run. The email reports became longer too. Sound familiar?

The script solution worked, of course, but it was a costly reinvention of the wheel. What I needed was a way to make the application *fail early* whenever a user submitted invalid data. Guess what foreign key constraints do?

5.1 Objective: Simplify Database Architecture

Relational database design is almost as much about relationships between tables as it is about the individual tables themselves. *Referential integrity* is an important part of proper database design and operation. When you declare a foreign key constraint for a column or set of columns, the values in these columns must exist in the primary key or unique key columns of the parent table. This seems simple enough.

However, some software developers recommend avoiding referential integrity constraints. The reasons you might hear to ignore foreign keys include the following:

- Your data updates can conflict with the constraints.
- You're using a database design that's so flexible it can't support referential integrity constraints.
- You believe that the index the database creates for the foreign key will impact performance.
- You use a database brand that doesn't support foreign keys.
- You have to look up the syntax for declaring foreign keys.

5.2 Antipattern: Leave Out the Constraints

Even though it seems at first that skipping foreign key constraints makes your database design simpler, more flexible, or speedier, you pay for this in other ways. It becomes your responsibility to write code to ensure referential integrity manually.

Assuming Flawless Code

Many people's solution for referential integrity is to write application code so that data relationships are always satisfied. Every time you insert a row, make sure that values in foreign key columns reference existing values in the referenced table. Every time you delete a row, make sure that any child tables are also updated appropriately. In other words, the popular answer is simply to *make no mistakes.*

To avoid making referential integrity mistakes when you have no foreign key constraints, you'd have to run extra SELECT queries before you apply changes to confirm the change won't result in broken references. For instance, to insert a new row, you'd check that the parent row exists:

Download Keyless-Entry/anti/insert.sql

```sql
SELECT account_id FROM Accounts WHERE account_id = 1;
```

Then you could add a bug that references it:

Download Keyless-Entry/anti/insert.sql

```sql
INSERT INTO Bugs (reported_by) VALUES (1);
```

To delete a row, you'd have to make sure no child rows exist:

Download Keyless-Entry/anti/delete.sql

```sql
SELECT bug_id FROM Bugs WHERE reported_by = 1;
```

Then you could delete the account:

Download Keyless-Entry/anti/delete.sql

```sql
DELETE FROM Accounts WHERE account_id = 1;
```

What if the user with account_id 1 sneaks in and enters a new bug in the moment after your query and before you delete that account? This may seem unlikely, but as Gordon Letwin, architect of DOS 4, famously said, "One in a million is next Tuesday." That still leaves a broken reference—a bug reported by an account that no longer exists.

The only remedy is for you to explicitly lock the Bugs table while you're checking it and unlock it after you have finished deleting the account. Any architecture that requires that kind of locking is never going to do well when high concurrency and scalability are required.

Checking for Mistakes

The antisolution described in the story in this chapter uses developer-written scripts to report corrupted data.

For example, in our bugs database, the Bugs.status column references the lookup table BugStatus. To find bugs with an invalid status value, you could use a query like the following:

Download Keyless-Entry/anti/find-orphans.sql

```sql
SELECT b.bug_id, b.status
FROM Bugs b LEFT OUTER JOIN BugStatus s
  ON (b.status = s.status)
WHERE s.status IS NULL;
```

You can imagine that you'd have to write a similar query for every referential relationship in your database.

If you find yourself in the habit of checking for broken references like this, your next question is, how often do you need to run these checks? Running hundreds of checks every day, or even more frequently, becomes quite a chore.

What happens when you do find a broken reference? Can you correct it? You can—sometimes. For instance, you might change an invalid bug status value to a sensible default.[1]

Download Keyless-Entry/anti/set-default.sql

```
UPDATE Bugs SET status = DEFAULT WHERE status = 'BANANA';
```

Inevitably, there are other cases where you can't synthesize data to correct these kinds of mistakes. For example, the Bugs.reported_by column should reference the account of the user who reported the given bug, but if this value is invalid, which user's account should you use as a replacement?

"It's Not My Fault!"

It's pretty unlikely that all your code touching the database is perfect. You could easily perform similar database updates in several functions in your application. When you have to change the code, how can you be sure you've applied compatible changes to every case in your application?

You may also have users applying changes directly to the database, using an SQL query tool or through private scripts. It's easy to introduce broken references through ad hoc SQL statements. You should assume this will happen at some point in the life of your application.

You need the database to be *consistent*—that is, you need to be able to depend on references in the database being satisfied at all times. But you can't be certain that all applications and scripts that have accessed your database have made their changes correctly.

Catch-22 Updates

Many developers avoid foreign key constraints because the constraints make it inconvenient to update related columns in multiple tables. For instance, if you need to delete a row that other rows depend on, you have to delete the child rows first to avoid violating foreign key constraints:

Download Keyless-Entry/anti/delete-child.sql

```
DELETE FROM BugStatus WHERE status = 'BOGUS'; -- ERROR!

DELETE FROM Bugs WHERE status = 'BOGUS';

DELETE FROM BugStatus WHERE status = 'BOGUS'; -- retry succeeds
```

---

1. SQL supports using the keyword DEFAULT as shown.

You have to execute multiple statements manually, one for each child table. If you add another child table in a future enhancement to your database, you have to fix your code to delete from the new table too. But this problem is solvable.

The unsolvable problem is when you UPDATE a column that child rows depend on. You can't update the child rows before you update the parent, and you can't update the parent before you update the child values that reference it. You need to make both changes simultaneously, but that's impossible using two separate updates. It's a catch-22 scenario.

Download Keyless-Entry/anti/update-catch22.sql

```sql
UPDATE BugStatus SET status = 'INVALID' WHERE status = 'BOGUS'; -- ERROR!

UPDATE Bugs SET status = 'INVALID' WHERE status = 'BOGUS'; -- ERROR!
```

Some developers find these scenarios difficult to manage, so they decide not to use foreign keys at all. We'll see later how foreign keys address multitable updates and deletes in a simple and effective way.

5.3 How to Recognize the Antipattern

If you hear people use phrases like the following, they're probably practicing the Keyless Entry antipattern:

- "How do I query to check for a value that exists in one table and not the other table?"

  Usually this is to find orphan child rows whose parent has been updated or deleted.

- "Is there a quick way to check that a value exists in one table as part of my insert to a second table?"

  This is to ensure that the parent row exists. A foreign key does this for you automatically and uses any index on the parent table to make the check as efficient as possible.

- "Foreign keys? I was told not to use them because they slow down the database."

  Performance is often used as a justification for cutting corners, but it usually creates more problems than it solves—including performance problems.

5.4 Legitimate Uses of the Antipattern

Sometimes you're forced to use a database brand that doesn't support foreign key constraints (for example MySQL's MyISAM storage engine or SQLite prior to version 3.6.19). If that's the case, then you have to find a way to compensate, like the quality control scripts described in this chapter's story.

There are also some ultra-flexible database designs where foreign keys can't model the relationships. It should be a strong clue that you're using another SQL antipattern if you can't use traditional referential integrity constraints. For more detail, you may want to look at Chapter 6, *Entity-Attribute-Value*, on page 73 and Chapter 7, *Polymorphic Associations*, on page 89.

5.5 Solution: Declare Constraints

The Japanese phrase *poka-yoke* means "mistake-proofing."[2] This term refers to a manufacturing process that helps eliminate product defects by preventing, correcting, or drawing attention to errors as they occur. This practice improves quality and decreases the need for correction, which more than makes up for the cost of its use.

You can apply the poka-yoke principle to your database design by using foreign key constraints to enforce referential integrity. Instead of searching for and correcting data integrity mistakes, you can prevent these mistakes from entering your database in the first place.

Download Keyless-Entry/soln/foreign-keys.sql

```sql
CREATE TABLE Bugs (
  -- . . .
  reported_by      BIGINT UNSIGNED NOT NULL,
  status           VARCHAR(20) NOT NULL DEFAULT 'NEW',
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (status) REFERENCES BugStatus(status)
);
```

Your existing code and also ad hoc queries obey the same constraints, so there's no way for any forgotten code or back doors to bypass enforcement. The database rejects any improper change, no matter where the change comes from.

---

2. Poka-yoke was coined by industrial engineer Dr. Shigeo Shingo in his study of the Toyota Production System.

Using foreign keys saves you from writing unnecessary code and ensures that all your code works the same way if you change the database. This reduces the time to develop the code and also many hours of debugging and maintenance. The software industry average is 15 to 50 bugs per 1,000 lines of code. All other things being equal, if you have fewer lines of code, you have fewer bugs.

Supporting Multitable Changes

Foreign keys have another feature you can't mimic using application code: *cascading updates.*

Download **Keyless-Entry/soln/cascade.sql**

```sql
CREATE TABLE Bugs (
  -- . . .
  reported_by      BIGINT UNSIGNED NOT NULL,
  status           VARCHAR(20) NOT NULL DEFAULT 'NEW',
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id)
    ON UPDATE CASCADE
    ON DELETE RESTRICT,
  FOREIGN KEY (status) REFERENCES BugStatus(status)
    ON UPDATE CASCADE
    ON DELETE SET DEFAULT
);
```

This solution allows you to update or delete the parent row and lets the database takes care of any child rows that reference it. Updates to the parent tables BugStatus and Accounts propagate automatically to child rows in Bugs. There's no longer a catch-22 problem.

The way you declare the ON UPDATE or ON DELETE clauses in the foreign key constraint allow you to control the result of a cascading operation. For example, RESTRICT for the foreign key on reported_by means that you can't delete an account if some rows in Bugs reference it. The constraint blocks the delete and raises an error. Whereas if you delete a status value, any bugs with that status are automatically reset to the default status value.

In either case, the database changes both tables atomically. The foreign key references remain satisfied both before and after the changes.

If you add a new child table to the database, the foreign keys in the child table dictate the cascading behavior. You don't need to change your application code. Neither do you need to change anything about the parent table, no matter how many child tables reference it.

Overhead? Not Really

It's true that foreign key constraints have a bit of overhead. But compared to the alternative, foreign keys prove to be a lot more efficient.

- You don't need to run SELECT queries to check before you insert or update or delete.

- You don't need to lock tables to protect multitable changes.

- You don't need to run periodic quality control scripts to correct the inevitable orphans.

Foreign keys are easy to use, improve performance, and help you maintain consistent referential integrity during any data change, both simple and complex.

Make your database mistake-proof with constraints.

---

*If you try and take a cat apart to see how it works, the first
thing you have in your hands is a non-working cat.*
  ▶ Douglas Adams

<div align="right">

Chapter 6

</div>

Entity-Attribute-Value

"How do I count the number of rows by date?" This is an example of a simple task for a database programmer. This solution is covered in any introductory tutorial on SQL. It involves basic SQL syntax:

```sql
SELECT date_reported, COUNT(*)
FROM Bugs
GROUP BY date_reported;
```

However, the simple solution assumes two things:

- Values are stored in the same column, as in Bugs.date_reported.

- Values can be compared to one another so that GROUP BY can accurately group dates with equal values together.

What if you can't rely on those assumptions? What if the date is stored in the date_reported or report_date column or in any other column name that may be different on each row? What if dates can take a variety of different formats and the computer can't easily compare two dates?

You may encounter these problems and others when you employ the antipattern known as Entity-Attribute-Value.

6.1 Objective: Support Variable Attributes

Extensibility is frequently a goal of software projects. We would like to design software that can adapt fluidly to future usage with little or no additional programming.

This is not a new problem; similar arguments against the inflexibility of relational database metadata have been made almost continuously

since 1970, when the relational model was first proposed in *A Relational Model of Data for Large Shared Data Banks* [Cod70] by E. F. Codd.

A conventional table consists of attribute columns that are relevant for every row in the table, since every row represents an instance of a similar object. A different set of attributes represents a different type of object, so it belongs in a different table.

In modern object-oriented programming models, however, different object types can be related, for instance, by extending the same base type. In object-oriented design, these objects are considered instances of the same base type, as well as instances of their respective subtypes. We would like to store objects as rows in a single database table to simplify comparisons and calculations over multiple objects. But we also need to allow objects of each subtype to store their respective attribute columns, which may not apply to the base type or to other subtypes.

Let's use an example from our bugs database. In Figure 6.1, on the following page, we can see that a Bug and a Feature Request share some attributes in common, seen in the Issue base type. Every issue is associated with a person who reported it. It's also associated with a product, and it has a priority for completion. However, a Bug has some distinct attributes: the version of the product in which the bug occurs and the severity or impact of the bug. Likewise, a FeatureRequest may have its own attributes as well. For this example, suppose a feature is associated with a sponsor whose budget supports that feature's development.

6.2 Antipattern: Use a Generic Attribute Table

The solution that appeals to some programmers when they need to support variable attributes is to create a second table, storing attributes as rows. See the diagram showing the two tables in Figure 6.2, on page 76. Each row in this attribute table has three columns:

- The *Entity*. Typically this is a foreign key to a parent table that has one row per entity.

- The *Attribute*. This is simply the name of a column in a conventional table, but in this new design, we have to identify the attribute on each given row.

- The *Value*. Each entity has a value for each of its attributes.
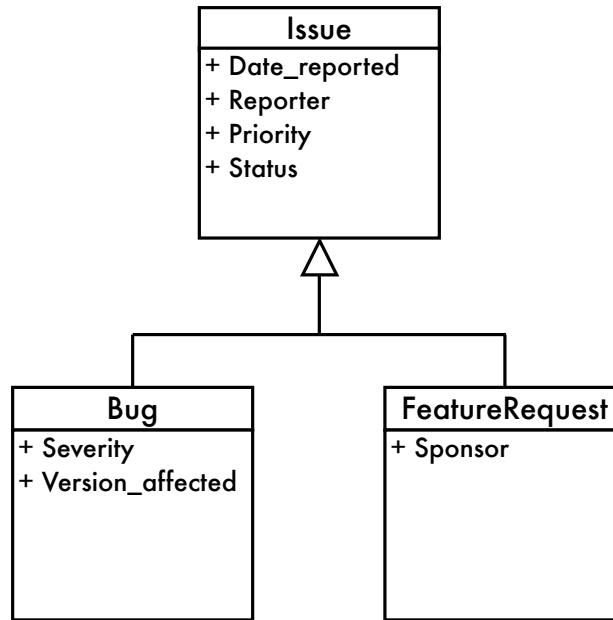
Figure 6.1: Object-oriented class diagram for bug types

For example, a given bug is an entity we identify by its primary key value 1234. It has an attribute called status. The value of that attribute for bug 1234 is *NEW*.

This design is called Entity-Attribute-Value, or *EAV* for short. It's also sometimes called *open schema*, *schemaless*, or *name-value pairs*.

Download EAV/anti/create-eav-table.sql

```
CREATE TABLE Issues (
  issue_id    SERIAL PRIMARY KEY
);

INSERT INTO Issues (issue_id) VALUES (1234);

CREATE TABLE IssueAttributes (
  issue_id    BIGINT UNSIGNED NOT NULL,
  attr_name   VARCHAR(100) NOT NULL,
  attr_value  VARCHAR(100),
  PRIMARY KEY (issue_id, attr_name),
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id)
);
```
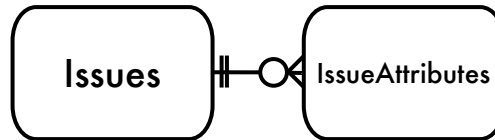
Figure 6.2: EAV entity relationship

```
INSERT INTO IssueAttributes (issue_id, attr_name, attr_value)
  VALUES
    (1234, 'product',          '1'),
    (1234, 'date_reported',    '2009-06-01'),
    (1234, 'status',           'NEW'),
    (1234, 'description',      'Saving does not work'),
    (1234, 'reported_by',      'Bill'),
    (1234, 'version_affected', '1.0'),
    (1234, 'severity',         'loss of functionality'),
    (1234, 'priority',         'high');
```

By adding one additional table, you seem to gain the following benefits:

- Both tables have few columns.

- The number of columns doesn't need to grow to support new attributes.

- You avoid a clutter of columns that contain null in rows where the attribute is inapplicable.

This appears to be an improved design. However, the simple database structure doesn't make up for the difficulty of using it.

Querying an Attribute

Your boss needs to run a report of the bugs reported per day. In a conventional table design, the Issues table would have a simple attribute column such as date_reported. To query all bugs with their report dates, your boss could use a simple query like this:

Download EAV/anti/query-plain.sql

```
SELECT issue_id, date_reported FROM Issues;
```

To get the same information as the previous query using the EAV design, your boss needs to fetch rows from the IssueAttributes table that

stores an attribute named by the string *date_reported*. This query is more verbose but less clear.

Download EAV/anti/query-eav.sql

```sql
SELECT issue_id, attr_value AS "date_reported"
FROM IssueAttributes
WHERE attr_name = 'date_reported';
```

Supporting Data Integrity

When you use EAV, you sacrifice many advantages that a conventional database design would have given you.

You Can't Make Mandatory Attributes

To help your boss generate accurate project reports, you should also require that the date_reported attribute has a value. In a conventional database design, it would be simple to enforce a mandatory column by declaring the column NOT NULL.

In the EAV design, each attribute corresponds to a row in the IssueAttributes table, not a column. You would need a constraint that checks that a row exists for each issue_id value, and the row must have the string *date_reported* in its attr_name column.

However, SQL doesn't support a constraint that can do this. So, you must write application code to enforce it. If you do find a bug with no reported date, should you add a value for this attribute? What value should you give it? If you make a guess or use some default value for a missing attribute, how does that affect the accuracy of your boss's reports?

You Can't Use SQL Data Types

Your boss tells you he is having trouble running his report because people have entered dates in different formats or sometimes even a string that isn't a date. In a conventional database, you can prevent this if you declared the column with the DATE data type.

Download EAV/anti/insert-plain.sql

```sql
INSERT INTO Issues (date_reported) VALUES ('banana'); -- ERROR!
```

In the EAV design, the data type of the IssueAttributes.attr_value column is typically a string to accommodate all possible attributes in a single column. So, it has no way of rejecting invalid data.

Download **EAV/anti/insert-eav.sql**

```sql
INSERT INTO IssueAttributes (issue_id, attr_name, attr_value)
  VALUES (1234, 'date_reported', 'banana');  -- Not an error!
```

Some people try to extend the EAV design by defining a separate attr_value column for each SQL data type, leaving null in the unused columns. This allows you to use data types but makes queries even worse:

Download **EAV/anti/data-types.sql**

```sql
SELECT issue_id, COALESCE(attr_value_date, attr_value_datetime,
  attr_value_integer, attr_value_numeric, attr_value_float,
  attr_value_string, attr_value_text) AS "date_reported"
FROM IssueAttributes
WHERE attr_name = 'date_reported';
```

You would need to add even more columns to support user-defined data types or domains.

You Can't Enforce Referential Integrity

In a conventional database, you can restrict the range of some attributes by defining a foreign key to a lookup table. For example, the status attribute of a bug or issue should be one of a short list of values stored in the BugStatus table.

Download **EAV/anti/foreign-key-plain.sql**

```sql
CREATE TABLE Issues (
  issue_id        SERIAL PRIMARY KEY,
  -- other columns
  status          VARCHAR(20) NOT NULL DEFAULT 'NEW',
  FOREIGN KEY (status) REFERENCES BugStatus(status)
);
```

In the EAV design, you can't apply this kind of constraint on the attr_value column. A referential integrity constraint applies to every row in the table.

Download **EAV/anti/foreign-key-eav.sql**

```sql
CREATE TABLE IssueAttributes (
  issue_id        BIGINT UNSIGNED NOT NULL,
  attr_name       VARCHAR(100) NOT NULL,
  attr_value      VARCHAR(100),
  FOREIGN KEY (attr_value) REFERENCES BugStatus(status)
);
```

If you define this constraint, it would force *every* attribute to match a value in BugStatus, not just the status attribute.

You Can't Make Up Attribute Names

Your boss' reports are still not reliable. You find that attributes are not being named consistently. One bug uses an attribute named by the string *date_reported*, but another bug names the attribute by the string *report_date*. Both are clearly intended to represent the same information.

How would you count bugs per date?

Download EAV/anti/count.sql

```sql
SELECT date_reported, COUNT(*) AS bugs_per_date
FROM (SELECT DISTINCT issue_id, attr_value AS date_reported
      FROM IssueAttributes
      WHERE attr_name IN ('date_reported', 'report_date'))
GROUP BY date_reported;
```

How would you know if a given bug has stored an attribute by yet another name? How would you know if a given bug has stored a given attribute twice, by two different names? How can you prevent such mistakes?

One remedy might be to declare a foreign key on the attr_name column to a lookup table that contains your approved attribute names. However, this doesn't support attributes you define on the fly for each entity. That's a common use of the EAV design.

Reconstructing a Row

It's natural to retrieve a row from the Issues table with all its attributes in columns. You want to fetch an issue in a single row as though it were stored in a conventional table.

issue_id	date_reported	status	priority	description
1234	2009-06-01	NEW	HIGH	Saving does not work

Because each attribute is stored on a separate row of the IssueAttributes table, retrieving them all as part of a single row requires a join for each attribute. You must know all attributes at the time you write this query. The following query reconstructs the row shown earlier:

Download EAV/anti/reconstruct.sql

```sql
SELECT i.issue_id,
  i1.attr_value AS "date_reported",
  i2.attr_value AS "status",
  i3.attr_value AS "priority",
  i4.attr_value AS "description"
```

```
FROM Issues AS i
  LEFT OUTER JOIN IssueAttributes AS i1
    ON i.issue_id = i1.issue_id AND i1.attr_name = 'date_reported'
  LEFT OUTER JOIN IssueAttributes AS i2
    ON i.issue_id = i2.issue_id AND i2.attr_name = 'status'
  LEFT OUTER JOIN IssueAttributes AS i3
    ON i.issue_id = i3.issue_id AND i3.attr_name = 'priority';
  LEFT OUTER JOIN IssueAttributes AS i4
    ON i.issue_id = i4.issue_id AND i4.attr_name = 'description';
WHERE i.issue_id = 1234;
```

You must use outer joins because inner joins would cause the query to return no rows if any one of the attributes were not present in the IssueAttributes table. As the number of attributes increases, so does the number of joins, and the cost of this query increases exponentially.

6.3 How to Recognize the Antipattern

If you hear phrases like the following spoken by your project team, it's a clue that someone is employing the EAV antipattern:

- "This database is totally extensible without metadata changes. You can define new attributes at runtime."

  Relational databases don't support that degree of flexibility. When someone claims to have designed an arbitrarily extensible database, they're probably using the EAV design.

- "What's the maximum number of joins I can do in a query?"

  If you need a query to support such a high number of joins that you're concerned about exceeding the database's limits, you may have a problem in your database design. It's common for an EAV design to lead to this problem.

- "I can't figure out how to write a report for our e-commerce platform. We need to hire a consultant to do it for us."

  It seems that many turnkey database-driven software packages designed for customizability use the EAV design. This makes most common reporting queries very complex or even impractical.

6.4 Legitimate Uses of the Antipattern

It's hard to justify using the EAV antipattern in a relational database. You have to compromise too many features that are strengths of the

relational paradigm. But that doesn't address the legitimate need in some applications to support dynamic attributes.

Most applications that need schemaless data really need it for only a few tables or even just one table. The rest of your data requirements conform to standard table designs. If you account for the extra work and risk of EAV in your project plan, it may be the lesser evil to use it sparingly. But keep in mind that experienced database consultants report that systems using EAV become unwieldy within a year.

If you have nonrelational data management needs, the best answer is to use a *nonrelational* technology. This is a book about SQL, not about SQL alternatives, so I'll list only a sampling of these technologies:

- *Berkeley DB* is a popular key-value store that's easy to embed in a variety of applications.

  http://www.oracle.com/technology/products/berkeley-db/

- *Cassandra* is a distributed column-oriented database developed at Facebook and contributed to the Apache project.

  http://incubator.apache.org/cassandra/

- *CouchDB* is a document-oriented database—a distributed key-value store that encodes values in JSON.

  http://couchdb.apache.org/

- *Hadoop* and *HBase* make up an open source DBMS inspired by Google's MapReduce algorithm for distributing queries against very large-scale semistructured data stores.

  http://hadoop.apache.org/

- *MongoDB* is a document-oriented database like CouchDB.

  http://www.mongodb.org/

- *Redis* is a document-oriented in-memory database.

  http://code.google.com/p/redis/

- *Tokyo Cabinet* is a key-value store, designed in the vein of POSIX DBM, GNU GDBM, or Berkeley DB.

  http://1978th.net/

Many other nonrelational projects are also emerging. However, the weaknesses of EAV relative to relational databases also apply to these

alternatives. When metadata is fluid, it's harder to formulate simple queries. Applications spend a lot of energy discovering the structure of data and adapting to it.

6.5 Solution: Model the Subtypes

If EAV seems like the right design, you should take a second look before you implement it. If you do some good old-fashioned analysis, you will probably find that your project's data can be modeled in a traditional table design more easily and with greater assurance of data integrity.

There are several ways to store such data without using EAV. Most solutions work best when you have a finite number of subtypes and you know the attribute of each subtype. Which solution is best to use depends on how you intend to query the data, so you should decide on a design on a case-by-case basis.

Single Table Inheritance

The simplest design is to store all related types in one table, with distinct columns for every attribute that exists in any type. Use one attribute to define the subtype of a given row. In this example, this attribute is called issue_type. Some attributes are common to all subtypes. Many attributes are subtype-specific, and these columns must be given a null value on any row storing an object for which the attribute does not apply; the columns with non-null values become sparse.

The name of this design comes from Martin Fowler's book *Patterns of Enterprise Application Architecture* [Fow03].

Download EAV/soln/create-sti-table.sql

```sql
CREATE TABLE Issues (
  issue_id         SERIAL PRIMARY KEY,
  reported_by      BIGINT UNSIGNED NOT NULL,
  product_id       BIGINT UNSIGNED,
  priority         VARCHAR(20),
  version_resolved VARCHAR(20),
  status           VARCHAR(20),
  issue_type       VARCHAR(10),  -- BUG or FEATURE
  severity         VARCHAR(20),  -- only for bugs
  version_affected VARCHAR(20),  -- only for bugs
  sponsor          VARCHAR(50),  -- only for feature requests
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id)
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

As new object types are introduced, the database must accommodate the attributes that describe these new object types. You must alter the table to add more columns as you add distinct attributes for the new object types. You may encounter a practical limit on the number of columns per table.

Another limitation of Single Table Inheritance is that there is no metadata to define which attributes belong to which subtypes. In your application, you can ignore some attributes if you know they don't apply to the object subtype on a given row. But you must track manually which attributes are applicable to each subtype. It would be better if you could use metadata to define this in the database.

Single Table Inheritance is best when you have few subtypes and few subtype-specific attributes, and you need to use a single-table database access pattern like Active Record.

Concrete Table Inheritance

Another solution is to create a separate table for each subtype. Every table contains the same attributes that are common to the base type, as well as the respective subtype-specific attribute. The name of this design also comes from Martin Fowler's book.

Download EAV/soln/create-concrete-tables.sql

```sql
CREATE TABLE Bugs (
  issue_id         SERIAL PRIMARY KEY,
  reported_by      BIGINT UNSIGNED NOT NULL,
  product_id       BIGINT UNSIGNED,
  priority         VARCHAR(20),
  version_resolved VARCHAR(20),
  status           VARCHAR(20),
  severity         VARCHAR(20), -- only for bugs
  version_affected VARCHAR(20), -- only for bugs
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);

CREATE TABLE FeatureRequests (
  issue_id         SERIAL PRIMARY KEY,
  reported_by      BIGINT UNSIGNED NOT NULL,
  product_id       BIGINT UNSIGNED,
  priority         VARCHAR(20),
  version_resolved VARCHAR(20),
  status           VARCHAR(20),
  sponsor          VARCHAR(50),  -- only for feature requests
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

An advantage of Concrete Table Inheritance over Single Table Inheritance is that you are prevented from storing a row containing values for attributes that don't apply to that row's subtype. If you reference an attribute column that doesn't exist in that table, the database informs you of the error automatically. For example, the severity column does not appear in the FeatureRequests table:

Download EAV/soln/insert-concrete.sql

```
INSERT INTO FeatureRequests (issue_id, severity) VALUES ( ... ); -- ERROR!
```

Another advantage of Concrete Table Inheritance is that you don't need an extra attribute to define the subtype on each row, as you do in the Single Table Inheritance design.

However, it's hard to tell the common attributes from subtype-specific attributes. Also, if you add a new attribute to the set of common attributes, you must alter every subtype table.

No metadata shows that the data stored in these subtype tables belong to related objects. That is, if a programmer new to your project looks at the table definitions, he would see that some columns are common to all these subtype tables, but the metadata does not tell him whether any logical relationship exists or whether the tables have similarities merely by coincidence.

If you want to search all objects regardless of their subtypes, this is complicated if each subtype is stored in a separate table. To make this query easier, define a view that is the union of the tables, selecting only common attributes.

Download EAV/soln/view-concrete.sql

```
CREATE VIEW Issues AS
  SELECT b.*, 'bug' AS issue_type
  FROM Bugs AS b
   UNION ALL
  SELECT f.*, 'feature' AS issue_type
  FROM FeatureRequests AS f;
```

The Concrete Table Inheritance design is best used when you seldom need to query against all subtypes at once.

Class Table Inheritance

A third solution mimics inheritance, as though tables were object-oriented classes. Create a single table for the base type, containing attributes common to all subtypes. Then for each subtype, create another

table, with a primary key that also serves as a foreign key to the base table. The name of this design also comes from Martin Fowler's book.

Download EAV/soln/create-class-tables.sql

```sql
CREATE TABLE Issues (
  issue_id         SERIAL PRIMARY KEY,
  reported_by      BIGINT UNSIGNED NOT NULL,
  product_id       BIGINT UNSIGNED,
  priority         VARCHAR(20),
  version_resolved VARCHAR(20),
  status           VARCHAR(20),
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);

CREATE TABLE Bugs (
  issue_id         BIGINT UNSIGNED PRIMARY KEY,
  severity         VARCHAR(20),
  version_affected VARCHAR(20),
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id)
);

CREATE TABLE FeatureRequests (
  issue_id         BIGINT UNSIGNED PRIMARY KEY,
  sponsor          VARCHAR(50),
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id)
);
```

The one-to-one relationship is enforced by the metadata, since the dependent table's foreign key is also a primary key and thus must be unique. This solution provides an efficient way to search against all subtypes, as long as your search references only the base type's attributes. Once you've found the entries that match your search, you can get the subtype-specific attributes by querying against the respective subtype tables.

You don't need to know from the row in the base table what subtype the row represents; as long as you have a small number of subtypes, you can write a join against all of them at once, producing a sparse result set like in the Single Table Inheritance table. Attributes are null where the attribute doesn't apply in the subtype for a given row.

Download EAV/soln/select-class.sql

```sql
SELECT i.*, b.*, f.*
FROM Issues AS i
  LEFT OUTER JOIN Bugs AS b USING (issue_id)
  LEFT OUTER JOIN FeatureRequests AS f USING (issue_id);
```

This is also a good candidate for defining a VIEW.

This design is best when you often need to query across all subtypes, referencing the columns they have in common.

Semistructured Data

If you have many subtypes or if you must support new attributes frequently, you can add a BLOB column to store data in a format such as XML or JSON, which encodes both the attribute names and their values. Martin Fowler calls this pattern the *Serialized LOB*.

Download EAV/soln/create-blob-tables.sql

```
CREATE TABLE Issues (
  issue_id         SERIAL PRIMARY KEY,
  reported_by      BIGINT UNSIGNED NOT NULL,
  product_id       BIGINT UNSIGNED,
  priority         VARCHAR(20),
  version_resolved VARCHAR(20),
  status           VARCHAR(20),
  issue_type       VARCHAR(10),   -- BUG or FEATURE
  attributes       TEXT NOT NULL, -- all dynamic attributes for the row
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

The advantage of this design is that it's completely extensible. You can store new attributes in the blob at any time. Every row stores a potentially distinct set of attributes, so you have as many subtypes as you have rows.

The disadvantage is that SQL has little support for accessing specific attributes in such a structure. You can't easily select individual attributes within the blob for row-based restriction, aggregate calculation, sorting, or other operations. You must fetch the whole blob of attributes as a single value and write application code to decode and interpret the attributes.

This design is best when you can't limit yourself to a finite set of subtypes and when you need complete flexibility to define new attributes at any time.

Post-Processing

Unfortunately, sometimes you're stuck with the EAV design, such as if you inherited a project and can't change it or if your company acquired a third-party software platform that uses EAV. If this is the case, familiarize yourself with the trouble areas in the "Antipattern" section so

you can anticipate and plan for the extra work it takes to work with this design.

Above all, don't try to write queries that fetch entities as a single row as though data were stored in a conventional table. Instead, query the attributes associated with the entity and fetch them as a set of rows, like they are stored.

```sql
SELECT issue_id, attr_name, attr_value
FROM IssueAttributes
WHERE issue_id = 1234;
```

The result of this query might look like the following:

issue_id	attr_name	attr_value
1234	date_reported	2009-06-01
1234	description	Saving does not work
1234	priority	HIGH
1234	product	Open RoundFile
1234	reported_by	Bill
1234	severity	loss of functionality
1234	status	NEW

This query is easier for you to write, and it's easier for the database to process. It returns all the attributes associated with the issue, even if you don't know how many there are when you write the query.

To use a result in this format, you need to write application code to loop over the rows of the result set and set properties of an object in your application. See the following PHP code for an example:

```php
<?php

$objects = array();

$stmt = $pdo->query(
    "SELECT issue_id, attr_name, attr_value
     FROM IssueAttributes
     WHERE issue_id = 1234");

while ($row = $stmt->fetch()) {
  $id    = $row['issue_id'];
  $field = $row['attr_name'];
  $value = $row['attr_value'];
```

```
  if (!array_key_exists($id, $objects)) {
    $objects[$id] = new stdClass();
  }

  $objects[$id]->$field = $value;
}
```

This might seem like too much work, but it's the consequence of a
system-within-a-system like EAV. SQL already offers a way to identify
distinct attributes—in distinct columns. By using EAV, you're layering
onto SQL a new way to identify attributes, so it should be no surprise
that SQL supports this awkwardly and inefficiently.

Use metadata for metadata.

Chapter 7

Polymorphic Associations

Let's allow users to make *comments* on bugs. A given bug may have many comments, but any given comment must pertain to a single bug. So, there's a one-to-many relationship between Bugs and Comments. The entity-relationship diagram for this kind of simple association is shown in Figure 7.1, on the following page, and the following SQL shows how you would create this table:

Download **Polymorphic/intro/comments.sql**

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

However, you might have two tables you can comment on. Bugs and FeatureRequests are similar entities, although you might store them as separate tables (see Section 6.5, *Concrete Table Inheritance*, on page 83). You'd like to store Comments in a single table regardless of whether they pertain to either type of issue—a bug or a feature—but you can't declare a foreign key that references multiple parent tables. The following declaration is nonsense:

Download **Polymorphic/intro/nonsense.sql**

```sql
  ...
  FOREIGN KEY (issue_id)
      REFERENCES Bugs(issue_id) OR FeatureRequests(issue_id)
);
```
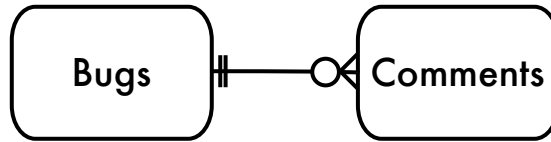
Figure 7.1: Simple association

Developers also try to write invalid SQL to query multiple tables, such as the following:

Download **Polymorphic/intro/nonsense.sql**

```
SELECT c.*, i.summary, i.status
FROM Comments AS c
JOIN c.issue_type AS i USING (issue_id);
```

But you can't join to a different table per row in SQL. SQL syntax requires that you name all the tables literally at the time you submit the query. The tables cannot vary during the query. What's wrong with this picture, and how do we solve it?

7.1 Objective: Reference Multiple Parents

The Scarecrow in *The Wizard of Oz* gives Dorothy uncertain directions when she asks which fork in the road she should take to get to the Emerald City. What should be a clear answer to Dorothy's simple question just confuses her when the Scarecrow tries to give her two answers at once.

This kind of association is illustrated in the entity-relationship diagram in Figure 7.2, on the following page. The foreign key in the child table "forks," so a row in the Comments table matches either a row in the Bugs table or a row in the FeatureRequests table. The curved arc in the diagram indicates an exclusive choice: a given comment must reference either one bug or one feature request.
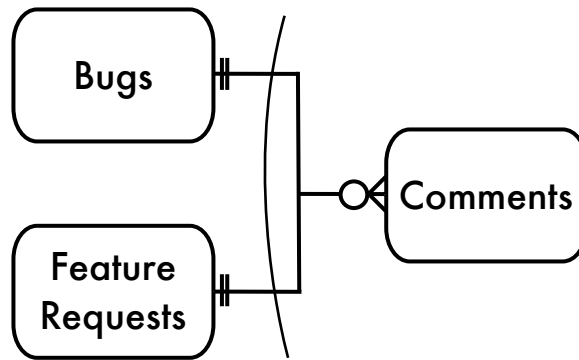
Figure 7.2: Polymorphic association

7.2 Antipattern: Use Dual-Purpose Foreign Key

A solution for these cases has become popular enough to be given a name, *Polymorphic Associations*. This is also sometimes called a *promiscuous* association, because it can reference multiple tables.

Defining a Polymorphic Association

To make Polymorphic Associations work, you must add an extra string column alongside the foreign key on issue_id. The extra column contains the name of the parent table referenced by the current row. In this example, the new column is called issue_type, and it contains either *Bugs* or *FeatureRequests*, corresponding to the names of the two possible parent tables in this association.

Download **Polymorphic/anti/comments.sql**

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  issue_type   VARCHAR(20),      -- "Bugs" or "FeatureRequests"
  issue_id     BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME,
  comment      TEXT,
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);
```

You can see one difference immediately: the foreign key declaration for issue_id is missing. In fact, since a foreign key must specify exactly one

> ### Mixing Data with Metadata
>
> You might have noticed a similar characteristic between the Polymorphic Associations antipattern and the Entity-Attribute-Value antipattern described in the previous chapter. In both antipatterns, the name of a metadata object is stored as a string value. In EAV, the name of an attribute column is stored as a string in the attr_name column. In Polymorphic Associations, the names of the parent tables are stored in the issue_type column. This is sometimes called *mixing data with metadata*. This concept appears in another form in Chapter 8, *Multicolumn Attributes*, on page 102.

table, using a Polymorphic Association means that you can't declare this association in metadata. As a result, there is no enforcement of data integrity to ensure that the value in Comments.issue_id matches a value in the parent table.

Likewise, no metadata ensures that the string in Comments.issue_type corresponds to a table that exists in this database.

Querying a Polymorphic Association

The issue_id value in the Comments table may occur in the primary key column of both parent tables, Bugs and FeatureRequests. Or the value may occur in one parent table but be missing in the other parent table. It's therefore crucial to use the issue_type correctly when joining the child table to the parent table. You must not match an issue_id value to the FeatureRequests table if it was intended to be matched to the Bugs table.

For example, this will retrieve comments for a given bug by its primary key value 1234:

Download **Polymorphic/anti/select.sql**

```
SELECT *
FROM Bugs AS b JOIN Comments AS c
  ON (b.issue_id = c.issue_id AND c.issue_type = 'Bugs')
WHERE b.issue_id = 1234;
```

Although the previous query works if bugs are stored in the single table Bugs, you run into a problem when Comments is associated with both tables Bugs and FeatureRequests. In SQL, you must specify all tables

in a join; you cannot join Comments to two separate tables, switching between them row by row, depending on the value in the Comments.issue_type column.

To retrieve either a bug or a feature given a specific comment, you need to run a query with an outer join to both parent tables. Only one of the parent tables will satisfy its join, since part of the join condition relies on the value in the Comment.issue_type column. Using an outer join means that fields from the table that does not match contain null in the result set.

Download **Polymorphic/anti/select.sql**

```sql
SELECT *
FROM Comments AS c
  LEFT OUTER JOIN Bugs AS b
    ON (b.issue_id = c.issue_id AND c.issue_type = 'Bugs')
  LEFT OUTER JOIN FeatureRequests AS f
    ON (f.issue_id = c.issue_id AND c.issue_type = 'FeatureRequests');
```

The result may look something like this:

c.comment_id	c.issue_type	c.issue_id	c.comment	b.issue_id	f.issue_id
6789	Bugs	1234	It crashes!	1234	NULL
9876	Feature...	2345	Great idea!	NULL	2345

Non-Object-Oriented Example

In the example of Bugs and FeatureRequests, these two parent tables are meant to model related subtypes. Polymorphic Associations may also be used when the parent tables are completely unrelated to each other. For example, in an ecommerce database, both tables Users and Orders may be associated with Addresses, as illustrated in Figure 7.3, on the following page.

Download **Polymorphic/anti/addresses.sql**

```sql
CREATE TABLE Addresses (
  address_id   SERIAL PRIMARY KEY,
  parent       VARCHAR(20),      -- "Users" or "Orders"
  parent_id    BIGINT UNSIGNED NOT NULL,
  address      TEXT
);
```

In this case, the Addresses table contains a polymorphic column that names either *Users* or *Orders* as the parent table for a given address. Notice that you have to choose one or the other. You can't associate a
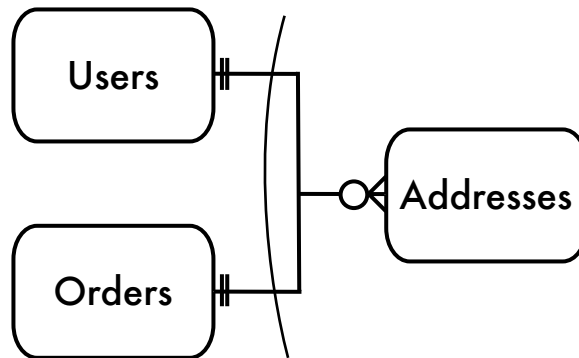
Figure 7.3: Polymorphic Associations for addresses

given address with both a user and an order, even an order placed by
that user, to ship merchandise to himself.

Also, if a user has a shipping address as well as a billing address, you
need some way to make this distinction in the Addresses table; likewise,
any other parents need to note the special usage of addresses in the
Addresses table. These notes can propagate like weeds.

Download Polymorphic/anti/addresses.sql

```sql
CREATE TABLE Addresses (
  address_id    SERIAL PRIMARY KEY,
  parent        VARCHAR(20),      -- "Users" or "Orders"
  parent_id     BIGINT UNSIGNED NOT NULL,
  users_usage   VARCHAR(20),      -- "billing" or "shipping"
  orders_usage  VARCHAR(20),      -- "billing" or "shipping"
  address       TEXT
);
```

7.3 How to Recognize the Antipattern

If you hear statements like the following, it's a clue that the Polymorphic
Associations antipattern is being employed:

- "This tagging schema allows you to associate a tag (or other attri-
  bute) with *any* other resource in the database."

  Like in EAV, you should be suspicious of any claims of unlimited
  flexibility.

- "You can't declare foreign keys in our database design."

  This is another red flag. Foreign keys are a fundamental feature of relational databases, and a design that can't work with proper referential integrity has a lot of problems.

- "What's the entity_type column for? Oh, that tells you which thing this other column points to."

  Any foreign key must reference the same table on all rows.

The Ruby on Rails framework supports Polymorphic Associations by declaring Active Record classes with the :polymorphic attribute. For example, you could associate Comments to Bugs and FeatureRequests as follows:

Download **Polymorphic/recog/commentable.rb**

```ruby
class Comment < ActiveRecord::Base
  belongs_to :commentable, :polymorphic => true
end

class Bug < ActiveRecord::Base
  has_many :comments, :as => :commentable
end

class FeatureRequest < ActiveRecord::Base
  has_many :comments, :as => :commentable
end
```

The Hibernate framework for Java supports Polymorphic Associations using a variety of schema declarations.[1]

7.4 Legitimate Uses of the Antipattern

You should avoid the Polymorphic Associations antipattern—use constraints like foreign keys to ensure referential integrity. Polymorphic Associations often relies too much on application code instead of metadata.

You may find that this antipattern is unavoidable if you use an object-relational programming framework such as Hibernate. Such a framework may mitigate the risks introduced by Polymorphic Associations by encapsulating application logic to maintain referential integrity. If

---

1. See http://www.hibernate.org/hib_docs/reference/en/html/inheritance.html. "Hibernate" is a service mark of Red Hat, Inc.

you choose a mature and reputable framework, then you have some confidence that its designers have written the code to implement the association without error. However, if you are implementing Polymorphic Associations from scratch without the aid of a framework, you're reinventing the wheel.

7.5 Solution: Simplify the Relationship

It's better to redesign your database to avoid the weaknesses of Polymorphic Associations but still support the data modeling you need. The following sections describe a few solutions that accommodate the data relationship but make better use of metadata to enforce integrity.

Reverse the Reference

One solution to this antipattern is simple once you see the nature of the problem: *Polymorphic Associations are backward.*

Creating Intersection Tables

A foreign key in the child table Comments can't reference multiple parent tables, so instead, use multiple foreign keys to reference the Comments table. Create a separate intersection table for each parent table, and in each intersection table include a foreign key to Comments, as well as a foreign key to the respective parent table. This design is illustrated in the entity-relationship diagram in Figure 7.4, on the next page.

Download Polymorphic/soln/reverse-reference.sql

```sql
CREATE TABLE BugsComments (
  issue_id    BIGINT UNSIGNED NOT NULL,
  comment_id  BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (issue_id, comment_id),
  FOREIGN KEY (issue_id) REFERENCES Bugs(issue_id),
  FOREIGN KEY (comment_id) REFERENCES Comments(comment_id)
);

CREATE TABLE FeaturesComments (
  issue_id    BIGINT UNSIGNED NOT NULL,
  comment_id  BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (issue_id, comment_id),
  FOREIGN KEY (issue_id) REFERENCES FeatureRequests(issue_id),
  FOREIGN KEY (comment_id) REFERENCES Comments(comment_id)
);
```
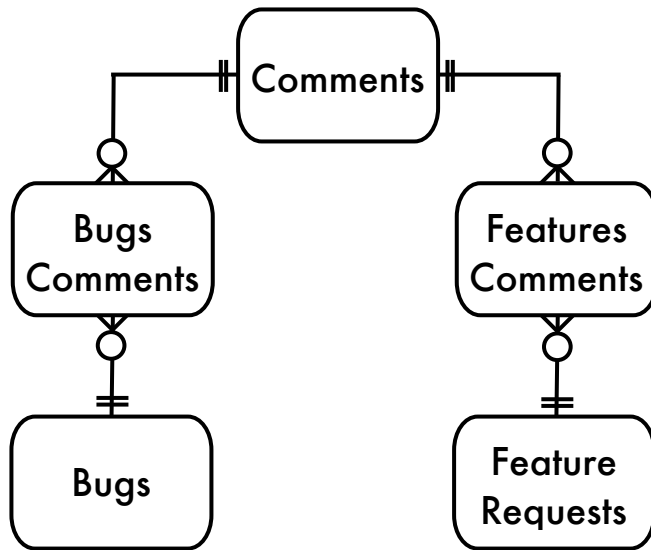
Figure 7.4: Reversing a polymorphic association

This solution removes the need for the Comments.issue_type column. Now the metadata can enforce data integrity, instead of relying on application code to manage the associations without error.

Putting Up Traffic Lights

A potential weakness of this solution is that it permits associations that you might not want to be permitted. Intersection tables usually model many-to-many associations, so this would allow a given comment to be associated with multiple bugs or multiple feature requests. However, you probably want each comment to pertain to only one bug or one feature request. You can enforce at least part of this rule by declaring a UNIQUE constraint on the comment_id column of each intersection table.

Download Polymorphic/soln/reverse-unique.sql

```
CREATE TABLE BugsComments (
  issue_id     BIGINT UNSIGNED NOT NULL,
  comment_id   BIGINT UNSIGNED NOT NULL,
  UNIQUE KEY (comment_id),
  PRIMARY KEY (issue_id, comment_id),
  FOREIGN KEY (issue_id) REFERENCES Bugs(issue_id),
  FOREIGN KEY (comment_id) REFERENCES Comments(comment_id)
);
```

This ensures that a given comment can be referenced only once in the intersection table, which naturally prevents it from being associated with multiple bugs or multiple feature requests. However, the meta-data doesn't prevent a given comment from being referenced once in both intersection tables, associating the comment with both a bug and a feature request. This is probably not what you want, but ensuring against it remains the responsibility of your application code.

Looking Both Ways

You can query comments given a specific bug or feature request simply by using the intersection table.

```
Download  Polymorphic/soln/reverse-join.sql
SELECT *
FROM BugsComments AS b
  JOIN Comments AS c USING (comment_id)
WHERE b.issue_id = 1234;
```

You can query the matching bug or feature request based on an in-stance of a comment by using an outer join to both intersection tables. You have to name all the possible parent tables, but that's no more complex than the query you had to use in the Polymorphic Associations antipattern. Also, you can depend on referential integrity when using intersection tables, whereas with Polymorphic Associations you couldn't.

```
Download  Polymorphic/soln/reverse-join.sql
SELECT *
FROM Comments AS c
  LEFT OUTER JOIN (BugsComments JOIN Bugs AS b USING (issue_id))
    USING (comment_id)
  LEFT OUTER JOIN (FeaturesComments JOIN FeatureRequests AS f USING (issue_id))
    USING (comment_id)
WHERE c.comment_id = 9876;
```

Merging Lanes

Sometimes you need to make the result of a query against multiple parent tables appear is if you had stored the parents in a single table (see Section 6.5, *Single Table Inheritance*, on page 82). You can do this in either of two ways.

First look at the following query using UNION:

```
Download  Polymorphic/soln/reverse-union.sql
SELECT b.issue_id, b.description, b.reporter, b.priority, b.status,
    b.severity, b.version_affected,
    NULL AS sponsor
```

```
     FROM Comments AS c
       JOIN (BugsComments JOIN Bugs AS b USING (issue_id))
         USING (comment_id)
     WHERE c.comment_id = 9876;
UNION
     SELECT f.issue_id, f.description, f.reporter, f.priority, f.status,
       NULL AS severity, NULL AS version_affected,
       f.sponsor
     FROM Comments AS c
       JOIN (FeaturesComments JOIN FeatureRequests AS f USING (issue_id))
         USING (comment_id)
     WHERE c.comment_id = 9876;
```

This query should be guaranteed to return a single row if your application has associated each comment with exactly one parent table. Since query results can be combined with UNION only if their columns are the same in number and data type, you must provide null placeholders for columns that are unique to each parent table. You must list the columns in the same order in both queries involved in the UNION.

Alternatively, look at the following query using the SQL COALESCE() function. This function returns its first non-null argument. Since you are using an outer join in the query, a comment that pertains to a feature request and has no matching row in Bugs would return all fields in b.* as null. Likewise, all fields in f.* would be null if the comment pertains to a bug instead of a feature request. List the fields specific to one parent table or the other in a simple manner; if they are irrelevant to the matching parent table, they are returned as null.

Download Polymorphic/soln/reverse-coalesce.sql

```
SELECT c.*,
  COALESCE(b.issue_id,    f.issue_id   ) AS issue_id,
  COALESCE(b.description, f.description) AS description,
  COALESCE(b.reporter,    f.reporter   ) AS reporter,
  COALESCE(b.priority,    f.priority   ) AS priority,
  COALESCE(b.status,      f.status     ) AS status,
  b.severity,
  b.version_affected,
  f.sponsor
FROM Comments AS c
  LEFT OUTER JOIN (BugsComments JOIN Bugs AS b USING (issue_id))
    USING (comment_id)
  LEFT OUTER JOIN (FeaturesComments JOIN FeatureRequests AS f USING (issue_id))
    USING (comment_id)
WHERE c.comment_id = 9876;
```

Both of these queries are pretty complex, so they're good candidates for a database view, and you can use them more simply in your application.
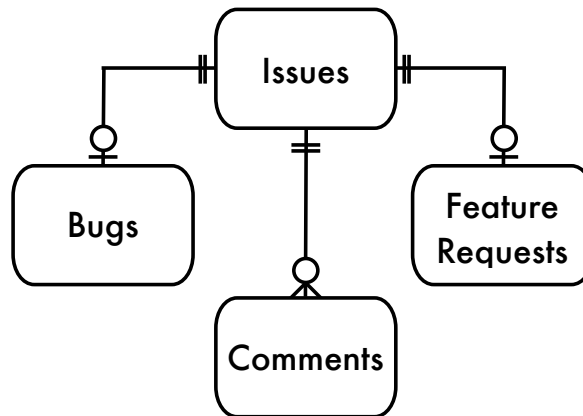
Figure 7.5: Association of Comments to the Base Issues table

Create a Common Super-Table

In object-oriented polymorphism, two subtypes can be referenced similarly because they implicitly share a common supertype. In SQL, the Polymorphic Associations antipattern leaves out that crucial entity: the common supertype. You can fix that by creating a base table that all of your parent tables extend (see Section 6.5, *Class Table Inheritance*, on page 84). Add the foreign key in the child Comments table to reference the base table. You don't need an issue_type column. This solution is illustrated in the entity-relationship diagram in Figure 7.5.

Download Polymorphic/soln/super-table.sql

```sql
CREATE TABLE Issues (
  issue_id    SERIAL PRIMARY KEY
);

CREATE TABLE Bugs (
  issue_id    BIGINT UNSIGNED PRIMARY KEY,
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id),
  . . .
);

CREATE TABLE FeatureRequests (
  issue_id    BIGINT UNSIGNED PRIMARY KEY,
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id),
  . . .
);
```

SOLUTION: SIMPLIFY THE RELATIONSHIP ◀ 101

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

Note that the primary keys of Bugs and FeatureRequests are also foreign keys. They reference the surrogate key value generated in the Issues table, instead of generating a new value for themselves.

Given a specific comment, you can retrieve the referenced bug or feature request using a relatively simple query. You don't have to include the Issues table in that query at all, unless you defined attribute columns in that table. Also, since the primary key value of the Bugs table and its ancestor Issues table are the same, you can join Bugs directly to Comments. You can join two tables even if there is no foreign key constraint linking them directly, as long as you use columns that represent comparable information in your database.

Download Polymorphic/soln/super-join.sql

```
SELECT *
FROM Comments AS c
  LEFT OUTER JOIN Bugs AS b USING (issue_id)
  LEFT OUTER JOIN FeatureRequests AS f USING (issue_id)
WHERE c.comment_id = 9876;
```

Given a specific bug, you can retrieve its comments just as easily.

Download Polymorphic/soln/super-join.sql

```
SELECT *
FROM Bugs AS b
  JOIN Comments AS c USING (issue_id)
WHERE b.issue_id = 1234;
```

The point is that if you use an ancestor table like Issues, you can rely on the enforcement of your database's data integrity by foreign keys.

---

*In every table relationship, there is one referencing table
and one referenced table.*

Chapter 8

Multicolumn Attributes

I can't count the number of times I have created a table to store people's contact information. Always this kind of table has commonplace columns such as the person's name, salutation, address, and probably company name.

Phone numbers are a little trickier. People use multiple numbers: a home number, a work number, a fax number, and a mobile number are common. In the contact information table, it's easy to store these in four columns.

But what about additional numbers? The person's assistant, second mobile phone, or field office have distinct phone numbers, and there could be other unforeseen categories. I could create more columns for the less common cases, but that seems clumsy because it adds seldom-used fields to data entry forms. How many columns is enough?

8.1 Objective: Store Multivalue Attributes

This is the same objective as in Chapter 2, *Jaywalking*, on page 25: an attribute seems to belong in one table, but the attribute has multiple values. Previously, we saw that combining multiple values into a comma-separated string makes it hard to validate the values, hard to read or change individual values, and hard to compute aggregate expressions such as counting the number of distinct values.

We'll use a new example to illustrate this antipattern. We want the bugs database to allow *tags* so we can categorize bugs. Some bugs may be categorized by the software subsystem that they affect, for instance *printing*, *reports*, or *email*. Other bugs may be categorized by the nature

of the defect; for instance, a crash bug could be tagged *crash*, while you could tag a report of slowness with *performance*, and you could tag a bad color choice in the user interface with *cosmetic*.

The bug-tagging feature must support multiple tags, because tags are not necessarily mutually exclusive. A defect could affect multiple systems or could affect the performance of printing.

8.2 Antipattern: Create Multiple Columns

We still have to account for multiple values in the attribute, but we know the new solution must store only a single value in each column. It might seem natural to create multiple columns in this table, each containing a single tag.

Download **Multi-Column/anti/create-table.sql**

```
CREATE TABLE Bugs (
  bug_id      SERIAL PRIMARY KEY,
  description VARCHAR(1000),
  tag1        VARCHAR(20),
  tag2        VARCHAR(20),
  tag3        VARCHAR(20)
);
```

As you assign tags to a given bug, you'd put values in one of these three columns. Unused columns remain null.

Download **Multi-Column/anti/update.sql**

```
UPDATE Bugs SET tag2 = 'performance' WHERE bug_id = 3456;
```

bug_id	description	tag1	tag2	tag3
1234	Crashes while saving	crash	NULL	NULL
3456	Increase performance	printing	performance	NULL
5678	Support XML	NULL	NULL	NULL

Most tasks you could do easily with a conventional attribute now become more complex.

Searching for Values

When searching for bugs with a given tag, you must search all three columns, because the tag string could occupy any of these columns.

For example, to retrieve bugs that reference *performance*, use a query like the following:

Download **Multi-Column/anti/search.sql**

```sql
SELECT * FROM Bugs
WHERE tag1 = 'performance'
   OR tag2 = 'performance'
   OR tag3 = 'performance';
```

You might need to search for bugs that reference both tags, *performance* and *printing*. To do this, use a query like the following one. Remember to use parentheses correctly, because OR has lower precedence than AND.

Download **Multi-Column/anti/search-two-tags.sql**

```sql
SELECT * FROM Bugs
WHERE (tag1 = 'performance' OR tag2 = 'performance' OR tag3 = 'performance')
  AND (tag1 = 'printing' OR tag2 = 'printing' OR tag3 = 'printing');
```

The syntax required to search for a single value over multiple columns is lengthy and tedious to write. You can make it more compact by using an IN predicate in a slightly untraditional manner:

Download **Multi-Column/anti/search-two-tags.sql**

```sql
SELECT * FROM Bugs
WHERE 'performance' IN (tag1, tag2, tag3)
  AND 'printing'    IN (tag1, tag2, tag3);
```

Adding and Removing Values

Adding and removing a value from the set of columns presents its own issues. Simply using UPDATE to change one of the columns isn't safe, since you can't be sure which column is unoccupied, if any. You might have to retrieve the row into your application to see.

Download **Multi-Column/anti/add-tag-two-step.sql**

```sql
SELECT * FROM Bugs WHERE bug_id = 3456;
```

In this case, for instance, the result shows you that tag2 is null. Then you can form the UPDATE statement.

Download **Multi-Column/anti/add-tag-two-step.sql**

```sql
UPDATE Bugs SET tag2 = 'performance' WHERE bug_id = 3456;
```

You face the risk that in the moment after you query the table and before you update it, another client has gone through the same steps of reading the row and updating it. Depending on who applied their update first, either you or he risks getting an update conflict error or

having his changes overwritten by the other. You can avoid this two-step query by using complex SQL expressions.

The following statement uses the NULLIF() function to make each column null if it equals a specific value. NULLIF() returns null if its two arguments are equal.[1]

Download **Multi-Column/anti/remove-tag.sql**

```sql
UPDATE Bugs
SET tag1 = NULLIF(tag1, 'performance'),
    tag2 = NULLIF(tag2, 'performance'),
    tag3 = NULLIF(tag3, 'performance')
WHERE bug_id = 3456;
```

The following statement adds the new tag *performance* to the first column that is currently null. However, if none of the three columns is null, then the statement makes no change to the row, and the new tag value is not recorded at all. Also, constructing this statement is laborious. Notice you must repeat the string *performance* six times.

Download **Multi-Column/anti/add-tag.sql**

```sql
UPDATE Bugs
SET tag1 = CASE
       WHEN 'performance' IN (tag2, tag3) THEN tag1
       ELSE COALESCE(tag1, 'performance') END,
    tag2 = CASE
       WHEN 'performance' IN (tag1, tag3) THEN tag2
       ELSE COALESCE(tag2, 'performance') END,
    tag3 = CASE
       WHEN 'performance' IN (tag1, tag2) THEN tag3
       ELSE COALESCE(tag3, 'performance') END
WHERE bug_id = 3456;
```

Ensuring Uniqueness

You probably don't want the same value to appear in multiple columns, but when you use the Multicolumn Attributes antipattern, the database can't prevent this. In other words, it's hard to prevent the following statement:

Download **Multi-Column/anti/insert-duplicate.sql**

```sql
INSERT INTO Bugs (description, tag1, tag2, tag3)
  VALUES ('printing is slow', 'printing', 'performance', 'performance');
```

---

1. The NULLIF() is a standard function in SQL; it's supported by all brands except Informix and Ingres.

Handling Growing Sets of Values

Another weakness of this design is that three columns might not be enough. To keep the design of one value per column, you must define as many columns as the maximum number of tags a bug can have. How can you predict, at the time you define the table, what that greatest number will be?

One tactic is to guess at a moderate number of columns and expand later, if necessary, by adding more columns. Most databases allow you to restructure existing tables, so you can add Bugs.tag4, or even more columns, as you need them.

Download **Multi-Column/anti/alter-table.sql**

```sql
ALTER TABLE Bugs ADD COLUMN tag4 VARCHAR(20);
```

However, this change is costly in three ways:

- Restructuring a database table that already contains data may require locking the entire table, blocking access for other concurrent clients.

- Some databases implement this kind of table restructure by defining a new table to match the desired structure, copying the data from the old table, and then dropping the old table. If the table in question has a lot of data, this transfer can take a long time.

- When you add a column in the set for a multicolumn attribute, you must revisit every SQL statement in every application that uses this table, editing the statement to support new columns.

  Download **Multi-Column/anti/search-four-columns.sql**

  ```sql
  SELECT * FROM Bugs
  WHERE tag1 = 'performance'
    OR tag2 = 'performance'
    OR tag3 = 'performance'
    OR tag4 = 'performance'; -- you must add this new term
  ```

  This is a meticulous and time-consuming development task. If you miss any queries that need edits, it can lead to bugs that are difficult to detect.

8.3   How to Recognize the Antipattern

If the user interface or documentation for your project describes an attribute to which you can assign multiple values but is limited to a

> **Patterns Among Antipatterns**
>
> The Jaywalking and Multicolumn Attributes antipatterns have a common thread: these two antipatterns are both solutions for the same objective: to store an attribute that may have multiple values.
>
> In the examples for Jaywalking, we saw how that antipattern relates to many-to-many relationships. In this chapter, we see a simpler one-to-many relationship. Be aware that both antipatterns are sometimes used for both types of relationships.

fixed maximum number of values, this might indicate that the Multicolumn Attributes antipattern is in use.

Admittedly, some attributes might have a limit on the number of selections on purpose, but it's more common that there's no such limit. If the limit seems arbitrary or unjustified, it might be because of this antipattern.

Another clue that the antipattern might be in use is if you hear statements such as the following:

- "How many is the greatest number of tags we need to support?"

  You need to decide how many columns to define in the table for a multivalue attribute like tag.

- "How can I search multiple columns at the same time in SQL?"

  If you're searching for a given value across multiple columns, this is a clue that the multiple columns should really be stored as a single logical attribute.

8.4 Legitimate Uses of the Antipattern

In some cases, an attribute may have a fixed number of choices, and the position or order of these choices may be significant. For example, a given bug may be associated with several users' accounts, but the nature of each association is unique. One is the user who reported the bug, another is a programmer assigned to fix the bug, and another is the quality control engineer assigned to verify the fix. Even though the

values in each of these columns are compatible, their significance and usage actually makes them logically different attributes.

It would be valid to define three ordinary columns in the Bugs table to store each of these three attributes. The drawbacks described in this chapter aren't as important, because you are more likely to use them separately. Sometimes you might still need to query over all three columns, for instance to report everyone involved with a given bug. But you can accept this complexity for a few cases in exchange for greater simplicity in most other cases.

Another way to structure this is to create a dependent table for multiple associations from the Bugs table the Accounts table and give this new table an extra column to note the role each account has in relation to that bug. However, this structure might lead to some of the problems described in Chapter 6, *Entity-Attribute-Value*, on page 73.

8.5 Solution: Create Dependent Table

As we saw in Chapter 2, *Jaywalking*, on page 25, the best solution is to create a dependent table with one column for the multivalue attribute. Store the multiple values in multiple rows instead of multiple columns. Also, define a foreign key in the dependent table to associate the values to its parent row in the Bugs table.

Download Multi-Column/soln/create-table.sql

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

INSERT INTO Tags (bug_id, tag)
  VALUES (1234, 'crash'), (3456, 'printing'), (3456, 'performance');
```

When all the tags associated with a bug are in a single column, searching for bugs with a given tag is more straightforward.

Download Multi-Column/soln/search.sql

```sql
SELECT * FROM Bugs JOIN Tags USING (bug_id)
WHERE tag = 'performance';
```

Even more complex searches, such as a bug that relates to two specific tags, is easy to read.

```sql
SELECT * FROM Bugs
  JOIN Tags AS t1 USING (bug_id)
  JOIN Tags AS t2 USING (bug_id)
WHERE t1.tag = 'printing' AND t2.tag = 'performance';
```

You can add or remove an association much more easily than with the Multicolumn Attributes antipattern. Simply insert or delete a row from the dependent table. There's no need to inspect multiple columns to see where you can add a value.

```sql
INSERT INTO Tags (bug_id, tag) VALUES (1234, 'save');

DELETE FROM Tags WHERE bug_id = 1234 AND tag = 'crash';
```

The PRIMARY KEY constraint ensures that no duplication is allowed. A given tag can be applied to a given bug only once. If you attempt to insert a duplicate, SQL returns a duplicate key error.

You are not limited to three tags per bug, as you were when there were only three tag*N* columns in the Bugs table. Now you can apply as many tags per bug as you need.

Store each value with the same meaning in a single column.

Chapter 9

Metadata Tribbles

My wife worked for years as a programmer in Oracle PL/SQL and Java. She described a case that showed how a database design that was intended to simplify work instead created more work.

A table Customers used by the Sales division at her company kept data such as customers' contact information, their business type, and how much revenue had been received from that customer:

Download Metadata-Tribbles/intro/create-table.sql

```sql
CREATE TABLE Customers (
  customer_id   NUMBER(9) PRIMARY KEY,
  contact_info  VARCHAR(255),
  business_type VARCHAR(20),
  revenue       NUMBER(9,2)
);
```

But the Sales division needed to break down the revenue by year so they could track recently active customers. They decided to add a series of new columns, each column's name indicating the year it covered:

Download Metadata-Tribbles/intro/alter-table.sql

```sql
ALTER TABLE Customers ADD (revenue2002 NUMBER(9,2));
ALTER TABLE Customers ADD (revenue2003 NUMBER(9,2));
ALTER TABLE Customers ADD (revenue2004 NUMBER(9,2));
```

Then they entered incomplete data, only for customers they thought were interesting to track. On most rows, they left null in those revenue columns. The programmers started wondering whether they could store other information in these mostly unused columns.

Each year, they needed to add one more column. A database administrator was responsible for managing Oracle's tablespaces. So each year, they had to have a series of meetings, schedule a data migration

to restructure the tablespace, and add the new column. Ultimately they wasted a lot of time and money.

9.1   Objective: Support Scalability

Performance degrades for any database query as the volume of data goes up. Even if a query returns results promptly with a few thousand rows, the tables naturally accumulate data to the point where the same query may not have acceptable performance. Using indexes intelligently helps, but nevertheless the tables grow, and this affects the speed of queries against them.

The objective is to structure a database to improve the performance of queries and support tables that grow steadily.

9.2   Antipattern: Clone Tables or Columns

In the television series *Star Trek*,[1] "tribbles" are small furry animals kept as pets. Tribbles are very appealing at first, but soon they reveal their tendency to reproduce out of control, and managing the overpopulation of tribbles becomes a serious problem.

Where do you put them? Who's responsible for them? How long would it take to pick up every tribble? Eventually, Captain Kirk discovers that his ship and crew can't function, and he has to order his crew to make it top priority to remove the tribbles.

We know from experience that querying a table with few rows is quicker than querying a table with many rows, all other things being equal. This leads to a common fallacy that we must make every table contain fewer rows, no matter what we have to do. This leads to two forms of the antipattern:

- Split a single long table into multiple smaller tables, using table names based on distinct data values in one of the table's attributes.

- Split a single column into multiple columns, using column names based on distinct values in another attribute.

But you can't get something for nothing; to meet the goal of having few rows in every table, you have to either create tables that have too many

---

1.   "Star Trek" and related marks are trademarks of CBS Studios Inc.

> ### Mixing Metadata with Data
>
> Notice that by appending the year onto the base table name, we've combined a data value with a metadata identifier.
>
> This is the reverse of *mixing data with metadata* that we saw earlier in the Entity-Attribute-Value and Polymorphic Associations antipatterns. In those cases, we stored metadata identifiers (a column name and table name) as string data.
>
> In Multicolumn Attributes and Metadata Tribbles, we're making a data value into a column name or a table name. If you use any of these antipatterns, you create more problems than you solve.

columns or else create a greater number of tables. In both cases, you find that the number of tables or columns continues to grow, since new data values can make you create new schema objects.

Spawning Tables

To split data into separate tables, you'd need some policy for which rows belong in which tables. For example, you could split them up by the year in the date_reported column:

Download **Metadata-Tribbles/anti/create-tables.sql**

```sql
CREATE TABLE Bugs_2008 ( . . . );
CREATE TABLE Bugs_2009 ( . . . );
CREATE TABLE Bugs_2010 ( . . . );
```

As you insert rows into the database, it's your responsibility to use the correct table, depending on the values you insert:

Download **Metadata-Tribbles/anti/insert.sql**

```sql
INSERT INTO Bugs_2010 (..., date_reported, ...) VALUES (..., '2010-06-01', ...);
```

Fast forward to January 1 of the next year. Your application starts getting an error from all new bug reports, because you didn't remember to create the Bugs_2011 table.

Download **Metadata-Tribbles/anti/insert.sql**

```sql
INSERT INTO Bugs_2011 (..., date_reported, ...) VALUES (..., '2011-02-20', ...);
```

This means that introducing a new *data* value can cause a need for a new *metadata* object. This is not usually the relationship between data and metadata in SQL.

Managing Data Integrity

Suppose your boss is trying to count bugs reported during the year, but his numbers don't adding up. After investigating, you discover that some 2010 bugs were entered in the Bugs_2009 table by mistake. The following query should always return an empty result, and if it doesn't, you have a problem:

Download Metadata-Tribbles/anti/data-integrity.sql

```sql
SELECT * FROM Bugs_2009
WHERE date_reported NOT BETWEEN '2009-01-01' AND '2009-12-31';
```

There's no way to limit the data relative to the name of its table automatically, but you can declare a CHECK constraint in each of your tables:

Download Metadata-Tribbles/anti/check-constraint.sql

```sql
CREATE TABLE Bugs_2009 (
  -- other columns
  date_reported DATE CHECK (EXTRACT(YEAR FROM date_reported) = 2009)
);

CREATE TABLE Bugs_2010 (
  -- other columns
  date_reported DATE CHECK (EXTRACT(YEAR FROM date_reported) = 2010)
);
```

Remember to adjust the value in the CHECK constraint when you create Bugs_2011. If you make a mistake, you could create a table that rejects the rows it's supposed to accept.

Synchronizing Data

One day, your customer support analyst asks to change a bug report date. It's in the database as reported on 2010-01-03, but the customer who reported it actually sent it in by fax a week earlier, on 2009-12-27. You could change the date with a simple UPDATE:

Download Metadata-Tribbles/anti/anomaly.sql

```sql
UPDATE Bugs_2010
SET date_reported = '2009-12-27'
WHERE bug_id = 1234;
```

But this correction makes the row an invalid entry in the Bugs_2010 table. You would need to remove the row from one table and insert it into the other table, in the infrequent case that a simple UPDATE would cause this anomaly.

Download Metadata-Tribbles/anti/synchronize.sql

```
INSERT INTO Bugs_2009 (bug_id, date_reported, ...)
  SELECT bug_id, date_reported, ...
  FROM Bugs_2010
  WHERE bug_id = 1234;

DELETE FROM Bugs_2010 WHERE bug_id = 1234;
```

Ensuring Uniqueness

You should make sure that the primary key values are unique across all the split tables. If you need to move a row from one table to another, you need some assurance that the primary key value doesn't conflict with another row.

If you use a database that supports sequence objects, you can use a single sequence to generate values for all the split tables. For databases that support only per-table ID uniqueness, this may be more awkward. You have to define one extra table solely to produce primary key values:

Download Metadata-Tribbles/anti/id-generator.sql

```
CREATE TABLE BugsIdGenerator (bug_id SERIAL PRIMARY KEY);

INSERT INTO BugsIdGenerator (bug_id) VALUES (DEFAULT);
ROLLBACK;

INSERT INTO Bugs_2010 (bug_id, . . .)
  VALUES (LAST_INSERT_ID(), . . .);
```

Querying Across Tables

Inevitably, your boss needs a query that references multiple tables. For example, he may ask for a count of all open bugs regardless of the year they were created. You can reconstruct the full set of bugs using a UNION of all the split tables and query that as a derived table:

Download Metadata-Tribbles/anti/union.sql

```
SELECT b.status, COUNT(*) AS count_per_status FROM (
  SELECT * FROM Bugs_2008
    UNION
  SELECT * FROM Bugs_2009
    UNION
  SELECT * FROM Bugs_2010 ) AS b
GROUP BY b.status;
```

As the years go on and you create more tables such as Bugs_2011, you need to keep your application code up-to-date to reference the newly created tables.

Synchronizing Metadata

Your boss tells you to add a column to track the hours of work required to resolve each bug.

Download **Metadata-Tribbles/anti/alter-table.sql**

```sql
ALTER TABLE Bugs_2010 ADD COLUMN hours NUMERIC(9,2);
```

If you've split the table, then the new column applies only to the one table you alter. None of the other tables contains the new column.

If you use a UNION query across your split tables as in the previous section, you stumble upon a new problem: you can combine tables using UNION if they have the same columns. If they differ, then you have to name only the columns that all tables have in common, without using the * wildcard.

Managing Referential Integrity

If a dependent table like Comments references Bugs, the dependent table cannot declare a foreign key. A foreign key must specify a single table, but in this case the parent table is split into many.

Download **Metadata-Tribbles/anti/foreign-key.sql**

```sql
CREATE TABLE Comments (
  comment_id       SERIAL PRIMARY KEY,
  bug_id           BIGINT UNSIGNED NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs_????(bug_id)
);
```

The split table may also have problems being a dependent instead of a parent. For example, Bugs.reported_by references the Accounts table. If you want to query all bugs reported by a given person regardless of the year, you need a query like the following:

Download **Metadata-Tribbles/anti/join-union.sql**

```sql
SELECT * FROM Accounts a
JOIN (
    SELECT * FROM Bugs_2008
    UNION ALL
    SELECT * FROM Bugs_2009
    UNION ALL
    SELECT * FROM Bugs_2010
  ) t ON (a.account_id = t.reported_by)
```

Identifying Metadata Tribbles Columns

Columns can be Metadata Tribbles, too. You can create a table containing columns that are bound to propagate by their nature, as we saw in the story at the beginning of this chapter.

Another example we might have in our bugs database is a table that records summary data for project metrics, where individual columns store subtotals. For instance, in the following table, it's only a matter of time before you need to add the column bugs_fixed_2011:

Download **Metadata-Tribbles/anti/multi-column.sql**

```sql
CREATE TABLE ProjectHistory (
  bugs_fixed_2008  INT,
  bugs_fixed_2009  INT,
  bugs_fixed_2010  INT
);
```

9.3 How to Recognize the Antipattern

The following phrases may indicate that the Metadata Tribbles antipattern is growing in your database:

- "Then we need to create a table (or column) per . . . "

  When you describe your database with phrases using *per* in this way, you're splitting tables by distinct values in one of the columns.

- "What's the maximum number of tables (or columns) that the database supports?"

  Most brands of database can handle many more tables and columns than you would need, if you used a sensible database design. If you think you might exceed the maximum, it's a strong sign that you need to rethink your design.

- "We found out why the application failed to add new data this morning: we forgot to create a new table for the new year."

  This is a common consequence of Metadata Tribbles. When new data demands new database objects, you need to define those objects proactively or else risk unforeseen failures.

- "How do I run a query to search many tables at once? All the tables have the same columns."

If you need to search many tables with identical structure, you should have stored them together in a single table, with one extra attribute column to distinguish the rows.

• "How do I pass a parameter for a table name? I need to query a table name appended with the year number dynamically."

You wouldn't need to do this if your data were in one table.

9.4 Legitimate Uses of the Antipattern

One good use of manually splitting tables is for *archiving*—removing historical data from day-to-day use. Often the need to run queries against historical data is greatly reduced after the data is no longer current.

If you have no need to query current data and historical data together, it's appropriate to copy the older data to another location and delete it from the active tables. Archiving keeps the data in a compatible table structure for occasional analysis but allows queries against current data to run with greater performance.

Sharding Databases at WordPress.com

At the MySQL Conference & Expo 2009, I had lunch with Barry Abrahamson, database architect for WordPress.com, a popular hosting service for blogging software.

Barry said when he started out hosting blogs, he hosted all his customers together in a single database. The content of a single blog site really wasn't that much, after all. It stood to reason that a single database is more manageable.

This did work well for the site initially, but it soon grew to very large-scale operations. Now it hosts 7 million blogs on 300 database servers. Each server hosts a subset of their customers.

When Barry adds a server, it would be very hard to separate data within a single database that belongs to an individual customer's blog. By splitting the data into a separate database per customer, he made it much easier to move any individual blog from one server to another. As customers come and go and some customers' blogs are busy while others go stale, his job to rebalance the load over multiple servers becomes even more important.

It's easier to back up and restore individual databases of moderate size than a single database containing terabytes of data. For example, if a customer calls and says their data got SNAFU'd because of bad data

entry, how would Barry restore one customer's data if all the customers share a single, monolithic database backup?

Although it seems like the right thing to do from a data modeling perspective to keep everything in a single database, splitting the database sensibly makes database administration tasks easier after the database size passes a certain threshold.

9.5 Solution: Partition and Normalize

There are better ways to improve performance if a table gets too large, instead of splitting the table manually. These include horizontal partitioning, vertical partitioning, and using dependent tables.

Using Horizontal Partitioning

You can gain the benefits of splitting a large table without the drawbacks by using a feature that is called either *horizontal partitioning* or *sharding*. You define a logical table with some rule for separating rows into individual partitions, and the database manages the rest. Physically, the table is split, but you can still execute SQL statements against the table as though it were whole.

You have flexibility in that you can define the way each individual table splits its rows into separate storage. For example, using the partitioning support in MySQL version 5.1, you can specify partitions as an optional part of a CREATE TABLE statement.

Download Metadata-Tribbles/soln/horiz-partition.sql

```
CREATE TABLE Bugs (
  bug_id SERIAL PRIMARY KEY,
  -- other columns
  date_reported DATE
) PARTITION BY HASH ( YEAR(date_reported) )
  PARTITIONS 4;
```

The previous example achieves a partitioning similar to that which we saw earlier in this chapter, separating rows based on the year in the date_reported column. However, its advantages over splitting the table manually are that rows are never placed in the wrong split table, even if the value of date_reported column is updated, and you can run queries against the Bugs table without the need to reference individual split tables.

The number of separate physical tables used to store the rows is fixed at four in this example. When you have rows spanning more than four years, one of the partitions will be used to store more than one year's worth of data. This will continue as the years go on. You don't need to add new partitions unless the volume of data becomes so great that you feel the need to split it further.

Partitioning is not defined in the SQL standard, so each brand of database implements it in their own nonstandard way. The terminology, syntax, and specific features of partitioning vary between brands. Nevertheless, some form of partitioning is now supported by every major brand of database.

Using Vertical Partitioning

Whereas horizontal partitioning splits a table by rows, vertical partitioning splits a table by columns. Splitting a table by columns can have advantages when some columns are bulky or seldom needed.

BLOB and TEXT columns have variable size, and they may be very large. For efficiency of both storage and retrieval, many database brands automatically store columns with these data types separately from the other columns of a given row. If you run a query without referencing any BLOB or TEXT columns of a table, you can access the other columns more efficiently. But if you use the column wildcard * in your query, the database retrieves all columns from that table, including any BLOB or TEXT columns.

For example, in the Products table of our bugs database, we might store a copy of the installation file for the respective product. This file is typically a self-extracting archive with an extension such as .exe on Windows or .dmg on a Mac. The files are usually very large, but a BLOB column can store binary data of enormous size.

Logically, the installer file should be an attribute of the Products table. But in most queries against that table, you wouldn't need the installer. Storing such a large volume of data in the Products table, which you use infrequently, could lead to inadvertent performance problems if you're in the habit of retrieving all columns using the * wildcard.

The remedy is to store the BLOB column in another table, separate from but dependent on the Products table. Make its primary key also serve as a foreign key to the Products table to ensure there is at most one row per product row.

```sql
CREATE TABLE ProductInstallers (
  product_id      BIGINT UNSIGNED PRIMARY KEY,
  installer_image BLOB,
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

The previous example is extreme to make the point, but it shows the benefit of storing some columns in a separate table. For example, in MySQL's MyISAM storage engine, querying a table is most efficient when the rows are of fixed size. VARCHAR is a variable-length data type, so the presence of a single column with that data type in a table prevents the table from gaining that advantage. If you store all variable-length columns in a separate table, then queries against the primary table can benefit (if even a little bit).

```sql
CREATE TABLE Bugs (
  bug_id        SERIAL PRIMARY KEY, -- fixed length data type
  summary       CHAR(80),           -- fixed length data type
  date_reported DATE,               -- fixed length data type
  reported_by   BIGINT UNSIGNED,    -- fixed length data type
  FOREIGN KEY  (reported_by) REFERENCES Accounts(account_id)
);

CREATE TABLE BugDescriptions (
  bug_id       BIGINT UNSIGNED PRIMARY KEY,
  description  VARCHAR(1000),      -- variable length data type
  resolution   VARCHAR(1000)       -- variable length data type
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

Fixing Metadata Tribbles Columns

Similar to the solution we saw in Chapter 8, *Multicolumn Attributes*, on page 102, the remedy for Metadata Tribbles columns is to create a dependent table.

```sql
CREATE TABLE ProjectHistory (
  project_id  BIGINT,
  year        SMALLINT,
  bugs_fixed  INT,
  PRIMARY KEY (project_id, year),
  FOREIGN KEY (project_id) REFERENCES Projects(project_id)
);
```

Instead of one row per project with multiple columns for each year, use multiple rows, with one column for bugs fixed. If you define the table in this way, you don't need to add new columns to support subsequent years. You can store any number of rows per project in this table as time goes on.

Don't let data spawn metadata.

Part II

Physical Database Design Antipatterns

10.0 times 0.1 is hardly ever 1.0.
▶ Brian Kernighan

Chapter 10

Rounding Errors

Your boss asks you to produce a report of the cost of programmer time for the project, based on the total work to fix each bug. Each programmer in the Accounts table has a different hourly rate, so you record the number of hours required to fix each bug in the Bugs table, and you multiply it by the hourly_rate of the programmer assigned to do the work.

Download *Rounding-Errors/intro/cost-per-bug.sql*

```sql
SELECT b.bug_id, b.hours * a.hourly_rate AS cost_per_bug
FROM Bugs AS b
  JOIN Accounts AS a ON (b.assigned_to = a.account_id);
```

To support this query, you need to create new columns in the Bugs and Accounts tables. Both columns should support fractional values, because you need to track the costs precisely. You decide to define the new columns as FLOAT, because this data type supports fractional values.

Download *Rounding-Errors/intro/float-columns.sql*

```sql
ALTER TABLE Bugs ADD COLUMN hours FLOAT;

ALTER TABLE Accounts ADD COLUMN hourly_rate FLOAT;
```

You update the columns with information from the bug work logs and the programmers' rates, test the report, and call it a day.

The next day, your boss shows up in your office with a copy of the project cost report. "These numbers don't add up," he tells you through gritted teeth. "I did the calculation by hand for comparison, and your report is inaccurate—slightly, by only a few dollars. How do you explain this?" You start to perspire. What could have gone wrong with such a simple calculation?

10.1   Objective: Use Fractional Numbers Instead of Integers

The integer is a useful data type, but it stores only whole numbers like 1 or 327 or -19. It can't represent fractional values like 2.5. You need a different data type if you need numbers with more precision than an integer. For example, sums of money are usually represented by numbers with two decimal places, like $19.95.

So, the objective is to store numeric values that aren't whole numbers and to use them in arithmetic computations. There is an additional objective, although it ought to go without saying: the results of arithmetic computations must be *correct*.

10.2   Antipattern: Use FLOAT Data Type

Most programming languages support a data type for real numbers, called float or double. SQL supports a similar data type of the same name. Many programmers naturally use the SQL FLOAT data type everywhere they need fractional numeric data, because they are accustomed to programming with the float data type.

The FLOAT data type in SQL, like float in most programming languages, encodes a real number in a binary format according to the IEEE 754 standard. You need to understand some characteristics of floating-point numbers in this format to use them effectively.

Rounding by Necessity

Many programmers are not aware of a characteristic of this floating-point format: not all values you can describe in decimal can be stored in binary. Out of necessity, some numbers must be rounded to a value that is very close.

To give some context for this rounding behavior, compare with rational numbers such as one-third, represented by a repeating decimal number like 0.333.... The true value cannot be represented in decimal, because you would need to write an infinite number of digits. The number of digits is the precision of the number, so a repeating decimal number would require *infinite precision*.

The compromise is to use *finite precision*, choosing a numeric value as close as possible to the original value, for example 0.333. However, this means that the value isn't exactly the same number we intended.

$$\frac{1}{3} + \frac{1}{3} + \frac{1}{3} \qquad = 1.000$$
$$0.333 + 0.333 + 0.333 \quad = 0.999$$

Even if we increase the precision, we still can't add three of these approximations of one-third to get a true value of 1.0. This is the necessary compromise of using finite precision to represent numbers that may have repeating decimals.

$$\frac{1}{3} + \frac{1}{3} + \frac{1}{3} \qquad = 1.000000$$
$$0.333333 + 0.333333 + 0.333333 \quad = 0.999999$$

This means some legitimate numbers that you can imagine cannot be represented with finite precision. You might think this is OK, because you can't really type a number with infinite digits anyway, so naturally any number you can type has finite precision and should be stored precisely, right? Unfortunately not.

IEEE 754 represents floating-point numbers in a base-2 format. The values that require infinite precision in binary are different values from those that behave this way in decimal. Some values that only need finite precision in decimal, for instance 59.95, require infinite precision to be represented exactly in binary. The FLOAT data type can't do this, so it uses the closest value in base-2 it can store, which is equal to 59.950000762939 in base-10.

Some values coincidentally use finite precision in both formats. In theory, if you understand the details of storing numbers in the IEEE 754 format, you can predict how a given decimal value is represented in binary. But in practice, most people won't do this computation for every floating-point value they use. You can't guarantee that a FLOAT column in the database will be given only values that are cooperative, so your application should assume that any value in this column may have been rounded.

Some databases support related data types called DOUBLE PRECISION and REAL. The precision that these data types and FLOAT support varies by database implementation, but they all represent floating-point values with a finite number of binary digits, so they all have similar rounding behavior.

Using FLOAT in SQL

Some databases can compensate for the inexactness and display the intended value.

Meet the IEEE 754 Format

The proposals for a standard binary format for floating-point numbers dates back to 1979. It was formally made a standard in 1985, and it is now widely implemented in software, most programming languages, and microprocessors.

The format has three fields to encode a floating-point value: a field for a *fraction* portion of the value, a field for the *exponent* to which to raise the fraction, and a single-bit *sign* field.

One advantage of IEEE 754 is that by using the exponent, it can represent fractional values that are both very small and very large. The format not only supports real numbers, but the range of values it supports is greater than integers in fixed-point format. The double-precision format supports an even greater range of values. So, these formats are useful for scientific applications.

But the most common use of fractional numeric values is probably to represent quantities of money. There's no need to use IEEE 754 for money, because the scaled decimal format described in this chapter can handle money values just as easily and more accurately.

Good references for learning more about this format are the Wikipedia article (http://en.wikipedia.org/wiki/IEEE_754-1985) or David Goldberg's article, "What Every Computer Scientist Should Know About Floating-Point Arithmetic." (Gol91)

Goldberg's article is also reprinted at http://www.validlab.com/goldberg/paper.pdf.

Download Rounding-Errors/anti/select-rate.sql

```sql
SELECT hourly_rate FROM Accounts WHERE account_id = 123;
```

Returns: 59.95

But the actual value stored in the FLOAT column may not be exactly this value. If you magnify the value by a billion, you see the discrepancy:

Download Rounding-Errors/anti/magnify-rate.sql

```sql
SELECT hourly_rate * 1000000000 FROM Accounts WHERE account_id = 123;
```

Returns: 59950000762.939

You might expect the magnified value returned by the previous query to be 59950000000.000. This shows that the value 59.95 has been rounded to a value that can be represented in the finite precision offered by the IEEE 754 binary format. In this case, the value is within 1 ten-millionth, which is close enough for many calculations.

However, it's not close enough for some other kinds of calculations to be accurate. One example is using a FLOAT in an equality comparison.

Download **Rounding-Errors/anti/inexact.sql**

```sql
SELECT * FROM Accounts WHERE hourly_rate = 59.95;
```

Result: empty set; no rows match.

We saw before that the value stored in hourly_rate is actually slightly more than 59.95. So even though you assigned the value 59.95 to this column for account_id 123, now the row fails to match the previous query.

One common workaround for this issue is to treat floating-point values as "effectively equal" if they are close within a small threshold. Subtract one value from the other, and use SQL's absolute value function ABS() to strip the sign from the difference. If the result is zero, then the two values were exactly equal. If the result is small enough, then the two values can be treated as effectively equal. The following query succeeds in finding the row:

Download **Rounding-Errors/anti/threshold.sql**

```sql
SELECT * FROM Accounts WHERE ABS(hourly_rate - 59.95) < 0.000001;
```

However, the difference is still large enough that a comparison of finer precision fails:

Download **Rounding-Errors/anti/threshold.sql**

```sql
SELECT * FROM Accounts WHERE ABS(hourly_rate - 59.95) < 0.0000001;
```

The appropriate threshold varies, because the absolute difference between the base-10 value and the rounded base-2 value varies.

Another example of the inexact nature of FLOAT causing accuracy problems is when you calculate aggregates of many values. For example, if you use SUM() to add up the floating-point values in a column, the sum accumulates the discrepancy caused by rounding all the values.

Download **Rounding-Errors/anti/cumulative.sql**

```sql
SELECT SUM( b.hours * a.hourly_rate ) AS project_cost
FROM Bugs AS b
JOIN Accounts AS a ON (b.assigned_to = a.account_id);
```

The cumulative impact of inexact floating-point numbers is even more severe when calculating the aggregate product of a set of numbers instead of the sum. The difference seems small, but it compounds. For example, if you multiply the value 1 by a factor of exactly 1.0, the result is always 1. It doesn't matter how many times you apply this factor. However, if the factor is actually 0.999, this has a different result. If you multiply a value of one by 0.999 a thousand times in succession, you get a result of about 0.3677. The more times you multiply, the more the discrepancy grows.

A good example of applying a multiplication many times in succession is to calculate compounding interest in a financial application. Using inexact floating-point numbers introduces an error that seems tiny but grows as it compounds on itself. So, using exact values in financial applications is important.

10.3   How to Recognize the Antipattern

Virtually any use of FLOAT, REAL, or DOUBLE PRECISION data types is suspect. Most applications that use floating-point numbers don't require the range of values supported by IEEE 754 formats.

It seems natural to use FLOAT data types in SQL, because it shares its name with a data type found in most programming languages. But there is a better choice for the data type.

10.4   Legitimate Uses of the Antipattern

FLOAT is a good data type when you need real number values with a greater range than INTEGER or NUMERIC data types support. Scientific applications are often cited as the best use of a FLOAT.

Oracle uses the FLOAT data type to mean an exact scaled numeric, whereas the BINARY_FLOAT data type is an inexact numeric, using the IEEE 754 encoding.

10.5   Solution: Use NUMERIC Data Type

Instead of FLOAT or its siblings, use the NUMERIC or DECIMAL SQL data types for fixed-precision fractional numbers.

```
ALTER TABLE Bugs ADD COLUMN hours NUMERIC(9,2);

ALTER TABLE Accounts ADD COLUMN hourly_rate NUMERIC(9,2);
```

These data types store numeric values exactly, up to the *precision* you specify in the column definition. Specify precision as an argument to the data type, similar to the syntax you would use for the length of a VARCHAR data type. The precision is the total number of decimal digits you can use in a value in this column. A precision of 9 means that you can store a value like 123456789, but you may not be able to store 1234567890.[1]

You may also specify a *scale* in a second argument to the data type. The scale is the number of digits to the right of the decimal point. These digits are included in the precision digits, so a precision of 9 with a scale of 2 means you can store a value like 1234567.89, but not 12345678.91 or 123456.789.

The precision and scale you specify applies to the column on all rows in the table. In other words, you can't store values with scale 2 on some rows and scale 4 on other rows. It's ordinary in SQL that a column's data type applies uniformly on all rows (just as a column defined as VARCHAR(20) would allow a string of that length on every row).

The advantage of NUMERIC and DECIMAL are that they store rational numbers without rounding, as the FLOAT data types do. After you set a value to 59.95, you can depend on that value being stored exactly. When you compare it for equality to a literal value 59.95, the comparison succeeds.

```
SELECT hourly_rate FROM Accounts WHERE hourly_rate = 59.95;
```

Returns: 59.95

Likewise, if you scale up the value by a billion, you get the expected value:

```
SELECT hourly_rate * 1000000000 FROM Accounts WHERE hourly_rate = 59.95;
```

Returns: 59950000000

---

1. In some brands of database, the size of the column is rounded up internally to the nearest byte, word, or double word, so the maximum value of a NUMERIC column may have more digits than the precision you specified.

The data types NUMERIC and DECIMAL behave identically; there should be no difference between them. DEC is also a synonym for DECIMAL.

You still can't store values that require infinite precision, such as one-third. But at least we're more familiar with values that have this restriction in decimal format.

If you need exact decimal values, use the NUMERIC data type. The FLOAT data type is unable to represent many decimal rational numbers, so they should be treated as inexact values.

Do not use FLOAT if you can avoid it.

Science is feasible when the variables are few and can be enumerated; when their combinations are distinct and clear.

▶ Paul Valéry

Chapter 11

31 Flavors

In a personal contact information table, the *salutation* is a good example of a column that can have only a few values. Once you support *Mr.*, *Mrs.*, *Ms.*, *Dr.*, and *Rev.*, you've accounted for virtually everyone. You could specify this list in the column definition, using a data type or a constraint, so that no one can accidentally enter an invalid string into the salutation column.

Download 31-Flavors/intro/create-table.sql

```sql
CREATE TABLE PersonalContacts (
  -- other columns
  salutation VARCHAR(4)
    CHECK (salutation IN ('Mr.', 'Mrs.', 'Ms.', 'Dr.', 'Rev.')),
);
```

That should settle it, since there are no other salutations you need to support, right?

Unfortunately, your boss tells you that your company is opening a subsidiary in France. You need to support the salutations *M.*, *Mme.*, and *Mlle.* Your assignment is to alter your contact table to permit these values. This is a delicate job and may not be possible without interrupting availability of that table.

You also thought your boss mentioned that the company is trying to open an office next month in Brazil.

11.1 Objective: Restrict a Column to Specific Values

Restricting a column's values to a fixed set of values is very useful. If we can ensure that the column never contains an invalid entry, it can simplify use of that column.

> ### Baskin-Robbins 31 Ice Cream
>
> In 1953, this famous chain of ice cream parlors offered one flavor for each day of the month. The chain used the slogan *31 Flavors* for many years.
>
> Today, more than sixty years later, Baskin-Robbins offers twenty-one classic flavors, twelve seasonal flavors, sixteen regional flavors, as well as a variety of Bright Choices and Flavors of the Month. Even though its ice cream flavors were once an immutable set that defined its brand, Baskin-Robbins expanded its choices and made them configurable and variable.
>
> The same thing could happen in the project for which you're designing a database—in fact, you should count on it.

For example, in the Bugs table of our example database, the status column indicates whether a given bug is *NEW*, *IN PROGRESS*, *FIXED*, and so on. The significance of each of these status values depends on how we manage bugs in our project, but the point is that the data in the column must be one of these values.

Ideally, we need the database to reject invalid data:

Download 31-Flavors/obj/insert-invalid.sql

```sql
INSERT INTO Bugs (status) VALUES ('NEW'); -- OK

INSERT INTO Bugs (status) VALUES ('BANANA'); -- Error!
```

11.2   Antipattern: Specify Values in the Column Definition

Many people choose to specify the valid data values when they define the column. The column definition is part of the *metadata*—the definition of the table structure itself.

For example, you could define a *check constraint* on the column. This constraint disallows any insert or update that would make the constraint false.

Download 31-Flavors/anti/create-table-check.sql

```sql
CREATE TABLE Bugs (
  -- other columns
  status    VARCHAR(20) CHECK (status IN ('NEW', 'IN PROGRESS', 'FIXED'))
);
```

MySQL supports a nonstandard data type called ENUM that restricts the column to a specific set of values.

Download 31-Flavors/anti/create-table-enum.sql

```
CREATE TABLE Bugs (
  -- other columns
  status ENUM('NEW', 'IN PROGRESS', 'FIXED'),
);
```

In MySQL's implementation, you declare the values as strings, but internally the column is stored as the ordinal number of the string in the enumerated list. The storage is therefore compact, but when you sort a query by this column, the result is ordered by the ordinal value, not alphabetically by the string value. You may not expect this behavior.

Other solutions include *domains* and *user-defined types* (UDTs). You can use these to restrict a column to a specific set of values and conveniently apply the same domain or data type to several columns within your database. But these features are not supported widely among brands of RDBMSs yet.

Finally, you could write a trigger that contains the set of permitted values and raises an error unless the status matches one of these values.

All of these solutions share some disadvantages. The following sections describe some of these problems.

What Was the Middle One?

Suppose you're developing a user interface for the bug tracker so that a user can edit bug reports. To make the interface guide the user to pick one of the valid status values, you choose to fill a drop-down menu control with these values. How do you query the database for an enumerated list of values that are currently allowed in the status column?

Your first instinct might be to query all the values currently in use, with a simple query like the following one:

Download 31-Flavors/anti/distinct.sql

```
SELECT DISTINCT status FROM Bugs;
```

However, if all the bugs are new, the previous query returns only *NEW*. If you use this result to populate a user interface control for the status of bugs, you could create a chicken-and-egg situation; you can't change a bug to any status other than those currently in use.

To get the complete list of permitted status values, you need to query the definition of that column's metadata. Most SQL databases support *system views* for these kinds of queries, but using them can be complex. For example, if you used MySQL's ENUM data type, you can use the following query to query the INFORMATION_SCHEMA system views:

Download 31-Flavors/anti/information-schema.sql

```sql
SELECT column_type
FROM information_schema.columns
WHERE table_schema = 'bugtracker_schema'
  AND table_name = 'bugs'
  AND column_name = 'status';
```

You can't simply get the discrete enumeration values from the INFORMATION_SCHEMA in a conventional result set. Instead, you get a string containing the definition of the check constraint or ENUM data type. For example, the previous query in MySQL returns a column of type LONGTEXT, with the value *ENUM('NEW', 'IN PROGRESS', 'FIXED')*, including the parentheses, commas, and single quotes. You must write application code to parse this string and extract the individual quoted values before you can use them to populate a user interface control.

The queries needed to report check constraints, domains, or UDTs are progressively more complex. Most people choose the better part of valor and manually maintain a parallel list of values in application code. This is an easy way for bugs to affect your project as application data becomes out of sync with the database metadata.

Adding a New Flavor

The most common alterations are to add or remove one of the permitted values. There's no syntax to add or remove a value from an ENUM or check constraint; you can only redefine the column with a new set of values. The following is an example of adding *DUPLICATE* as one new status value in the MySQL ENUM:

Download 31-Flavors/anti/add-enum-value.sql

```sql
ALTER TABLE Bugs MODIFY COLUMN status
  ENUM('NEW', 'IN PROGRESS', 'FIXED', 'DUPLICATE');
```

You need to know that the previous definition of the column allowed *NEW*, *IN PROGRESS*, and *FIXED*. This leads you back to the difficulty of querying the current set of values as described earlier.

Some database brands can't change the definition of a column unless the table is empty. You might need to dump the contents of the table, redefine the table, and then import your saved data, making the table

inaccessible in the meantime. This work is common enough that it has a name: *ETL* for "extract, transform, and load." Other brands of database support restructuring a populated table with ALTER TABLE commands, but it can still be complex and expensive to perform these changes.

As a matter of policy, changing metadata—that is, changing the definition of tables and columns—should be infrequent and with attention to testing and quality assurance. If you need to change metadata to add or remove a value from an ENUM, then you either have to skip the appropriate testing or spend a lot of software engineering effort on short notice to make the change. Either way, these changes introduce risk and destabilize your project.

Old Flavors Never Die

If you make a value obsolete, you could upset historical data. For example, you change your quality control process to replace *FIXED* with two stages, *CODE COMPLETE* and *VERIFIED*:

Download 31-Flavors/anti/remove-enum-value.sql

```
ALTER TABLE Bugs MODIFY COLUMN status
    ENUM('NEW', 'IN PROGRESS', 'CODE COMPLETE', 'VERIFIED');
```

If you remove *FIXED* from the enumeration, what do you do with bugs whose status was *FIXED*? Should you advance all *FIXED* bugs to *VERIFIED*? Should you instead set obsolete values to null or a default value?

You may have to keep an obsolete value that old rows reference. But then how can you distinguish between obsolete values and exclude them from your user interface so that no one can set a bug's status to the obsolete value?

Portability Is Hard

Check constraints, domains, and UDTs are not supported uniformly among brands of SQL databases. The ENUM data type is a proprietary feature in MySQL. Each brand of database may have a different limit on the length of the list you can give in a column definition. Trigger languages vary as well. These variations make it hard to choose a solution if you need to support multiple brands of database.

11.3 How to Recognize the Antipattern

The problems with using ENUM or a check constraint arise when the set of values is not fixed. If you're considering using ENUM, first ask yourself

whether the set of values are expected to change or even whether they *might* change. If so, it's probably not a good time to employ an ENUM.

- "We have to take the database offline so we can add a new choice in one of our application's menus. It should take no more than thirty minutes, if all goes well."

  This is a sign that a set of values is baked into the definition of a column. You should never need to interrupt service for a change like this.

- "The status column can have one of the following values. We should not need to revise this list."

  Shouldn't need to are weasel words, and this says something quite different from *can't*.

- "The list of values in the application code got out of sync with the business rules in the database—again."

  This is a risk of maintaining information in two different places.

11.4 Legitimate Uses of the Antipattern

As we discussed, ENUM may cause fewer problems if the set of values is unchanging. It's still difficult to query the metadata for the set of values, but you can maintain a matching list of values in application code without getting out of sync.

ENUM is most likely to succeed when it would make no sense to alter the set of permitted values, such as when a column represents an either/or choice with two mutually exclusive values:*LEFT/RIGHT*, *ACTIVE/IN-ACTIVE*, *ON/OFF*, *INTERNAL/EXTERNAL*, and so on.

Check constraints can be used in many ways other than simply to implement an ENUM-like mechanism, such as checking that a time interval's start is less than its end.

11.5 Solution: Specify Values in Data

There's a better solution to restrict values in a column: create a *lookup table* with one row for each value you allow in the Bugs.status column. Then declare a foreign key constraint on Bugs.status referencing the new table.

Download 31-Flavors/soln/create-lookup-table.sql

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

INSERT INTO BugStatus (status) VALUES ('NEW'), ('IN PROGRESS'), ('FIXED');

CREATE TABLE Bugs (
  -- other columns
  status  VARCHAR(20),
  FOREIGN KEY (status) REFERENCES BugStatus(status)
    ON UPDATE CASCADE
);
```

When you insert or update a row in the Bugs table, you must use a status value that exists in the BugStatus table. This enforces the status values like the ENUM or a check constraint, but there are several ways this solution offers more flexibility.

Querying the Set of Values

The set of permitted values is now stored in data, not metadata as it was with the ENUM data type. You can query data values from a lookup table with SELECT, just like any other table. This makes it much easier to retrieve the set of values as a data set to present in your user interface. You can even sort the set of values the user can choose from.

Download 31-Flavors/soln/query-canonical-values.sql

```sql
SELECT status FROM BugStatus ORDER by status;
```

Updating the Values in the Lookup Table

When you use a lookup table, you can add a value to the set with an ordinary INSERT statement. You can make a change like this without interrupting access to the table. You don't need to redefine any columns, schedule downtime, or perform an ETL operation. You also don't need to know the current set of values in the lookup table to add or remove a value.

Download 31-Flavors/soln/insert-value.sql

```sql
INSERT INTO BugStatus (status) VALUES ('DUPLICATE');
```

You can also rename a value easily, if you declared the foreign key with the ON UPDATE CASCADE option.

Download 31-Flavors/soln/update-value.sql

```sql
UPDATE BugStatus SET status = 'INVALID' WHERE status = 'BOGUS';
```

Supporting Obsolete Values

You can't DELETE a row from the lookup table if it's referenced by a row in Bugs. The foreign key on the status column enforces referential integrity, so the value must exist in the lookup table.

However, you can add another attribute column to the lookup table to designate some values as obsolete. This allows you to maintain historical data in the Bugs.status column, while distinguishing between the obsolete values and values that are eligible to appear in your user interface.

Download 31-Flavors/soln/inactive.sql

```sql
ALTER TABLE BugStatus ADD COLUMN active
  ENUM('INACTIVE', 'ACTIVE') NOT NULL DEFAULT 'ACTIVE';
```

Use UPDATE instead of DELETE to make a value obsolete:

Download 31-Flavors/soln/update-inactive.sql

```sql
UPDATE BugStatus SET active = 'INACTIVE' WHERE status = 'DUPLICATE';
```

When you retrieve the set of values to show in a user interface for users to pick, restrict the query to status values that are *ACTIVE*:

Download 31-Flavors/soln/select-active.sql

```sql
SELECT status FROM BugStatus WHERE active = 'ACTIVE';
```

This gives you more flexibility than an ENUM or a check constraint, because those solutions don't support extra attributes per value.

Portability Is Easy

Unlike the ENUM data type, check constraints, or domains or UDTs, the lookup table solution relies only on the standard SQL feature of declarative referential integrity using foreign key constraints. This makes the solution more portable.

You can also keep a virtually unlimited number of values in your lookup table, since you store each value on a separate row.

Use metadata when validating against a fixed set of values.
Use data when validating against a fluid set of values.

Chapter 12

Phantom Files

Catastrophe strikes your database server. While relocating a rack full of hard drives, the rack tipped over and crashed. Fortunately, no one was hurt, but the massive hard drives shattered. Even the raised floor was broken where they fell. Fortunately, the IT department is prepared: they make good backups of every important system every day, and they quickly deploy a new server and restore your database.

It doesn't take long during smoke testing to notice a problem: your application associates graphic images with many database entities, but all the images are missing! You call the IT technician immediately.

"We restored the database and verified it's complete as of the last backup," the technician says. "Where were the images stored?"

You remember now that in this application, images are stored outside the database, and ordinary files are stored on the filesystem. The database stores the path to the image, and the application opens each image file at that path. "The images were stored as files. They were on the /var filesystem, same as the databases."

The technician shakes his head. "We don't back up files on the /var filesystem unless you specifically told us which ones. We back up any databases, of course, but other files on /var are usually just logs, cache data, or other temporary files. By default, they don't get backed up."

Your heart sinks. There were more than 11,000 images used in your product catalog database. Most of them probably exist in other places, but tracking them all down, reformatting them, and generating thumbnail versions for web searches will take weeks.

12.1 Objective: Store Images or Other Bulky Media

Images and other media are used in most applications these days. Sometimes media are associated with entities stored in the database. For example, you may allow a user to have a portrait or avatar that is displayed when he posts a comment. In our bugs database, bugs often need a screenshot to illustrate the circumstances of the defect.

The objective described in this chapter is to store images and associate them with database entities, such as user accounts or bugs. When we query these entities from the database, we need the capability to retrieve the associated images in the application.

12.2 Antipattern: Assume You Must Use Files

Conceptually, an image is an attribute in a table. For example, the Accounts table may have a portrait_image column.

Download Phantom-Files/anti/create-accounts.sql

```sql
CREATE TABLE Accounts (
  account_id      SERIAL PRIMARY KEY
  account_name    VARCHAR(20),
  portrait_image  BLOB
);
```

Likewise, you can store multiple images of the same type in a dependent table. For example, each bug may have multiple screenshots that illustrate the bug.

Download Phantom-Files/anti/create-screenshots.sql

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

That much is straightforward, but choosing the data type for an image is a subject of controversy. Raw binary data for an image can be stored in a BLOB data type, as shown previously. However, many people instead store the image as a file on the filesystem and store the path to this file as a VARCHAR.

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

Software developers argue passionately about this issue. There are good reasons for both solutions, but it's common for programmers to be unequivocal that we should always store files external to the database. It may put me in an unpopular position, but I'm going to describe several real risks to this design in the following sections.

Files Don't Obey DELETE

The first problem is one of garbage collection. If your images are outside the database and you delete the row that contains the path, there is no way for the file named by that path to be removed automatically.

```sql
DELETE FROM Screenshots WHERE bug_id = 1234 and image_id = 1;
```

Unless you design your application to remove these "orphaned" image files as you delete the database row that references them, they will accumulate.

Files Don't Obey Transaction Isolation

Normally, when you update or delete data, these changes aren't visible to other clients until you finish your transaction with COMMIT.

However, any change you make to files outside the database don't work this way. If you remove a file, it is immediately inaccessible to other clients. And if you change the contents of the file, other clients see those changes immediately, instead of seeing the previous content of the file while your transaction is still uncommitted.

```php
<?php

$stmt = $pdo->query("DELETE FROM Screenshots
                     WHERE bug_id = 1234 AND image_id =1");

unlink('images/screenshot1234-1.jpg');
```

```
// Other clients still see the row in the database,
// but not the image file.

$pdo->commit();
```

In practice, these kinds of anomalies may be infrequent. Also, the impact is minor in this example; a missing image is hardly rare in a web application. But in other scenarios, the consequences could be unfortunate.

Files Don't Obey ROLLBACK

It's normal to roll back transactions in case of errors, or even if the logic of your application requires that changes be canceled.

For example, suppose you remove a screenshot file as you execute a DELETE statement to remove the corresponding row in the database. If you roll back this change, the deletion of the row in the database is reversed, but the file is still gone.

Download **Phantom-Files/anti/rollback.php**

```
<?php

$stmt = $pdo->query("DELETE FROM Screenshots
                     WHERE bug_id = 1234 AND image_id =1");

unlink("images/screenshot1234-1.jpg");

$pdo->rollback();
```

The row in the database is restored but not the image file.

Files Don't Obey Database Backup Tools

Most database brands provide a client tool to assist in backing up a database that is in use. For example, MySQL provides mysqldump, Oracle provides rman, PostgreSQL provides pg_dump, SQLite provides the .dump command, and so on. Using a backup tool is important because if other clients are making changes concurrently, your backup could contain partial changes, potentially breaking referential integrity or even making the backup corrupt and useless for recovery.

A backup tool doesn't know how to include files referenced by pathname in a VARCHAR column of a table. So when you back up a database, you need to remember a two-step process: use the database backup tool, and then use a filesystem backup tool for the collection of external image files.

Even if you include the external files with the backup, it's hard to ensure that copies of these files are in sync with the transaction you used to back up the database. Applications may add or change image files at any time, perhaps only a moment after you began your database backup.

Files Don't Obey SQL Access Privileges

External files circumvent any privileges that you assign with the GRANT and REVOKE SQL statements. SQL privileges manage access to tables and columns, but they don't apply to external files named by strings in the database.

Files Are Not SQL Data Types

The path stored in screenshot_path is merely a string. The database doesn't verify that the string is a valid pathname, nor can the database verify that the file exists at the path you name. If the file is renamed, moved, or deleted, the database doesn't update the string in the database automatically. Any logic that treats this string as a pathname depends on code you write in your application.

Download Phantom-Files/anti/file-get.php

```php
<?php

define('DATA_DIRECTORY', '/var/bugtracker/data/');

$stmt = $pdo->query("SELECT image_path FROM Screenshots
                     WHERE bug_id = 1234 AND image_id = 1");
$row = $stmt->fetch();
$image_path = $row[0];

// Read the actual image -- I hope the path is correct!
$image = file_get_contents(DATA_DIRECTORY . $image_path);
```

One advantage of using a database is that it helps us preserve data integrity. When you put some of your data in external files, you circumvent this advantage, and you have to write application code to perform checks that should be handled by the database.

12.3 How to Recognize the Antipattern

The signs of this antipattern require a little investigation. If the project has any documentation to guide software administrators or if you have

the opportunity to interview the programmers who designed it (even if that's you), seek the answers to questions like the following:

- What is the data backup and restore procedure? How can a back-up be verified? Have you tested restoring data on a clean server or a different server than where the backup was made?
- Do images accumulate, or are they removed from the system when they are obsolete? What is the procedure for removing them? Is this an automated or manual procedure?
- Which users of the application have access to view images? How is access enforced? What do users see if they request to view images they don't have privilege to see?
- Can I cancel a change to an image? If so, should the application restore the previous state of an image?

Projects that are guilty of the antipattern typically fail to think through some or all of these questions. Not every application needs robust trans-action management or SQL access control for image files. You might find that taking a database offline during backups is a fair trade-off. If these answers are unclear or not forthcoming, it could indicate that the project designed their use of external files carelessly.

12.4 Legitimate Uses of the Antipattern

There are good reasons to store images or other large objects in files outside the database:

- The database is much leaner without images, because images tend to be large compared to simple datatypes like integers and strings.
- Backing up the database is faster and the result is smaller if images are not included. You must copy images from the filesys-tem as a separate backup step, but this can be more manageable than a huge database backup.
- If images are in files external to the database, it's easier to do ad hoc image previewing or editing. For example, if you need to apply a batch edit to all your images, it's especially good to keep images external to the database.

If these advantages of storing images in files are important and the issues described earlier are not deal-breakers, you may decide that it's the right thing to do in this project.

Some database brands support special SQL data types that do reference external files more or less transparently. Oracle calls this data type BFILE, while SQL Server 2008 calls it FILESTREAM.

> **Don't Rule Out Either Design**
>
> I designed an application that stored images outside a database for a contract project in 1992. My employer was hired to develop a registration application for a technical conference. As conference attendees arrived, a video camera took their picture, added it to their registration record, and printed it on their conference badge.
>
> My application was fairly simple. Each image could be inserted and updated only by one client application (if the person blinked or didn't like their photo, we could replace it during registration). There was no requirement for sophisticated transaction handling, concurrent access from multiple clients, or rollback. We were not using SQL access privileges. Previewing the images was simpler without having to fetch them from the database.
>
> I worked on this project at a time when the practical limits of applications and databases were much lower than what today's technology can handle. It made sense given these constraints to store images in a collection of directories and manage them with application code.

You need to plan how your application uses images to know whether the issues described in the "Antipattern" section would affect you. Make an informed decision, instead of listening to generalizations from programmers that storing images in external files is always the best solution.

12.5 Solution: Use BLOB Data Types As Needed

If any of the issues described in the "Antipattern" section of this chapter apply to you, you should consider storing images inside the database instead of in external files. All database brands support the BLOB data type, which you can use to store any binary data.

Download Phantom-Files/soln/create-screenshots.sql

```sql
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)
);
```

If you store an image in this way in a BLOB column, all the issues are solved:

- The image data is stored in the database. There is no extra step to load it. There's no risk that the file's pathname is incorrect.
- Deleting a row deletes the image automatically.
- Changes to an image are not visible to other clients until you commit the change.
- Rolling back a transaction restores the previous state of the image.
- Updating a row creates a lock, so no other client can update the same image concurrently.
- Database backups include all the images.
- SQL privileges control access to the image as well as the row.

The maximum size for a BLOB varies by database brand, but it's enough to store most images. All databases should support BLOB or something akin to it. MySQL, for example, provides a data type called MEDIUMBLOB that stores up to 16 megabytes, which is enough for most images. Oracle supports data types called LONG RAW or BLOB, with capacity up to 2 or 4 gigabytes, respectively. Similar data types are available in other database brands.

Images usually exist in a file to begin with, so you need some way to load them into a BLOB column in the database. Some databases provide functions to load external files. For example, MySQL has a function called LOAD_FILE() you can use to read a file, typically to store the content in a BLOB column.

Download Phantom-Files/soln/load-file.sql

```sql
UPDATE Screenshots
SET screenshot_image = LOAD_FILE('images/screenshot1234-1.jpg')
WHERE bug_id = 1234 AND image_id = 1;
```

You can also save the contents of a BLOB column to a file. For example, MySQL has an optional clause of the SELECT statement to store the result of a query verbatim, without any formatting to denote column or row termination.

Download Phantom-Files/soln/dumpfile.sql

```sql
SELECT screenshot_image
INTO DUMPFILE 'images/screenshot1234-1.jpg'
FROM Screenshots
WHERE bug_id = 1234 AND image_id =1;
```

You can also fetch the image data from the BLOB and output it directly. In a web application, you can output binary content such as an image, but you need to set the content type appropriately.

Download Phantom-Files/soln/binary-content.php

```php
<?php

header('Content-type: image/jpg');

$stmt = $pdo->query("SELECT screenshot_image FROM Screenshots
                        WHERE bug_id = 1234 AND image_id = 1");
$row = $stmt->fetch();

print $row[0];
```

Resources outside the database are not managed by the database.

*Whenever any result is sought by its aid, the question will
then arise—By what course of calculation can these results
be arrived at by the machine in the shortest time?*

▶ Charles Babbage, *Passages from the Life of a
Philosopher* (1864)

Chapter 13

Index Shotgun

"Hey! You got a minute? I could use your help," the Oklahoman accent on the phone is shouting over the data center ventilation. It's the lead database administrator for your company.

"Sure," you answer, a little unsure what he could want.

"The thing is, you've got a database here that's pretty much taken over the server," the DBA continues. "I got in there to take a look, and I see the problem. You've got no indexes on some tables and every index in the world on some other tables.

We've got to get this worked out or give you a server all to yerself, because nobody else can get any time!"

"I'm sorry—actually, I don't know that much about databases," you reply, trying to calm down the DBA. "We did our best to guess at the optimization, but obviously that's what an expert like you can do. Isn't there some database tuning you can do?"

"Son, I tuned everything I can; that's why we're still running down here at all," the DBA answers. "The only option left is to throttle your app, and I don't think you want that. We've got to stop guessing and start getting some answers on what your app needs the database to do."

You can tell this is getting over your head. Warily you ask, "What do you have in mind? I told you, we don't have expert database knowledge in our team."

"That's no problem," the DBA laughs. "You do know your application, right? That's the part that counts—and the part I *can't* help with. I'll get one of my boys to set you up with the right tools, and then we'll fix your bottleneck. You just need a little mentoring. You'll see."

13.1 Objective: Optimize Performance

Performance is the single most common concern I hear from database developers. Just look at the talks scheduled at any technical conference: they're full of tools and techniques to squeeze more work out of your database. When I give a talk about a way to structure a database or write SQL to give better reliability, security, or correctness, I'm not surprised when the only question from the audience is, "OK, but how does that affect performance?"

The best technique for improving performance in your database is to use indexes well. An index is a data structure that the database uses to correlate values to the rows where these values occur in a given column. An index provides an easy way for the database to find values more quickly than the brute-force method of searching the whole table from top to bottom.

Software developers typically don't understand how or when to use an index. Documentation and books about databases rarely or never contains a clear guide for when to use an index. Developers can only guess how to use indexes effectively.

13.2 Antipattern: Using Indexes Without a Plan

When we choose our indexes by guessing, we inevitably make some wrong choices. Misunderstandings about when to use indexes leads to mistakes in one of these three categories:

- Defining no indexes or not enough indexes
- Defining too many indexes or indexes that don't help
- Running queries that no index can help

No Indexes

We commonly read that a database incurs overhead as it keeps an index up-to-date. Each time we use INSERT, UPDATE, or DELETE, the database has to update the index data structures for that table to be consistent so that our subsequent searches use these indexes to find the right set of rows reliably.

We're trained to think that overhead means waste. So when we read that the database incurs overhead to keep an index updated, we want to eliminate that overhead. Some developers conclude that the remedy

> **Indexes Aren't Standard**
>
> Did you know that the ANSI SQL standard says nothing about indexes? The implementation and optimization of data storage is not specified by the SQL language, so every brand of database is free to implement indexes differently.
>
> Most brands have similar CREATE INDEX syntax, but each brand has flexibility to innovate and add their own proprietary technology. There's no standard for index capabilities. Likewise, there is no standard for index maintenance, automatic query optimization, query plan reporting, or commands like EXPLAIN.
>
> To get the most out of indexes, you have to study the documentation for your brand of database. The specific syntax and features of indexes vary greatly, but the logical concepts apply across the board.

is to eliminate the indexes. This advice is common, but it ignores the fact that indexes have benefits that justify their overhead.

Not all overhead is waste. Does your company employ administrative staff, legal professionals, accountants, and pay for facilities, even though those expenses don't directly contribute to generating revenue? Yes, because those people contribute to the success of your company in important ways.

In a typical application, you'll run hundreds of queries against a table for every one update. Every time you run a query that uses an index, you win back the overhead that went into maintaining that index.

An index can also help an UPDATE or DELETE statement by finding the desired rows quickly. For example, the index on the bug_id primary key helps the following statement:

Download Index-Shotgun/anti/update.sql

```sql
UPDATE Bugs SET status = 'FIXED' WHERE bug_id = 1234;
```

A statement that searches an unindexed column has to perform a full table scan to find matching rows.

Download Index-Shotgun/anti/update-unindexed.sql

```sql
UPDATE Bugs SET status = 'OBSOLETE' WHERE date_reported < '2000-01-01';
```

Too Many Indexes

You benefit from an index only if you run queries that use that index. There's no benefit to creating indexes that you don't use. Here are some examples:

Download Index-Shotgun/anti/create-table.sql

```sql
CREATE TABLE Bugs (
  bug_id        SERIAL PRIMARY KEY,
  date_reported DATE NOT NULL,
  summary       VARCHAR(80) NOT NULL,
  status        VARCHAR(10) NOT NULL,
  hours         NUMERIC(9,2),
❶ INDEX (bug_id),
❷ INDEX (summary),
❸ INDEX (hours),
❹ INDEX (bug_id, date_reported, status)
);
```

In the previous example, there are several useless indexes:

❶ bug_id: Most databases create an index automatically for a primary key, so it's redundant to define another index. There's no benefit to it, and it could just be extra overhead. Each database brand has its own rules for when to create an index automatically. You need to read the documentation for the database you use.

❷ summary: An indexing for a long string datatype like VARCHAR(80) is larger than an index for a more compact data type. Also, you're not likely to run queries that search or sort by the full summary column.

❸ hours: This is another example of a column that you're probably not going to search for specific values.

❹ bug_id, date_reported, status: There are good reasons to use compound indexes, but many people create compound indexes that are redundant or seldom used. Also, the order of columns in a compound index is important; you should use the columns left-to-right in search criteria, join criteria, or sorting order.

Hedging Your Bets

Bill Cosby told a story about his vacation in Las Vegas: He was so frustrated by losing in the casinos that he decided he had to win something—once—before he left. So he bought $200 in quarter chips, went to the roulette table, and put chips on every square, red and black. *He covered the table.* The dealer spun the ball. . . and it fell on the floor.

Some people create indexes on every column—and every combination of columns—because they don't know which indexes will benefit their queries. If you cover a database table with indexes, you incur a lot of overhead with no assurance of payoff.

When No Index Can Help

The next type of mistake is to run a query that can't use any index. Developers create more and more indexes, trying to find some magical combination of columns or index options to make their query run faster.

We can think of a database index using an analogy to a telephone book. If I ask you to look up everyone in the telephone book whose last name is Charles, it's an easy task. All the people with the same last name are listed together, because that's how the telephone book is ordered.

However, if I ask you to look up everyone in the telephone book whose *first name* is Charles, this doesn't benefit from the order of names in the book. Anyone can have that first name, regardless of their last name, so you have to search through the entire book line by line.

The telephone book is ordered by last name and then by first name, just like a compound database index on last_name, first_name. This index doesn't help you search by first name.

Download **Index-Shotgun/anti/create-index.sql**

```
CREATE INDEX TelephoneBook ON Accounts(last_name, first_name);
```

Some examples of queries that can't benefit from this index include the following:

- ```
  SELECT * FROM Accounts ORDER BY first_name, last_name;
  ```

  This query shows the telephone book scenario. If you create a compound index for the columns last_name followed by first_name (as in a telephone book), the index doesn't help you sort primarily by first_name.

- ```
  SELECT * FROM Bugs WHERE MONTH(date_reported) = 4;
  ```

  Even if you create an index for the date_reported column, the order of the index doesn't help you search by month. The order of this index is based on the entire date, starting with the year. But each year has a fourth month, so the rows where the month is equal to 4 are scattered through the table.

Some databases support indexes on expressions, or indexes on generated columns, as well as indexes on plain columns. But you have to define the index prior to using it, and that index helps only for the expression you specify in its definition.

- `SELECT * FROM Bugs WHERE last_name = 'Charles' OR first_name = 'Charles';`

We're back to the problem that rows with that specific first name are scattered unpredictably with respect to the order of the index we defined. The result of the previous query is the same as the result of the following:

```
SELECT * FROM Bugs WHERE last_name = 'Charles'
UNION
SELECT * FROM Bugs WHERE first_name = 'Charles';
```

The index in our example helps find that last name, but it doesn't help find that first name.

- `SELECT * FROM Bugs WHERE description LIKE '%crash%';`

Because the pattern in this search predicate could occur anywhere in the string, there's no way the sorted index data structure can help.

13.3 How to Recognize the Antipattern

The following are symptoms of the Index Shotgun antipattern:

- "Here's my query; how can I make it faster?"

  This is probably the single most common SQL question, but it's missing details about table description, indexes, data volume, and measurements of performance and optimization. Without this context, any answer is just guesswork.

- "I defined an index on every field; why isn't it faster?"

  This is the classic Index Shotgun antisolution. You've tried every possible index—but you're shooting in the dark.

- "I read that indexes make the database slow, so I don't use them."

  Like many developers, you're looking for a one-size-fits-all strategy for performance improvement. No such blanket rule exists.

Low-Selectivity Indexes

Selectivity is a statistic about a database index. It's the ratio of the number of distinct values in the index to the total number of rows in the table:

```
SELECT COUNT(DISTINCT status) /
    COUNT(status) AS selectivity FROM Bugs;
```

The lower the selectivity ratio, the less effective an index is. Why is this? Let's consider an analogy.

This book has an index of a different type: each entry in a book's index lists the pages where the entry's words appear. If a word appears frequently in the book, it may list many page numbers. To find the part of the book you're looking for, you have to turn to each page in the list one by one.

Indexes don't bother to list words that appear on too many pages. If you have to flip back and forth from the index to the pages of the book too much, then you might as well just read the whole book cover to cover.

Likewise in a database index, if a given value appears on many rows in the table, it's more trouble to read the index than simply to scan the entire table. In fact, in these cases it can actually be more expensive to use that index.

Ideally your database tracks the selectivity of indexes and shouldn't use an index that gives no benefit.

13.4 Legitimate Uses of the Antipattern

If you need to design a database for general use, without knowing what queries are important to optimize, you can't be sure of which indexes are best. You have to make an educated guess. It's likely that you'll miss some indexes that could have given benefit. It's also likely that you'll create some indexes that turn out to be unneeded. But you have to make the best guess you can.

13.5 Solution: MENTOR Your Indexes

The Index Shotgun antipattern is about creating or dropping indexes without reason, so let's come up with ways to analyze a database and find good reasons to include indexes or omit them.

The Database Isn't Always the Bottleneck

Common wisdom in software developer communities is that the database is always the slowest part of your application and the source of performance issues. However, this isn't true.

For example, in one application I worked on, my manager asked me to find out why it was so slow, and he insisted it was the fault of the database. After I used a profiling tool to measure the application code, I found that it spent 80 percent of its time parsing its own HTML output to find form fields so it could populate values into forms. The performance issue had nothing to do with the database queries.

Before making assumptions about where the performance problem exists, use software diagnostic tools to measure. Otherwise, you could be practicing premature optimization.

You can use the mnemonic *MENTOR* to describe a checklist for analyzing your database for good index choices: *Measure*, *Explain*, *Nominate*, *Test*, *Optimize*, and *Rebuild*.

Measure

You can't make informed decisions without information. Most databases provide some way to log the time to execute SQL queries so you can identify the operations with the greatest cost. For example:

- Microsoft SQL Server and Oracle both have *SQL Trace* facilities and tools to report and analyze trace results. Microsoft calls this tool the *SQL Server Profiler*, and Oracle calls it *TKProf*.

- MySQL and PostgreSQL can log queries that take longer to execute than a specified threshold of time. MySQL calls this the *slow query log*, and its long_query_time configuration parameter defaults to 10 seconds. PostgreSQL has a similar configuration variable log_min_duration_statement.

    PostgreSQL also has a companion tool called *pgFouine*, which helps you analyze the query log and identify queries that need attention (http://pgfouine.projects.postgresql.org/).

Once you know which queries account for the most time in your application, you know where you should focus your optimizing attention for

the greatest benefit. You might even find that all queries are working efficiently except for one single bottleneck query. This is the query you should start optimizing.

The area of greatest cost in your application isn't necessarily the most time-consuming query if that query is run only rarely. Other simpler queries might be run frequently, more often than you would expect, so they account for more total time. Giving attention to optimizing these queries gives you more bang for your buck.

Disable any query result caching while you're measuring query performance. This type of cache is designed to bypass query execution and index usage, so it won't give an accurate measurement.

You can get more accurate information by profiling your application after you deploy it. Collect aggregate data of where the code spends its time when real users are using it, and against the real database. You should monitor profiling data from time to time to be sure you haven't acquired a new bottleneck.

Remember to disable or turn down the reporting rate of profilers after you're done measuring, because these tools incur some overhead.

Explain

Having identified the query that has the greatest cost, your next step is to find out *why* it's so slow. Every database uses an optimizer to pick indexes for your query. You can get the database to give you a report of its analysis, called the *query execution plan* (QEP).

The syntax to request a QEP varies by database brand:

Database Brand	QEP Reporting Solution
IBM DB2	EXPLAIN, db2expln command, or Visual Explain
Microsoft SQL Server	SET SHOWPLAN_XML, or Display Execution Plan
MySQL	EXPLAIN
Oracle	EXPLAIN PLAN
PostgreSQL	EXPLAIN
SQLite	EXPLAIN

There's no standard for what information a QEP report includes or the format of the report. In general, the QEP shows you which tables are involved in a query, how the optimizer chooses to use indexes, and what order it will access the tables. The report may also include statistics, such as the number of rows generated by each stage of the query.

table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
Bugs	ALL	PRIMARY,bug_id	NULL	NULL	NULL	4650	100	Using where; Using temporary; Using filesort
BugsProducts	ref	PRIMARY,product_id	PRIMARY	8	Bugs.bug_id	1	100	Using index
Products	ALL	PRIMARY,product_id	NULL	NULL	NULL	3	100	Using where; Using join buffer

Figure 13.1: MySQL query execution plan

Let's look at a sample SQL query and request a QEP report:

Download **Index-Shotgun/soln/explain.sql**

```
EXPLAIN SELECT Bugs.*
FROM Bugs
JOIN (BugsProducts JOIN Products USING (product_id))
  USING (bug_id)
WHERE summary LIKE '%crash%'
  AND product_name = 'Open RoundFile'
ORDER BY date_reported DESC;
```

In the MySQL QEP report shown in Figure 13.1, the key column shows that this query makes use of only the primary key index BugsProducts. Also, the extra notes in the last column indicate that the query will sort the result in a temporary table, without the benefit of an index.

The LIKE expression forces a full table scan in Bugs, and there is no index on Products.product_name. We can improve this query if we create a new index on product_name and also use a full-text search solution.[1]

The information in a QEP report is vendor-specific. In this example, you should read the MySQL manual page "Optimizing Queries with EXPLAIN" to understand how to interpret the report.[2]

Nominate

Now that you have the optimizer's QEP for your query, you should look for cases where the query accesses a table without using an index.

---

1. See Chapter 17, *Poor Man's Search Engine*, on page 190.
2. http://dev.mysql.com/doc/refman/5.1/en/using-explain.html

Covering Indexes

If an index provides all the columns we need, then we don't need to read rows of data from the table at all.

Imagine if telephone book entries contained only a page number; after you looked up a name, you would then have to turn to the page it referenced to get the actual phone number. It makes more sense to look up the information in one step. Looking up a name is quick because the book is ordered, and right there you can get other attributes you need for that entry, such as the phone number and perhaps also an address.

This is how a *covering index* works. You can define the index to include extra columns, even though they're not otherwise necessary for the index.

```
CREATE INDEX BugCovering ON Bugs
  (status, bug_id, date_reported, reported_by, summary);
```

If your query references only the columns included in the index data structure, the database generates your query results by reading only the index.

```
SELECT status, bug_id, date_reported, summary
FROM Bugs WHERE status = 'OPEN';
```

The database doesn't need to read the corresponding rows from this table. You can't use covering indexes for every query, but when you can, it's usually a great win for performance.

Some databases have tools to do this for you, collecting query trace statistics and proposing a number of changes, including creating new indexes that you're missing but would benefit your query. For example:

- IBM DB2 Design Advisor
- Microsoft SQL Server Database Engine Tuning Advisor
- MySQL Enterprise Query Analyzer
- Oracle Automatic SQL Tuning Advisor

Even without automatic advisors, you can learn how to recognize when an index could benefit a query. You need to study your database's documentation to interpret the QEP report.

Test

This step is important: after creating indexes, profile your queries again. It's important to confirm that your change made a difference so you know that your work is done.

You can also use this step to impress your boss and justify the work you put into this optimization. You don't want your weekly status to be like this: "I've tried everything I can think of to fix our performance issues, and we'll just have to wait and see. . . ." Instead, you should have the opportunity to report this: "I determined we could create one new index on a high-activity table, and I improved the performance of our critical queries by 38 percent."

Optimize

Indexes are compact, frequently used data structures, which makes them good candidates for keeping in cache memory. Reading indexes in memory improves performance an order of magnitude greater than reading indexes via disk I/O.

Database servers allow you to configure the amount of system memory to allocate for caching. Most databases set the cache buffer size pretty low to ensure that the database works well on a wide variety of systems. You probably want to raise the size of the cache.

How much memory should you allocate to cache? There's no single answer to this, because it depends on the size of your database and how much system memory you have available.

You may also benefit from preloading indexes into cache memory, instead of relying on database activity to bring the most frequently used data or indexes into the cache. For instance, on MySQL, use the LOAD INDEX INTO CACHE statement.

Rebuild

Indexes provide the most efficiency when they are *balanced*. Over time, as you update and delete rows, the indexes may become progressively imbalanced, similar to how filesystems become fragmented over time. In practice, you may not see a large difference between an index that is optimal vs. one that has some imbalance. But we want to get the most out of indexes, so it's worthwhile to perform maintenance on a regular schedule.

Like most features related to indexes, each database brand uses vendor-specific terminology, syntax, and capabilities.

Database Brand	Index Maintenance Command
IBM DB2	REBUILD INDEX
Microsoft SQL Server	ALTER INDEX … REORGANIZE, ALTER INDEX … REBUILD, or DBCC DBREINDEX
MySQL	ANALYZE TABLE or OPTIMIZE TABLE
Oracle	ALTER INDEX … REBUILD
PostgreSQL	VACUUM or ANALYZE
SQLite	VACUUM

How frequently should you rebuild an index? You might hear generic answers such as "once a week," but in truth there's no single answer that fits all applications. It depends on how frequently you commit changes to a given table that could introduce imbalance. It also depends on how large the table is and how important it is to get optimal benefit from indexes for this table. Is it worth spending hours rebuilding indexes for a large but seldom used table if you can expect to gain only an extra 1 percent performance? You're the best judge of this, because you know your data and your operation requirements better than anyone else does.

A lot of the knowledge about getting the most out of indexes is vendor-specific, so you'll need to research the brand of database you use. Your resources include the database manual, books and magazines, blogs and mailing lists, and also lots of experimentation on your own. The most important rule is that guessing blindly at indexing isn't a good strategy.

Know your data, know your queries, and MENTOR your indexes.

Part III

Query Antipatterns

As we know, there are known knowns; there are things we know we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns—the ones we don't know we don't know.
► Donald Rumsfeld

Chapter 14

Fear of the Unknown

In our example bugs database, the Accounts table has columns first_name and last_name. You can use an expression to format the user's full name as a single column using the string concatenation operator:

Download Fear-Unknown/intro/full-name.sql

```sql
SELECT first_name || ' ' || last_name AS full_name FROM Accounts;
```

Suppose your boss asks you to modify the database to add the user's middle initial to the table (perhaps two users have the same first name and last name, and the middle initial is a good way to avoid confusion). This is a pretty simple alteration. You also manually add the middle initials for a few users.

Download Fear-Unknown/intro/middle-name.sql

```sql
ALTER TABLE Accounts ADD COLUMN middle_initial CHAR(2);

UPDATE Accounts SET middle_initial = 'J.' WHERE account_id = 123;
UPDATE Accounts SET middle_initial = 'C.' WHERE account_id = 321;

SELECT first_name || ' ' || middle_initial || ' ' || last_name AS full_name
FROM Accounts;
```

Suddenly, the application ceases to show any names. Actually, on a second look, you notice it isn't universal. Only the names of users who have specified their middle initial appear normally; every else's name is now blank.

What happened to everyone else's names? Can you fix this before your boss notices and starts to panic, thinking you've lost data in the database?

14.1 Objective: Distinguish Missing Values

It's inevitable that some data in your database has no value. Either you need to insert a row before you have discovered the values for all the columns, or else some columns have no meaningful value in some legitimate circumstances. SQL supports a special null value, corresponding to the NULL keyword.

There are many ways you can use a null value productively in SQL tables and queries:

- You can use null in place of a value that is not available at the time the row is created, such as the date of termination for an employee who is still working.

- A given column can use a null value when it has no applicable value on a given row, such as the fuel efficiency rating for a car that is fully electric.

- A function can return a null value when given invalid inputs, as in DAY('2009-12-32').

- An outer join uses null values as placeholders for the columns of an unmatched table in an outer join.

The objective is to write queries against columns that contain null.

14.2 Antipattern: Use Null as an Ordinary Value, or Vice Versa

Many software developers are caught off-guard by the behavior of null in SQL. Unlike in most programming languages, SQL treats null as a special value, different from zero, false, or an empty string. This is true in standard SQL and most brands of database. However, in Oracle and Sybase, null is exactly the same as a string of zero length. The null value follows some special behavior, too.

Using Null in Expressions

One case that surprises some people is when you perform arithmetic on a column or expression that is null. For example, many programmers would expect the result to be 10 for bugs that have been given no estimate in the hours column, but instead the query returns null.

Download Fear-Unknown/anti/expression.sql

```
SELECT hours + 10 FROM Bugs;
```

Null is not the same as zero. A number ten greater than an unknown is still an unknown.

Null is not the same as a string of zero length. Combining any string with null in standard SQL returns null (despite the behavior in Oracle and Sybase).

Null is not the same as false. Boolean expressions with AND, OR, and NOT also produce results that some people find confusing.

Searching Nullable Columns

The following query returns only rows where assigned_to has the value 123, not rows with other values or rows where the column is null:

Download Fear-Unknown/anti/search.sql

```sql
SELECT * FROM Bugs WHERE assigned_to = 123;
```

You might think that the next query returns the complementary set of rows, that is, all rows *not* returned by the previous query:

Download Fear-Unknown/anti/search-not.sql

```sql
SELECT * FROM Bugs WHERE NOT (assigned_to = 123);
```

However, neither query result includes rows where assigned_to is null. Any comparison to null returns *unknown*, not true or false. Even the negation of null is still null.

It's common to make the following mistakes searching for null values or non-null values:

Download Fear-Unknown/anti/equals-null.sql

```sql
SELECT * FROM Bugs WHERE assigned_to = NULL;

SELECT * FROM Bugs WHERE assigned_to <> NULL;
```

The condition in a WHERE clause is satisfied only when the expression is true, but a comparison to NULL is never true; it's unknown. It doesn't matter whether the comparison is for equality or inequality; it's still unknown, which is certainly not true. Neither of the previous queries return rows where assigned_to is null.

Using Null in Query Parameters

It's also difficult to use null in a parameterized SQL expression as if the null were an ordinary value.

```
SELECT * FROM Bugs WHERE assigned_to = ?;
```

The previous query returns predictable results when you send an ordinary integer value for the parameter, but you can't use a literal NULL as the parameter.

Avoiding the Issue

If handling null makes queries more complex, many software developers choose to disallow nulls in the database. Instead, they choose an ordinary value to signify "unknown" or "inapplicable."

> **"We Hate Nulls!"**
>
> Jack, a software developer, described his client's request that he prevent any null values in their database. Their explanation was simply "We hate nulls" and that the presence of nulls would lead to errors in their application code. Jack asked what other value should he use to represent a missing value.
>
> I told Jack that representing a missing value is the exact purpose of null. No matter what other value he chooses to signify a missing value, he'd need to modify the application code to treat that value as special.
>
> Jack's client's attitude to null is wrong; similarly, I could say that I don't like writing code to prevent division by zero errors, but that doesn't make it a good choice to prohibit all instances of the value zero.

What exactly is wrong with this practice? In the following example, declare the previously nullable columns assigned_to and hours as NOT NULL:

```
CREATE TABLE Bugs (
  bug_id           SERIAL PRIMARY KEY,
  -- other columns
  assigned_to      BIGINT UNSIGNED NOT NULL,
  hours            NUMERIC(9,2) NOT NULL,
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id)
);
```

Let's say you use -1 to represent an unknown value.

```
INSERT INTO Bugs (assigned_to, hours) VALUES (-1, -1);
```

The hours column is numeric, so you're restricted to a numeric value to mean "unspecified." It has to have no meaning in that column, so you chose a negative value. But the value -1 would throw off calculations

such as SUM() or AVG(). You have to exclude rows with this value, using special-case expressions, which is what you were trying to avoid by prohibiting null.

```
SELECT AVG( hours ) AS average_hours_per_bug FROM Bugs
WHERE hours <> -1;
```

In another column, the value -1 might be significant, so you have to choose a different value on a case-by-case basis for each column. You also have to remember or document the special values used by each column. This adds a lot of meticulous and unnecessary work to a project.

Now let's look at the assigned_to column. It is a foreign key to the Accounts table. When a bug has been reported but not assigned yet, what non-null value can you use? Any non-null value must reference a row in Accounts, so you need to create a placeholder row in Accounts, meaning "no one" or "unassigned." It seems ironic to create an account to reference, so you can represent the absence of a reference to a real user's account.

When you declare a column as NOT NULL, it should be because it would make no sense for the row to exist without a value in that column. For example, the Bugs.reported_by column must have a value, because every bug was reported by someone. But a bug may exist without having been assigned yet. Missing values should be null.

14.3 How to Recognize the Antipattern

If you find yourself or another member of your team describing issues like the following, it could be because of improper handling of nulls:

- "How do I find rows where no value has been set in the assigned_to (or other) column?"

  You can't use the equality operator for null. We'll see how to use the IS NULL predicate later in this chapter.

- "The full names of some users appear blank in the application presentation, but I can see them in the database."

  The problem might be that you're concatenating strings with null, which produces null.

> ### Are Nulls Relational?
>
> There is some controversy about null in SQL. E. F. Codd, the computer scientist who developed relational theory, recognized the need for null to signify missing data. However, C. J. Date has shown that the behavior of null as defined in the SQL standard has some edge cases that conflict with relational logic.
>
> The fact is that most programming languages are not perfect implementations of computer science theories. The SQL language supports null, for better or for worse. We've seen some of the hazards, but you can learn how to account for these cases and use null productively.

- "The report of total hours spent working on this project includes only a few of the bugs that we completed! Only those for which we assigned a priority are included."

  Your aggregate query to sum the hours probably includes an expression in the WHERE clause that fails to be true when priority is null. Watch out for unexpected results when you use *not equals* expressions. For example, on rows where priority is null, the expression priority <> 1 will fail.

- "It turns out we can't use the string we've been using to represent *unknown* in the Bugs table, so we need to have a meeting to discuss what new special value we can use and estimate the development time to migrate our data and convert our code to use that value."

  This is a likely consequence of assigning a special flag value that could be a legitimate value in your column's domain. Eventually, you may find you need to use that value for its literal meaning instead of its flag meaning.

Recognizing problems with your handling of nulls can be elusive. Problems may not occur during application testing, especially if you overlooked some edge cases while designing sample data for tests. However, when your application is used in production, data can take many unanticipated forms. If a null can creep into the data, you can count on it happening.

14.4   Legitimate Uses of the Antipattern

Using null is not the antipattern; the antipattern is using null like an ordinary value or using an ordinary value like null.

One situation where you need to treat null as an ordinary value is when you import or export external data. In a text file with comma-separated fields, all values must be represented by text. For example, in MySQL's mysqlimport tool for loading data from a text file, \N represents a null.

Similarly, user input cannot represent a null directly. An application that accepts user input may provide a way to map some special input sequence to null. For example, Microsoft .NET 2.0 and newer supports a property called ConvertEmptyStringToNull for web user interfaces. Parameters and bound fields with this property automatically convert an empty string value ("") to null.

Finally, null won't work if you need to support several distinct missing-value cases. Let's say you want to distinguish between a bug that has never been assigned and a bug that was previously assigned to a person who has left the project—you have to use a distinct value for each state.

14.5   Solution: Use Null as a Unique Value

Most problems with null values are based on a common misunder-standing of the behavior of SQL's three-valued logic. For programmers accustomed to the conventional true/false logic implemented in most other languages, this can be a challenge. You can handle null values in SQL queries with a little study of how they work.

Null in Scalar Expressions

Suppose Stan is thirty years old, while Oliver's age is unknown. If I ask you whether Stan is older than Oliver, your only possible answer is "I don't know." If I ask you whether Stan is the same age as Oliver, your answer is also "I don't know." If I ask you what is the sum of Stan's age and Oliver's age, your answer is the same.

Suppose Charlie's age is also unknown. If I ask you whether Oliver's age is equal to Charlie's age, your answer is still "I don't know." This shows why the result of a comparison like NULL = NULL is also null.

The following table describes some cases where programmers expect one result but get something different.

Expression	Expected	Actual	Because
NULL = 0	TRUE	NULL	Null is not zero.
NULL = 12345	FALSE	NULL	Unknown if the unspecified value is equal to a given value.
NULL <> 12345	TRUE	NULL	Also unknown if it's unequal.
NULL + 12345	12345	NULL	Null is not zero.
NULL || 'string'	'string'	NULL	Null is not an empty string.
NULL = NULL	TRUE	NULL	Unknown if one unspecified value is the same as another.
NULL <> NULL	FALSE	NULL	Also unknown if they're different.

Of course, these examples apply not only when using the NULL keyword but also to any column or expression whose value is null.

Null in Boolean Expressions

The key concept for understanding how null values behave in boolean expressions is that null is neither true nor false.

The following table describes some cases where programmers expect one result but get something different.

Expression	Expected	Actual	Because
NULL AND TRUE	FALSE	NULL	Null is not false.
NULL AND FALSE	FALSE	FALSE	Any truth value AND FALSE is false.
NULL OR FALSE	FALSE	NULL	Null is not false.
NULL OR TRUE	TRUE	TRUE	Any truth value OR TRUE is true.
NOT (NULL)	TRUE	NULL	Null is not false.

A null value certainly isn't true, but it isn't the same as false. If it were, then applying NOT to a null value would result in true. But that's not the way it works; NOT (NULL) results in another null. This confuses some people who try to use boolean expressions with null.

Searching for Null

Since neither equality nor inequality return true when comparing one value to a null value, you need some other operation if you are searching for a null. Older SQL standards define the IS NULL predicate, which returns true if its single operand is null. The opposite, IS NOT NULL, returns false if its operand is null.

Download Fear-Unknown/soln/search.sql

```sql
SELECT * FROM Bugs WHERE assigned_to IS NULL;

SELECT * FROM Bugs WHERE assigned_to IS NOT NULL;
```

> ### The Right Result for the Wrong Reason
>
> Consider the following case, where a nullable column may behave in a more intuitive way by serendipity.
>
> ```sql
> SELECT * FROM Bugs WHERE assigned_to <> 'NULL';
> ```
>
> Here the nullable column assigned_to is compared to the string value 'NULL' (notice the quotes), instead of the actual NULL keyword.
>
> Where assigned_to is null, comparing it to the string 'NULL' is not true. The row is excluded from the query result, which is the programmer's intent.
>
> The other case is that the column is an integer compared to the string 'NULL'. The integer value of a string like 'NULL' is zero in most brands of database. The integer value of assigned_to is almost certainly greater than zero. It's unequal to the string, so therefore the row is included in the query result.
>
> Thus, by making another common mistake, that of putting quotes around the NULL keyword, some programmers may unwittingly get the result they wanted. Unfortunately, this coincidence doesn't hold in other searches, such as WHERE assigned_to = 'NULL'.

In addition, the SQL-99 standard defines another comparison predicate, IS DISTINCT FROM. This works like an ordinary inequality operator <>, except that it always returns true or false, even when its operands are null.

This relieves you from writing tedious expressions that must test IS NULL before comparing to a value. The following two queries are equivalent:

Download Fear-Unknown/soln/is-distinct-from.sql

```sql
SELECT * FROM Bugs WHERE assigned_to IS NULL OR assigned_to <> 1;

SELECT * FROM Bugs WHERE assigned_to IS DISTINCT FROM 1;
```

You can use this predicate with query parameters to which you want to send either a literal value or NULL:

Download Fear-Unknown/soln/is-distinct-from-parameter.sql

```sql
SELECT * FROM Bugs WHERE assigned_to IS DISTINCT FROM ?;
```

Support for IS DISTINCT FROM is inconsistent among database brands. PostgreSQL, IBM DB2, and Firebird do support it, whereas Oracle and Microsoft SQL Server don't support it yet. MySQL offers a proprietary operator <=> that works like IS NOT DISTINCT FROM.

Declare Columns NOT NULL

It's recommended to declare a NOT NULL constraint on a column for which a null would break a policy in your application or otherwise be nonsensical. It's better to allow the database to enforce constraints uniformly rather than rely on application code.

For example, it's reasonable that any entry in the Bugs table should have a non-null value for the date_reported, reported_by, and status columns. Likewise, rows in child tables like Comments must include a non-null bug_id, referencing an existing bug. You should declare these columns with the NOT NULL option.

Some people recommend that you define a DEFAULT for every column, so that if you omit the column in an INSERT statement, the column gets some value instead of null. That's good advice for some columns but not for other columns. For example, Bugs.reported_by should not be null. What default, if any, should you declare for this column? It's valid and common for a column to need a NOT NULL constraint yet have no logical default value.

Dynamic Defaults

In some queries, you may need to force a column or expression to be non-null for the sake of simplifying the query logic, but you don't want that value to be stored. What you need is a way to set a default for a given column or expression ad hoc, in a specific query only. For this you should use the COALESCE() function. This function accepts a variable number of arguments and returns its first non-null argument.

In the story about concatenating users' names shown in the story opening this chapter, you could use COALESCE() to make an expression that uses a single space in place of the middle initial, so a null-valued middle initial doesn't make the whole expression become null.

Download Fear-Unknown/soln/coalesce.sql

```sql
SELECT first_name || COALESCE(' ' || middle_initial || ' ', ' ') || last_name
  AS full_name
FROM Accounts;
```

COALESCE() is a standard SQL function. Some database brands support a similar function by another name, such as NVL() or ISNULL().

Use null to signify a missing value for any data type.

Chapter 15

Ambiguous Groups

Suppose your boss needs to know which projects in the bugs database are still active and which projects have been abandoned. One report he asks you to generate is the latest bug reported per product. You write a query using the MySQL database to calculate the greatest value in the date_reported column per group of bugs sharing a given product_id. The report looks like this:

product_name	latest	bug_id
Open RoundFile	2010-06-01	1234
Visual TurboBuilder	2010-02-16	3456
ReConsider	2010-01-01	5678

Your boss is a detail-oriented person, and he spends some time looking up each bug listed in the report. He notices that the row listed as the most recent for "Open RoundFile" shows a bug_id that isn't the latest bug. The full data shows the discrepancy:

product_name	date_reported	bug_id	
Open RoundFile	2009-12-19	*1234*	*This bug_id. . .*
Open RoundFile	*2010-06-01*	2248	*doesn't match this date*
Visual TurboBuilder	2010-02-16	3456	
Visual TurboBuilder	2010-02-10	4077	
Visual TurboBuilder	2010-02-16	5150	
ReConsider	2010-01-01	5678	
ReConsider	2009-11-09	8063	

How can you explain this problem? Why does it affect one product but not the others? How can you get the desired report?

15.1 Objective: Get Row with Greatest Value per Group

Most programmers who learn SQL get to the stage of using GROUP BY in a query, applying some aggregate function to groups of rows, and getting a result with one row per group. This is a powerful feature that makes it easy to get a wide variety of complex reports using relatively little code.

For example, a query to get the latest bug reported for each product in the bugs database looks like this:

Download Groups/anti/groupbyproduct.sql

```sql
SELECT product_id, MAX(date_reported) AS latest
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

A natural extension to this query is to request the ID of the specific bug with the latest date reported:

Download Groups/anti/groupbyproduct.sql

```sql
SELECT product_id, MAX(date_reported) AS latest, bug_id
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

However, this query results in either an error or an unreliable answer. This is a common source of confusion for programmers using SQL.

The objective is to run a query that not only reports the greatest value in a group (or the least value or the average value) but also includes other attributes of the row where that value is found.

15.2 Antipattern: Reference Nongrouped Columns

The root cause of this antipattern is simple, and it reveals a common misconception that many programmers have about how grouping queries work in SQL.

The Single-Value Rule

The rows in each group are those rows with the same value in the column or columns you name after GROUP BY. For example, in the following query, there is one row group for each distinct value in product_id.

Download Groups/anti/groupbyproduct.sql

```sql
SELECT product_id, MAX(date_reported) AS latest
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

Every column in the select-list of a query must have a single value row per row group. This is called the *Single-Value Rule*. Columns named in the GROUP BY clause are guaranteed to be exactly one value per group, no matter how many rows the group matches.

The MAX() expression is also guaranteed to result in a single value for each group: the highest value found in the argument of MAX() over all the rows in the group.

However, the database server can't be so sure about any other column named in the select-list. It can't always guarantee that the same value occurs on every row in a group for those other columns.

```
SELECT product_id, MAX(date_reported) AS latest, bug_id
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

In this example, there are many distinct values for bug_id for a given product_id, because the BugsProducts table associates multiple bugs to a given product. In a grouping query that reduces to a single row per product, there's no way to represent all the values of bug_id.

Since there is no guarantee of a single value per group in the "extra" columns, the database assumes that they violate the Single-Value Rule. Most brands of database report an error if you try to run any query that tries to return a column other than those columns named in the GROUP BY clause or as arguments to aggregate functions.

MySQL and SQLite have different behavior from other brands of database, which we'll explore in Section 15.4, *Legitimate Uses of the Antipattern*, on page 178.

Do-What-I-Mean Queries

The common misconception that programmers have is that SQL can guess which bug_id you want in the report, based on the fact that MAX() is used in another column. Most people assume that if the query fetches the greatest value, then other columns named will naturally take their value from the same row where that greatest value occurs.

Unfortunately, SQL can't make this inference in several cases:

- If two bugs have the exact same value for date_reported and that is the greatest value in the group, which value of bug_id should the query report?

- If you query for two different aggregate functions, for example MAX() and MIN(), these probably correspond to two different rows in the group. Which bug_id should the query return for this group?

  Download Groups/anti/maxandmin.sql

  ```
  SELECT product_id, MAX(date_reported) AS latest,
    MIN(date_reported) AS earliest, bug_id
  FROM Bugs JOIN BugsProducts USING (bug_id)
  GROUP BY product_id;
  ```

- If none of the rows in the table matches the value returned by the aggregate function, what is the value of bug_id? This is commonly true for the functions AVG(), COUNT(), and SUM().

  Download Groups/anti/sumbyproduct.sql

  ```
  SELECT product_id, SUM(hours) AS total_project_estimate, bug_id
  FROM Bugs JOIN BugsProducts USING (bug_id)
  GROUP BY product_id;
  ```

These are examples of why the Single-Value Rule is important. Not every query that fails to follow this rule would produce an ambiguous result, but many do. It would be clever if the database could tell an ambiguous query from an unambiguous one and produce an error only when the data contains ambiguity. But that would not be good for application reliability; it would mean that the same query might be valid or invalid, depending on the state of data.

15.3 How to Recognize the Antipattern

In most brands of database, writing a query that violates the Single-Value Rule should elicit an error immediately as you prepare the query. The following are examples of error messages given by some brands of database:

- Firebird 2.1:

  ```
  Invalid expression in the select list (not contained in either an
  aggregate function or the GROUP BY clause)
  ```

- IBM DB2 9.5:

  ```
  An expression starting with "BUG_ID" specified in a SELECT clause,
  HAVING clause, or ORDER BY clause is not specified in the GROUP BY
  clause or it is in a SELECT clause, HAVING clause, or ORDER BY clause
  with a column function and no GROUP BY clause is specified.
  ```

GROUP BY and DISTINCT

SQL supports a query modifier called DISTINCT that reduces the rows of the query result so that every row is unique. For example, the following query reports who reported bugs and which days they reported bugs, but only one row per date and person:

```
SELECT DISTINCT date_reported, reported_by FROM Bugs;
```

A grouping query can achieve the same result by omitting any aggregate function. The query result is reduced to one row for each distinct pair of values in the column named in the GROUP BY clause:

```
SELECT date_reported, reported_by FROM Bugs
GROUP BY date_reported, reported_by;
```

Both queries produce the same result and should be optimized and executed similarly, so the difference in this example is only a matter of preference.

- Microsoft SQL Server 2008:

  ```
  Column 'Bugs.bug_id' is invalid in the select list because it is not
  contained in either an aggregate function or the GROUP BY clause.
  ```

- MySQL 5.1, after setting the ONLY_FULL_GROUP SQL mode to disallow ambiguous queries.

  ```
  'bugs.b.bug_id' isn't in GROUP BY
  ```

- Oracle 10.2:

  ```
  not a GROUP BY expression
  ```

- PostgreSQL 8.3:

  ```
  column "bp.bug_id" must appear in the GROUP BY clause or be
  used in an aggregate function
  ```

In SQLite and in MySQL, ambiguous columns may contain unexpected and unreliable values. In MySQL, the value returned is from the first row in the group, where *first* corresponds to physical storage. SQLite gives the opposite result: the value is from the *last* row in the group. In both cases, the behavior is not documented, and these databases aren't obligated to work the same in future versions. It's your responsibility to notice these cases and to design your queries to avoid ambiguity.

15.4 Legitimate Uses of the Antipattern

As we've seen, MySQL and SQLite can't guarantee a reliable result for a column that doesn't fit the Single-Value Rule. There are cases when you can take advantage of the fact that these databases enforce the rule less strictly than other brands.

Download **Groups/legit/functional.sql**

```sql
SELECT b.reported_by, a.account_name
FROM Bugs b JOIN Accounts a ON (b.reported_by = a.account_id)
GROUP BY b.reported_by;
```

In the previous query, the account_name column technically violates the Single-Value Rule, since it's named neither in the GROUP BY clause nor in an aggregate function. Nevertheless, there is only one value possible for account_name in each group; the groups are based on Bugs.reported_by, which is a foreign key to the Accounts table. Therefore, the groups correspond one-to-one with rows in the Accounts table.

In other words, if you know the value of reported_by, then you know the value of account_name unambiguously, like if you had queried by the primary key of the Accounts table.

This kind of unambiguous relationship is called a *functional dependency*. The most common example of this is between the primary key of a table and the table's attributes: account_name is a functional dependency of its primary key, account_id. If you group a query by a table's primary key column(s), then the groups correspond to a single row of that table, and therefore all other columns of the same table must have a single value per group.

Bugs.reported_by has a similar relationship with the dependent attributes of the Accounts table, because it references the primary key of the Accounts table. When the query groups by the reported_by column, which is a foreign key, the attributes of the Accounts table are functionally dependent, and the query result contains no ambiguity.

However, most brands of database still return an error. Not only is this the behavior required by the SQL standard, but it's not too expensive to figure out functional dependencies on the fly.[1] But if you use MySQL or SQLite and you're careful to query only functionally dependent columns, you can use this kind of grouping query and still avoid problems of ambiguity.

---

1. The example queries in this chapter are simple. Figuring out functional dependencies for any arbitrary SQL query is harder.

15.5   Solution: Use Columns Unambiguously

The sections that follow describe several ways you can resolve this antipattern and write unambiguous queries.

Query Only Functionally Dependent Columns

The most straightforward solution is to eliminate ambiguous columns from the query.

Download **Groups/anti/groupbyproduct.sql**

```sql
SELECT product_id, MAX(date_reported) AS latest
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

The query reveals the date of the latest bug per product, even though it doesn't report the bug_id corresponding to that latest bug. Sometimes this is enough, so don't overlook a simple solution.

Using a Correlated Subquery

A correlated subquery contains a reference to the outer query and so produces different results for each row of the outer query. We can use this to find the latest bug per product by running a subquery to search for bugs with the same product and a greater date. When the subquery finds none, the bug in the outer query is the latest.

Download **Groups/soln/notexists.sql**

```sql
SELECT bp1.product_id, b1.date_reported AS latest, b1.bug_id
FROM Bugs b1 JOIN BugsProducts bp1 USING (bug_id)
WHERE NOT EXISTS
  (SELECT * FROM Bugs b2 JOIN BugsProducts bp2 USING (bug_id)
   WHERE bp1.product_id = bp2.product_id
     AND b1.date_reported < b2.date_reported);
```

Use this solution as a simple solution that is readable and easy to code. However, keep in mind that this solution isn't likely to be the best for performance, because correlated subqueries are executed once for each row of the outer query.

Using a Derived Table

You can use a subquery as a *derived table*, producing an interim result that contains only the product_id and the corresponding greatest bug report date for each product. Then use this result to join against the tables so that the query result contains only bugs with the latest date per product.

Download **Groups/soln/derived-table.sql**

```
SELECT m.product_id, m.latest, b1.bug_id
FROM Bugs b1 JOIN BugsProducts bp1 USING (bug_id)
  JOIN (SELECT bp2.product_id, MAX(b2.date_reported) AS latest
   FROM Bugs b2 JOIN BugsProducts bp2 USING (bug_id)
   GROUP BY bp2.product_id) m
  ON (bp1.product_id = m.product_id AND b1.date_reported = m.latest);
```

product_id	latest	bug_id
1	2010-06-01	2248
2	2010-02-16	3456
2	2010-02-16	5150
3	2010-01-01	5678

Notice that you can get multiple rows per product if the latest date returned by the subquery matches multiple rows. If you need to ensure a single row per product_id, you can use another grouping function in the outer query:

Download **Groups/soln/derived-table-no-duplicates.sql**

```
SELECT m.product_id, m.latest, MAX(b1.bug_id) AS latest_bug_id
FROM Bugs b1 JOIN
  (SELECT product_id, MAX(date_reported) AS latest
   FROM Bugs b2 JOIN BugsProducts USING (bug_id)
   GROUP BY product_id) m
  ON (b1.date_reported = m.latest)
GROUP BY m.product_id, m.latest;
```

product_id	latest	latest_bug_id
1	2010-06-01	2248
2	2010-02-16	5150
3	2010-01-01	5678

Use the derived table solution as a more scalable alternative to the correlated subquery. The derived table is noncorrelated, so most database brands should be able to execute the subquery once. However, the database must store the interim result set in a temporary table, so this still isn't the best for performance.

Using a JOIN

You can create a join that tries to match against a set of rows that may not exist. This type of join is called an *outer join*. Where the matching rows don't exist, null is used for all columns in that nonexistent row. So, where the query finds null, we know no such row was found.

Download **Groups/soln/outer-join.sql**

```sql
SELECT bp1.product_id, b1.date_reported AS latest, b1.bug_id
FROM Bugs b1 JOIN BugsProducts bp1 ON (b1.bug_id = bp1.bug_id)
LEFT OUTER JOIN (Bugs AS b2 JOIN BugsProducts AS bp2 ON (b2.bug_id = bp2.bug_id))
  ON (bp1.product_id = bp2.product_id AND (b1.date_reported < b2.date_reported
    OR b1.date_reported = b2.date_reported AND b1.bug_id < b2.bug_id))
WHERE b2.bug_id IS NULL;
```

product_id	latest	bug_id
1	2010-06-01	2248
2	2010-02-16	5150
3	2010-01-01	5678

It takes a few minutes of gazing at this query, and perhaps some doodles on notepaper, for most people to see how it works. But once you do, this technique can be an important tool.

Use the JOIN solution when the scalability of the query over large sets of data is important. Although it's a tougher concept to grasp and therefore more difficult to maintain, it often scales better than a subquery-based solution. Remember to measure the performance of several query forms, instead of assuming that one performs better than the other.

Using an Aggregate Function for Extra Columns

You can make the extra column comply with the Single-Value Rule by applying another aggregate function to it.

Download **Groups/soln/extra-aggregate.sql**

```sql
SELECT product_id, MAX(date_reported) AS latest,
  MAX(bug_id) AS latest_bug_id
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

Use this solution only when you can rely on the latest bug_id being the bug with the latest date, in other words, if bugs are guaranteed to be reported in chronological order.

Concatenating All Values per Group

Finally, you can use another aggregate function on bug_id to avoid violating the Single-Value Rule. MySQL and SQLite support a function GROUP_CONCAT() that concatenates all the values in the group into one value. By default, this is a comma-separated string.

```
SELECT product_id, MAX(date_reported) AS latest
  GROUP_CONCAT(bug_id) AS bug_id_list,
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

product_id	latest	bug_id_list
1	2010-06-01	1234,2248
2	2010-02-16	3456,4077,5150
3	2010-01-01	5678,8063

This query doesn't reveal which bug_id corresponds to the latest date; the bug_id_list includes all bug_id values in each group.

Another disadvantage of this solution is that it isn't standard SQL. and other brands of database don't support this function. Some brands of database support custom functions and custom aggregate functions. For example, here's the solution for PostgreSQL:

```
CREATE AGGREGATE GROUP_ARRAY (
  BASETYPE = ANYELEMENT,
    SFUNC = ARRAY_APPEND,
    STYPE = ANYARRAY,
  INITCOND = '{}'
);

SELECT product_id, MAX(date_reported) AS latest
  ARRAY_TO_STRING(GROUP_ARRAY(bug_id), ',') AS bug_id_list,
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

Some other brands of database don't support custom functions, so the solution may require writing a stored procedure to loop over an non-grouped query result, concatenating values manually.

Use this solution when you expect the extra column to have a single value per group but the column still violates the Single-Value Rule.

Follow the Single-Value Rule to avoid ambiguous query results.

---

*The generation of random numbers is too important to be
left to chance.*
▶ Robert R. Coveyou

Chapter 16

Random Selection

You're writing a web application that displays advertisements. You're
supposed to choose a random ad on each viewing so that all your adver-
tisers have an even chance of showing their ads and so that readers
don't get bored seeing the same ad repeatedly.

Things go well for the first few days, but the application gradually
becomes more sluggish. A few weeks later, people are complaining that
your website is too slow. You discover it's not just psychological; you
can measure a real difference in the page load time. Your readership is
starting to lose interest, and traffic is declining.

Learning from past experiences, you first try to find the performance
bottleneck using profiling tools and a test version of your database with
a sample of the data. You measure the time to load web page, but curi-
ously, there are no problems with the performance in any of the SQL
queries used to produce the page. Yet the production website is getting
slower and slower.

Finally, you realize that the database on your production website is
much greater than the sample in your tests. You repeat your tests with
a database of similar size to the production data and find that it's the
ad-selection query. With a greater number of ads to choose from, the
performance of that query drops sharply. You've discovered the query
that fails to scale, and that's an important first step.

How can you redesign the query that chooses random ads before your
website loses its audience and therefore your sponsors?

16.1 Objective: Fetch a Sample Row

It's surprising how frequently we need an SQL query that returns a random result. This seems to go against the principles of repeatability and deterministic programming. However, it's ordinary to ask for a sample from a large data set. The following are some examples:

- Displaying rotating content, such as an advertisement or a news story to highlight

- Auditing a subset of records

- Assigning incoming calls to available operators

- Generating test data

It's better to query the database for this sample, as an alternative to fetching the entire data set into your application just so you can pick a sample from the set.

The objective is to write an efficient SQL query that returns only a random sample of data.[1]

16.2 Antipattern: Sort Data Randomly

The most common SQL trick to pick a random row from a query is to sort the query randomly and pick the first row. This technique is easy to understand and easy to implement:

```
Download Random/anti/orderby-rand.sql
```
```sql
SELECT * FROM Bugs ORDER BY RAND() LIMIT 1;
```

Although this is a popular solution, it quickly shows its weakness. To understand this weakness, let's first compare it to conventional sorting, in which we compare values in a column and order the rows according to which row has a greater or lesser value in that column. This kind of sort is repeatable, in that it produces the same results when you run it more than once. It also benefits from an index, because an index is essentially a presorted set of the values from a given column.

```
Download Random/anti/indexed-sort.sql
```
```sql
SELECT * FROM Bugs ORDER BY date_reported;
```

---

[1]. Mathematicians and computer scientists make a distinction between truly random and *pseudorandom*. In practice, computers can produce only pseudorandom values.

If your sorting criteria is a function that returns a random value per row, this makes it random whether a given row is greater or less than another row. So, the order has no relation to the values in each row. The order is also different each time you sort in this way. So far so good—this is the result we want.

Sorting by a nondeterministic expression (RAND()) means the sorting cannot benefit from an index. There is no index containing the values returned by the random function. That's the point of them being random: they are different and unpredictable each time they're selected.

This is a problem for the performance of the query, because using an index is one of the best ways of speeding up sorting. The consequence of not using an index is that the query result set has to be sorted by the database "manually." This is called a *table scan*, and it often involves saving the entire result as a temporary table and sorting it by physically swapping rows. A table scan sort is much slower than an index-assisted sort, and the performance difference grows with the size of the data set.

Another weakness of the sort-by-random technique is that after the expensive process of sorting the entire data set, most of that work is wasted because all but the first row is immediately discarded. In a table with a thousand rows, why go to the trouble of randomizing all thousand when all we need is one row?

Both of these problems are unnoticeable when you run the query over a small number of rows, so during development and testing it may appear to be a good solution. But as the volume in your database increases over time, the query fails to scale well.

16.3 How to Recognize the Antipattern

The technique shown in the antipattern is straightforward, and many programmers use it, either after reading it in an article or coming up with it on their own. Some of the following quotes are clues that your colleague is practicing the antipattern:

- "In SQL, returning a random row is really slow."

  The query to select a random sample worked well against trivial data during development and testing, but it gets progressively slower as the real data grows. No amount of database server tuning, indexing, or caching can improve the scalability.

- "How can I increase memory for my application? I need to fetch all the rows so I can randomly pick one."

  You shouldn't have to load all the rows into the application, and it's wildly wasteful to do this. Besides, the database tends to grow larger than your application memory can handle.

- "Does it seem to you like some entries come up more frequently than they should? This randomizer doesn't seem very random."

  Your random numbers are not synchronized with the gaps in primary key values in the database (see Section 16.5, *Choose Next Higher Key Value*, on the following page).

16.4 Legitimate Uses of the Antipattern

The inefficiency of the sort-by-random solution is tolerable if your data set is bound to be small.

For example, you could use a random method for assigning a programmer to fix a given bug. It's safe to assume that you'll never have so many programmers that you need to use a highly scalable method for choosing a random person.

Another example could be selecting a random U.S. state from a list of the 50 states, which is a list of modest size and not likely to grow during our lifetimes.

16.5 Solution: In No Particular Order...

The sort-by-random technique is an example of a query that's bound to perform a table scan and an expensive manual sort. When you design solutions in SQL, you should be on the lookout for inefficient queries like this. Instead of searching fruitlessly for a way to optimize an unoptimizable query, rethink your approach. You can use the alternative techniques shown in the following sections to query a random row from a query result set. In different circumstances, each of these solutions can produce the same result with greater efficiency.

Choose a Random Key Value Between 1 and MAX

One technique that avoids sorting the table is to choose a random value between 1 and the greatest primary key value.

Download **Random/soln/rand-1-to-max.sql**

```sql
SELECT b1.*
FROM Bugs AS b1
JOIN (SELECT CEIL(RAND() * (SELECT MAX(bug_id) FROM Bugs)) AS rand_id) AS b2
  ON (b1.bug_id = b2.rand_id);
```

This solution assumes that primary key values start at 1 and that primary key values are contiguous. That is, there are no values unused between 1 and the greatest value. If there are gaps, a randomly chosen value may not match a row in the table.

Use this solution when you know your key uses all values between 1 and the greatest key value.

Choose Next Higher Key Value

This is similar to the preceding solution, but if you have gaps of unused values between 1 and the greatest key value, this query matches a random value to the first key value it finds.

Download **Random/soln/next-higher.sql**

```sql
SELECT b1.*
FROM Bugs AS b1
JOIN (SELECT CEIL(RAND() * (SELECT MAX(bug_id) FROM Bugs)) AS bug_id) AS b2
WHERE b1.bug_id >= b2.bug_id
ORDER BY b1.bug_id
LIMIT 1;
```

This solves the problem of a random number that misses any key value, but it means that a key value that follows a gap is chosen more often. Random values should be approximately even in distribution, but bug_id values aren't.

Use this solution when gaps are uncommon and when it's not important for all key values to be chosen with equal frequency.

Get a List of All Key Values, Choose One at Random

You can use application code to pick one value from the primary keys in the result set. Then query the full row from the database using that primary key. This technique is shown in the following PHP code:

Download **Random/soln/rand-key-from-list.php**

```php
<?php
$bug_id_list = $pdo->query("SELECT bug_id FROM Bugs")->fetchAll();

$rand = random( count($bug_id_list) );
$rand_bug_id = $bug_id_list[$rand]["bug_id"];
```

```
$stmt = $pdo->prepare("SELECT * FROM Bugs WHERE bug_id = ?");
$stmt->execute( array($rand_bug_id) );
$rand_bug = $stmt->fetch();
```

This avoids sorting the table, and the chance of choosing each key value is approximately equal, but this solution has other costs:

- Fetching all the bug_id values from the database might return a list of impractical size. It can even exceed application memory resources and cause an error such as the following:

  ```
  Fatal error:  Allowed memory size of 16777216 bytes exhausted
  ```

- The query must be run twice: once to produce the list of primary keys and a second time to fetch the random row. If the query is too complex and costly, this is a problem.

Use this solution when you're selecting a random row from a simple query with a moderately sized result set. This solution is good for choosing from a list of noncontiguous values.

Choose a Random Row Using an Offset

Still another technique that avoids problems found in the preceding alternatives is to count the rows in the data set and return a random number between 0 and the count. Then use this number as an offset when querying the data set.

Download Random/soln/limit-offset.php

```php
<?php

$rand = "SELECT ROUND(RAND() * (SELECT COUNT(*) FROM Bugs))";
$offset = $pdo->query($rand)->fetch(PDO::FETCH_ASSOC);

$sql = "SELECT * FROM Bugs LIMIT 1 OFFSET :offset";
$stmt = $pdo->prepare($sql);
$stmt->execute( $offset );
$rand_bug = $stmt->fetch();
```

This solution relies on the nonstandard LIMIT clause, supported by My-SQL, PostgreSQL, and SQLite.

An alternative that uses the ROW_NUMBER() window function works in Oracle, Microsoft SQL Server, and IBM DB2.

For example, here's the solution in Oracle:

```php
<?php
$rand = "SELECT 1 + MOD(ABS(dbms_random.random()),
  (SELECT COUNT(*) FROM Bugs)) AS offset FROM dual";
$offset = $pdo->query($rand)->fetch(PDO::FETCH_ASSOC);

$sql = "WITH NumberedBugs AS (
  SELECT b.*, ROW_NUMBER() OVER (ORDER BY bug_id) AS RN FROM Bugs b
) SELECT * FROM NumberedBugs WHERE RN = :offset";
$stmt = $pdo->prepare($sql);
$stmt->execute( $offset );
$rand_bug = $stmt->fetch();
```

Use this solution when you can't assume contiguous key values and you need to make sure each row has an even chance of being selected.

Proprietary Solutions

Any given brand of database might implement its own solution for this kind of task. For example, Microsoft SQL Server 2005 added a TABLE-SAMPLE clause:

```sql
SELECT * FROM Bugs TABLESAMPLE (1 ROWS);
```

Oracle uses a slightly different SAMPLE clause, for example to return 1 percent of the rows in the table:

```sql
SELECT * FROM (SELECT * FROM Bugs SAMPLE (1)
ORDER BY dbms_random.value) WHERE ROWNUM = 1;
```

You should read the documentation for the proprietary solution in your brand of database. There are often limitations or other options you need to know about.

Some queries cannot be optimized; take a different approach.

Some people, when confronted with a problem,
think "I know, I'll use regular expressions."
Now they have two problems.
   ▶ Jamie Zawinski

Chapter 17

Poor Man's Search Engine

I was working in a technical support job in 1995, at a time when companies were just starting to adopt the Web as a way to provide information to their customers. We had a collection of short documents describing solutions to common support questions, and we wanted to put them on the Web in a knowledge-base application.

We quickly realized that as the collection grew, it needed to be searchable, because customers didn't want to browse through hundreds of articles to find their answers. One strategy would be to organize the articles in categories, but even these groups were too large, and many articles belonged in multiple groups.

We wanted our customers to search the articles, narrowing down the list to those matching any criteria. The most flexible and straightforward interface was to allow the customer to enter any set of words and show them the articles in which those words appear. An article was weighted higher if it matched the search terms more fully. Also, we wanted to match word forms. For example, a search for the word *crash* should also match *crashed*, *crashes*, and *crashing*. Of course, the search had to work in a growing collection of documents quickly enough to be useful in a web application.

If that careful description sounds superfluous to you, that shouldn't be surprising. Searching through text online has become so common that we can hardly recall the time before it was available. But using SQL to search by keywords, while also making the solution both fast and accurate, is deceptively difficult.

17.1 Objective: Full-Text Search

Any application that stores text needs to search for words or phrases within that text. We use databases to store more textual data than ever, and at the same time we demand to be able to search for matching text at greater speeds. Web applications especially need high-performance and scalable database techniques for searching text.

One fundamental principle of SQL (and relational theory from which SQL is derived) is that a value in a column is *atomic*. That is, you can compare a value to another value, but you always compare the *whole* value when you do that. Comparing substrings is bound to be inefficient or inaccurate in SQL.

In spite of this, we need a way to compare a short string to a longer string and find a match when the short string occurs anywhere within the longer string. How can we bridge this gulf using SQL?

17.2 Antipattern: Pattern Matching Predicates

SQL provides pattern-matching predicates for comparing strings, and this is the first solution most programmers use when searching for key words. The most widely supported of these is the LIKE predicate.

The LIKE predicate supports a wildcard (%) that matches zero or more characters. Using this wildcard before and after a key word matches any string that contains that word. The first wildcard matches any text preceding the word, and the second wildcard matches any text following the word.

Download **Search/anti/like.sql**

```
SELECT * FROM Bugs WHERE description LIKE '%crash%';
```

Regular expressions are also supported by many database brands, although not in a standard way. You don't need wildcards, because conventionally regular expressions match the pattern against any substring anyway. Here's an example using MySQL's regular expression predicate:[1]

Download **Search/anti/regexp.sql**

```
SELECT * FROM Bugs WHERE description REGEXP 'crash';
```

---

1. Although SQL-99 defines the predicate SIMILAR TO for matching regular expressions, most brands of SQL database use nonstandard syntax.

The most important disadvantage of pattern-matching operators is that they have poor performance. They can't benefit from a conventional index, so they must scan every row in a table. Since matching a pattern against a string column is a fairly expensive operation (relative to, for instance, comparing two integers for equality), the total cost of a table scan for this search is very high.

A second problem of simple pattern-matching using LIKE or regular expressions is that it can find unintended matches.

Download **Search/anti/like-false-match.sql**

```sql
SELECT * FROM Bugs WHERE description LIKE '%one%';
```

The previous example matches text that contains the words *one*, but it also matches strings *money*, *prone*, *lonely*, and so on. Searching for a pattern with the key word delimited by spaces doesn't match occurrences of the word with punctuation or at the start or end of the text. The regular expressions supported by your database might support a special pattern for a *word boundary*, to solve this issue:[2]

Download **Search/anti/regexp-word.sql**

```sql
SELECT * FROM Bugs WHERE description REGEXP '[[:<:]]one[[:>:]]';
```

Given the problems of performance and scalability and the gymnastics you have to do to prevent irrelevant matches, simple pattern matching is a poor technique for searching for key words.

17.3 How to Recognize the Antipattern

Some questions like the following commonly indicate that the Poor Man's Search Engine antipattern is being employed:

- "How do I insert a variable in between two wildcards in a LIKE expression?"

  The question usually comes up when the programmer wants to do a pattern-matching search using input from a user.

- "How can I write a regular expression to check that a string contains multiple words, that the string *doesn't* contain a certain word, or that the string contains any form of a given word?"

---

2. This example uses MySQL syntax.

If a complex problem seems too hard to solve with a regular expression, it probably is.

- "The search feature of our website has become unusably slow as we've added more documents to the database. What's wrong?"

  As the volume of data goes up, the antipattern solution shows poor scalability.

17.4 Legitimate Uses of the Antipattern

The expressions shown in the antipattern section are legal SQL queries, and they have a straightforward and simple usage. That counts for a lot.

Performance is often important, but some queries are run so infrequently that it doesn't make sense to invest a lot of resources to optimize them. Maintaining indexes to benefit a rarely used query could be just as costly as running that query in an inefficient manner. If the nature of the query is ad hoc, there's no guarantee that the index you defined would benefit that given query anyway.

It's hard to use pattern-matching operators for complex queries, but if you design the patterns for simple cases, they can help you get the right results with a minimum of fuss.

17.5 Solution: Use the Right Tool for the Job

It's best to use a specialized search engine technology, instead of SQL. Another alternative is to reduce the recurring cost of search by saving the result.

The following sections describe some of the technologies offered as built-in extensions by different database brands and also technologies offered by independent projects. Also, we'll develop a solution that uses standard SQL but is more efficient on average than substring matching.

Vendor Extensions

Every major brand of database has invented their own answer to the common requirement of full-text search, but these features are not standard or compatible between database brands. If you use a single brand (or are willing to use vendor-dependent features), these features are the best way to get high-performance text search, with the greatest integration with SQL queries.

The following are brief descriptions of full-text search features in several brands of SQL database. The details are subject to change, so be sure to read the current documentation for your brand.

Full-Text Index in MySQL

MySQL provides a simple full-text index type for the MyISAM storage engine only. You can define a full-text index over columns of type CHAR, VARCHAR, or TEXT. Here's an example that defines a full-text index that includes content from the bug summary and description columns:

Download Search/soln/mysql/alter-table.sql

```
ALTER TABLE Bugs ADD FULLTEXT INDEX bugfts (summary, description);
```

Use the MATCH() function to search for a key word among the indexed text. You must name the columns in the full-text index (so you can match using another index that covers different columns in the same table).

Download Search/soln/mysql/match.sql

```
SELECT * FROM Bugs WHERE MATCH(summary, description) AGAINST ('crash');
```

Since MySQL 4.1, you can also use a simple boolean expression notation in the pattern to filter results more carefully.

Download Search/soln/mysql/match-boolean.sql

```
SELECT * FROM Bugs WHERE MATCH(summary, description)
  AGAINST ('+crash -save' IN BOOLEAN MODE);
```

Text Indexing in Oracle

Oracle has supported text-indexing features since Oracle 8 in 1997, when it was part of a data cartridge called ConText. The technology has been updated several times, and the feature is now integrated into the database software. The text indexing in Oracle is complex and rich, so here is a greatly simplified summary:

- CONTEXT

  Create an index of this type for a single text column. Use the CONTAINS() operator to search using this index. The index doesn't stay consistent with changes to data, so you have to rebuild the index manually or on a schedule.

  Download Search/soln/oracle/create-index.sql

```
CREATE INDEX BugsText ON Bugs(summary) INDEXTYPE IS CTSSYS.CONTEXT;

SELECT * FROM Bugs WHERE CONTAINS(summary, 'crash') > 0;
```

- CTXCAT

  This index type is specialized for short text samples such as those
  used in online catalogs, along with other structured columns from
  the same table. The index stays consistent as transactions update
  the indexed data.

  Download Search/soln/oracle/ctxcat-create.sql

  ```
  CTX_DDL.CREATE_INDEX_SET('BugsCatalogSet');
  CTX_DDL.ADD_INDEX('BugsCatalogSet', 'status');
  CTX_DDL.ADD_INDEX('BugsCatalogSet', 'priority');

  CREATE INDEX BugsCatalog ON Bugs(summary) INDEXTYPE IS CTSSYS.CTXCAT
    PARAMETERS('BugsCatalogSet');
  ```

  The CATSEARCH() operator takes two arguments for searching the
  text column and the structured column set, respectively.

  Download Search/soln/oracle/ctxcat-search.sql

  ```
  SELECT * FROM Bugs
  WHERE CATSEARCH(summary, '(crash save)', 'status = "NEW"') > 0;
  ```

- CTXXPATH

  This index type is specialized for searching an XML document with
  the existsNode() operator.

  Download Search/soln/oracle/ctxxpath.sql

  ```
  CREATE INDEX BugTestXml ON Bugs(testoutput) INDEXTYPE IS CTSSYS.CTXXPATH;

  SELECT * FROM Bugs
  WHERE testoutput.existsNode('/testsuite/test[@status="fail"]') > 0;
  ```

- CTXRULE

  Suppose you have a large collection of documents in your data-
  base and you need to classify them based on their content.

  With the CTXRULE index, you can design rules to analyze docu-
  ments and report their classification. Alternatively, you can pro-
  vide a sample set of documents with your idea of their classifica-
  tions and have Oracle design the rules to apply to the rest of the
  document collection. You can even fully automate the process, let-
  ting Oracle analyze your document collection and come up with a
  set of rules and classifications for identifying them.

  Examples using CTXRULE indexes are beyond the scope of this book.

Full-Text Search in Microsoft SQL Server

SQL Server 2000 and later support full-text searching, with complex configuration options for languages, a thesaurus, and automatic synchronization with data changes. SQL Server provides a series of stored procedures for creating full-text indexes, and you can use the CONTAINS() operator in queries to employ the full-text index.

To perform the familiar example of searching for bugs that include the word *crash*, first enable the full-text feature, and define a catalog in your database:

Download **Search/soln/microsoft/catalog.sql**

```
EXEC sp_fulltext_database 'enable'
EXEC sp_fulltext_catalog 'BugsCatalog', 'create'
```

Next, define a full-text index on the Bugs table, add columns to the index, and activate the index:

Download **Search/soln/microsoft/create-index.sql**

```
EXEC sp_fulltext_table 'Bugs', 'create', 'BugsCatalog', 'bug_id'
EXEC sp_fulltext_column 'Bugs', 'summary', 'add', '2057'
EXEC sp_fulltext_column 'Bugs', 'description', 'add', '2057'
EXEC sp_fulltext_table 'Bugs', 'activate'
```

Enable automatic synchronization for the full-text index so that changes to the indexed column are propagated to the index. Then begin the process of populating the index. This will run in the background, so it may take some time to complete before queries benefit from the index.

Download **Search/soln/microsoft/start.sql**

```
EXEC sp_fulltext_table 'Bugs', 'start_change_tracking'
EXEC sp_fulltext_table 'Bugs', 'start_background_updateindex'
EXEC sp_fulltext_table 'Bugs', 'start_full'
```

Finally, run a query using the CONTAINS() operator:

Download **Search/soln/microsoft/search.sql**

```
SELECT * FROM Bugs WHERE CONTAINS(summary, '"crash"');
```

Text Search in PostgreSQL

PostgreSQL 8.3 provides a sophisticated and highly configurable way of converting text into a searchable collections of lexical elements and matching these documents against patterns.

To get the best benefit of performance, you need to store content as its original text form and also as a searchable form using the special data type TSVECTOR.

Download **Search/soln/postgresql/create-table.sql**

```sql
CREATE TABLE Bugs (
  bug_id SERIAL PRIMARY KEY,
  summary      VARCHAR(80),
  description  TEXT,
  ts_bugtext   TSVECTOR
  -- other columns
);
```

Make sure the TSVECTOR column is kept in sync with the content in the text column(s) you want to make searchable. PostgreSQL provides a built-in trigger procedure to make this easier:

Download **Search/soln/postgresql/trigger.sql**

```sql
CREATE TRIGGER ts_bugtext BEFORE INSERT OR UPDATE ON Bugs
FOR EACH ROW EXECUTE PROCEDURE
  tsvector_update_trigger(ts_bugtext, 'pg_catalog.english', summary, description);
```

You should also create a *generalized inverted index* (*GIN*) index on the TSVECTOR column:

Download **Search/soln/postgresql/create-index.sql**

```sql
CREATE INDEX bugs_ts ON Bugs USING GIN(ts_bugtext);
```

After this, you can use the PostgreSQL text search operator @@ to search efficiently, aided by the full-text index:

Download **Search/soln/postgresql/search.sql**

```sql
SELECT * FROM Bugs WHERE ts_bugtext @@ to_tsquery('crash');
```

There are many other options for customizing searchable content, search queries, and search results.

Full-Text Search (FTS) in SQLite

Standard tables in SQLite don't support efficient full-text searches, but you can use an optional extension for SQLite to store searchable text in a *virtual table* specialized for searching text. Three versions of the searchable text extension exist, known as FTS1, FTS2, and FTS3.

FTS extensions are not typically enabled in a default build of SQLite, so you need to build it from source with one of the FTS extensions enabled. For example, add the following options to Makefile.in, and then build SQLite.

Download **Search/soln/sqlite/makefile.in**

```
TCC += -DSQLITE_CORE=1
TCC += -DSQLITE_ENABLE_FTS3=1
```

Once you have a version of SQLite with FTS enabled, you can create a virtual table for the searchable text. Any data type, constraints, or other column options are ignored.

Download **Search/soln/sqlite/create-table.sql**

```
CREATE VIRTUAL TABLE BugsText USING fts3(summary, description);
```

If you are indexing text from another table (as in this example using the Bugs table), you must copy the data into the virtual table. The FTS virtual table always contains a primary key column called docid, so you can correlate rows to those in a source table.

Download **Search/soln/sqlite/insert.sql**

```
INSERT INTO BugsText (docid, summary, description)
  SELECT bug_id, summary, description FROM Bugs;
```

Now you can query the FTS virtual table BugsText using the efficient full-text search predicate MATCH, and you can join matching rows to the source table Bugs. Using the name of the FTS table as a pseudocolumn matches the pattern against any column.

Download **Search/soln/sqlite/search.sql**

```
SELECT b.* FROM BugsText t JOIN Bugs b ON (t.docid = b.bug_id)
WHERE BugsText MATCH 'crash';
```

The matching pattern also supports limited boolean expressions.

Download **Search/soln/sqlite/search-boolean.sql**

```
SELECT * FROM BugsText WHERE BugsText MATCH 'crash -save';
```

Third-Party Search Engines

If you need to search text in a way that works the same regardless of which database brand you use, you need a search engine that runs independently from the SQL database. This section briefly describes two such products, Sphinx Search and Apache Lucene.

Sphinx Search

Sphinx Search (http://www.sphinxsearch.com/) is an open source search engine technology that integrates well with MySQL and PostgreSQL. As of this writing, an unofficial patch exists for using Sphinx Search with

the open source Firebird database. Perhaps in the future this search engine will support other databases.

Indexing and searching is fast in Sphinx Search, and it supports distributed queries as well. It's a good choice for high-scale searching applications that have data that updates infrequently.

You can use Sphinx Search to index data stored in a MySQL database. By modifying a few fields in a configuration file sphinx.conf, you can specify the database. You must also write an SQL query to fetch data for building the index. The first column in this query is the integer primary key. You may declare some columns as attributes for restricting or sorting results. The remaining columns are those to be full-text indexed. Finally, another SQL query fetches a full row from the database given a primary key value coded as $id.

Download Search/soln/sphinx/sphinx.conf

```
source bugsrc
{
  type                = mysql
  sql_user            = bugsuser
  sql_pass            = xyzzy
  sql_db              = bugsdatabase
  sql_query           = \
    SELECT bug_id, status, date_reported, summary, description \
    FROM Bugs
  sql_attr_timestamp  = date_reported
  sql_attr_str2ordinal = status
  sql_query_info      = SELECT * FROM Bugs WHERE bug_id = $id
}

index bugs
{
  source              = bugsrc
  path                = /opt/local/var/db/sphinx/bugs
}
```

Once you declare this configuration in sphinx.conf, you can create the index at the shell with the indexer command:

Download Search/soln/sphinx/indexer.sh

```
indexer -c sphinx.conf bugs
```

You can search the index using the search command:

Download Search/soln/sphinx/search.sh

```
search -b "crash -save"
```

Sphinx Search also has a daemon process and an API with which to invoke searches from popular scripting languages such as PHP, Perl, and Ruby. The major disadvantage of the current software is that the index algorithm doesn't support incremental updates efficiently. Using Sphinx Search over a data source that updates frequently requires some compromises. For example, split your searchable table into two tables, the first to store the majority of historical data that doesn't change and the second to store a smaller set of current data that grows incrementally and must be reindexed. Then your application must search two Sphinx Search indexes.

Apache Lucene

Lucene (http://lucene.apache.org/) is a mature search engine for Java applications. Work-alike projects exist for other languages including C++, C#, Perl, Python, Ruby, and PHP.

Lucene builds an index in its proprietary format for a collection of text *documents*. The Lucene index doesn't stay in sync with the source data it indexes. If you insert, delete, or update rows in the database, you must apply matching changes to a Lucene index.

Using the Lucene search engine is a bit like using a car engine; you need quite a bit of supporting technology around it to make it useful. Lucene doesn't read data collections from an SQL database directly. you have to write documents in the Lucene index. For example, you could run an SQL query and, for each row of the result, create one Lucene document and save it to the Lucene index. You can use Lucene through its Java API.

Fortunately, Apache also offers a complementary project called Solr (http://lucene.apache.org/solr/. Solr is a server that provides a gateway to a Lucene index. You can add documents to Solr and submit search queries using a REST-like interface so you can use it from any programming language.

You can also direct Solr to connect to a database itself, run a query, and index the results using its DataImportHandler tool.

Roll Your Own

Suppose you don't want to use proprietary search features, nor do you want to install an independent search engine product. You need an efficient, database-independent solution to make text searchable. In this section, we design what's called an *inverted index*. Basically, an

inverted index is a list of all words one might search for. In a many-to-many relationship, the index associates these words with the text entries that contain the respective word. That is, a word like *crash* can appear in many bugs, and each bug may match many other keywords. This section shows how to design an inverted index.

First, define a table Keywords to list keywords for which users search, and define an intersection table BugsKeywords to establish a many-to-many relationship:

Download Search/soln/inverted-index/create-table.sql

```sql
CREATE TABLE Keywords (
  keyword_id  SERIAL PRIMARY KEY,
  keyword     VARCHAR(40) NOT NULL,
  UNIQUE KEY (keyword)
);

CREATE TABLE BugsKeywords (
  keyword_id  BIGINT UNSIGNED NOT NULL,
  bug_id      BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (keyword_id, bug_id),
  FOREIGN KEY (keyword_id) REFERENCES Keywords(keyword_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

Next, add a row to BugsKeywords for every keyword that matches the description text for a given bug. We can use substring-match query to determine these matches using LIKE or regular expressions. This is nothing more costly than the naive searching method described in the "Antipattern" section, but we gain efficiency because we only need to perform the search once. If we save the result in the intersection table, all subsequent searches for the same keyword are much faster.

Next, we write a stored procedure to make it easier to search for a given keyword.[3] If the word has already been searched, the query is faster because the rows in BugsKeywords are a list of the documents that contain the keyword. If no one has searched for the given keyword before, we need to search the collection of text entries the hard way.

Download Search/soln/inverted-index/search-proc.sql

```sql
CREATE PROCEDURE BugsSearch(keyword VARCHAR(40))
BEGIN
  SET @keyword = keyword;
```

---

3. This example stored procedure uses MySQL syntax.

```
❶     PREPARE s1 FROM 'SELECT MAX(keyword_id) INTO @k FROM Keywords
        WHERE keyword = ?';
      EXECUTE s1 USING @keyword;
      DEALLOCATE PREPARE s1;

      IF (@k IS NULL) THEN
❷       PREPARE s2 FROM 'INSERT INTO Keywords (keyword) VALUES (?)';
        EXECUTE s2 USING @keyword;
        DEALLOCATE PREPARE s2;
❸       SELECT LAST_INSERT_ID() INTO @k;

❹       PREPARE s3 FROM 'INSERT INTO BugsKeywords (bug_id, keyword_id)
          SELECT bug_id, ? FROM Bugs
          WHERE summary REGEXP CONCAT(''[[:<:]]'', ?, ''[[:>:]]'')
            OR description REGEXP CONCAT(''[[:<:]]'', ?, ''[[:>]]'')';
        EXECUTE s3 USING @k, @keyword, @keyword;
        DEALLOCATE PREPARE s3;
      END IF;

❺     PREPARE s4 FROM 'SELECT b.* FROM Bugs b
        JOIN BugsKeywords k USING (bug_id)
        WHERE k.keyword_id = ?';
      EXECUTE s4 USING @k;
      DEALLOCATE PREPARE s4;
    END
```

❶ Search for the user-specified keyword. Return either the integer primary key from Keywords.keyword_id or null if the word has not been seen previously.

❷ If the word was not found, insert it as a new word.

❸ Query for the primary key value generated in Keywords.

❹ Populate the intersection table by searching Bugs for rows containing the new keyword.

❺ Finally, query the full rows from Bugs that match the keyword_id, whether the keyword was found or had to be inserted as a new entry.

Now we can call this stored procedure and pass the desired keyword. The procedure returns the set of matching bugs, whether it has to calculate the matching bugs and populate the intersection table for a new keyword or whether it simply benefits from the result of an earlier search.

Download Search/soln/inverted-index/search-proc.sql

```
CALL BugsSearch('crash');
```

There's another piece to this solution: we need to define a trigger to populate the intersection table as each new bug is inserted. If you need to support edits to bug descriptions, you may also have to write a trigger to reanalyze text and add or delete rows in the BugsKeywords table.

Download Search/soln/inverted-index/trigger.sql

```
CREATE TRIGGER Bugs_Insert AFTER INSERT ON Bugs
FOR EACH ROW
BEGIN
  INSERT INTO BugsKeywords (bug_id, keyword_id)
    SELECT NEW.bug_id, k.keyword_id FROM Keywords k
    WHERE NEW.description REGEXP CONCAT('[[:<:]]', k.keyword, '[[:>:]]')
          OR NEW.summary REGEXP CONCAT('[[:<:]]', k.keyword, '[[:>:]]');
END
```

The keyword list is populated naturally as users perform searches, so we don't need to fill the keyword list with every word found in the knowledge-base articles. On the other hand, if we can anticipate likely keywords, we can easily run a search for them, thus bearing the initial cost of being the first to search for each keyword so that doesn't fall on our users.

I used an inverted index for my knowledge-base application that I described at the start of this chapter. I also enhanced the Keywords table with an additional column num_searches. I incremented this column each time a user searched for a given keyword so I could track which searches were most in demand.

You don't have to use SQL to solve every problem.

---

*Enita non sunt multiplicanda praeter necessitatem
(Latin, "entities are not to be multiplied beyond necessity").*
▶ William of Ockham

Chapter 18

Spaghetti Query

Your boss is on the phone with his boss, and he waves to you to come over. He covers his phone receiver with his hand and whispers to you, "The executives are in a budget meeting, and we're going to have our staff cut unless we can feed my VP some statistics to prove that our department keeps a lot of people busy. I need to know how many products we work on, how many developers fixed bugs, the average bugs fixed per developer, and how many of our fixed bugs were reported by customers. Right now!"

You leap to your SQL tool and start writing. You want all the answers at once, so you make one complex query, hoping to do the least amount of duplicate work and therefore produce the results faster.

Download Spaghetti-Query/intro/report.sql

```sql
SELECT COUNT(bp.product_id) AS how_many_products,
  COUNT(dev.account_id) AS how_many_developers,
  COUNT(b.bug_id)/COUNT(dev.account_id) AS avg_bugs_per_developer,
  COUNT(cust.account_id) AS how_many_customers
FROM Bugs b JOIN BugsProducts bp ON (b.bug_id = bp.bug_id)
JOIN Accounts dev ON (b.assigned_to = dev.account_id)
JOIN Accounts cust ON (b.reported_by = cust.account_id)
WHERE cust.email NOT LIKE '%@example.com'
GROUP BY bp.product_id;
```

The numbers come back, but they seem wrong. How did we get dozens of products? How can the average bugs fixed be exactly 1.0? And it wasn't the number of customers; it was the number of bugs reported by customers that your boss needs. How can all the numbers be so far off? This query will be a lot more complex than you thought.

Your boss hangs up the phone. "Never mind," he sighs. "It's too late. Let's clean out our desks."

18.1 Objective: Decrease SQL Queries

One of the most common places where SQL programmers get stuck is when they ask, "How can I do this with a single query?" This question is asked for virtually any task. Programmers have been trained that one SQL query is difficult, complex, and expensive, so they reason that two SQL queries must be twice as bad. More than two SQL queries to solve a problem is generally out of the question.

Programmers can't reduce the complexity of their tasks, but they want to simplify the solution. They state their goal with terms like "elegant" or "efficient," and they think they've achieved those goals by solving the task with a single query.

18.2 Antipattern: Solve a Complex Problem in One Step

SQL is a very expressive language—you can accomplish a lot in a single query or statement. But that doesn't mean it's mandatory or even a good idea to approach every task with the assumption it has to be done in one line of code. Do you have this habit with any other programming language you use? Probably not.

Unintended Products

One common consequence of producing all your results in one query is a *Cartesian product*. This happens when two of the tables in the query have no condition restricting their relationship. Without such a restriction, the join of two tables pairs each row in the first table to *every* row in the other table. Each such pairing becomes a row of the result set, and you end up with many more rows than you expect.

Let's see an example. Suppose we want to query our bugs database to count the number of bugs fixed, and the number of bugs open, for a given product. Many programmers would try to use a query like the following to calculate these counts:

Download Spaghetti-Query/anti/cartesian.sql

```
SELECT p.product_id,
  COUNT(f.bug_id) AS count_fixed,
  COUNT(o.bug_id) AS count_open
FROM BugsProducts p
LEFT OUTER JOIN Bugs f ON (p.bug_id = f.bug_id AND f.status = 'FIXED')
LEFT OUTER JOIN Bugs o ON (p.bug_id = o.bug_id AND o.status = 'OPEN')
WHERE p.product_id = 1
GROUP BY p.product_id;
```

bug_id	status
1234	FIXED
3456	FIXED
4567	FIXED
5678	FIXED
6789	FIXED
7890	FIXED
8901	FIXED
9012	FIXED
10123	FIXED
11234	FIXED
12345	FIXED

bug_id	status
4077	OPEN
8063	OPEN
5150	OPEN
867	OPEN
5309	OPEN
6060	OPEN
842	OPEN
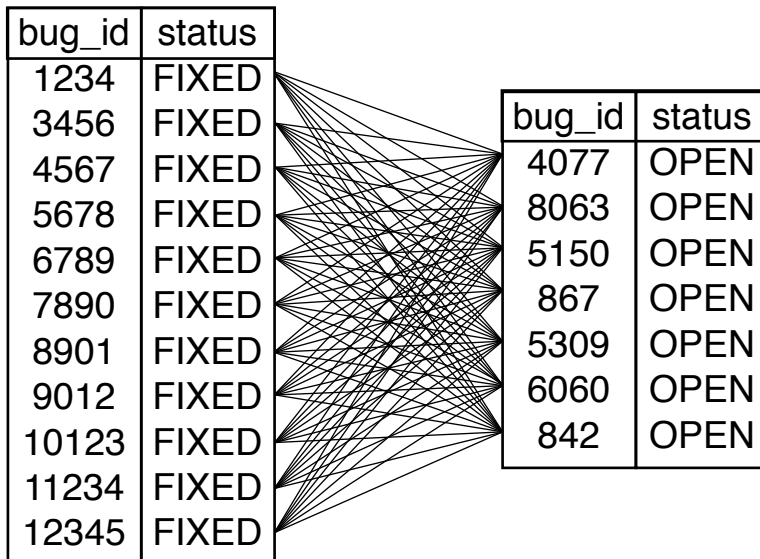
Figure 18.1: Cartesian product between fixed and open bugs

You happen to know that in reality there are twelve fixed bugs and seven open bugs for the given product. So, the result of the query is puzzling:

product_id	count_fixed	count_open
1	84	84

What caused this to be so inaccurate? It's no coincidence that 84 is 12 times 7. This example joins the Products table to two different subsets of Bugs, but this results in a Cartesian product between those two sets of bugs. Each of the twelve rows for *FIXED* bugs is paired with all seven rows for *OPEN* bugs.

You can visualize the Cartesian product graphically as shown in Figure 18.1. Each line connecting a fixed bug to an open bug becomes a row in the interim result set (before grouping is applied). We can see this interim result set by eliminating the GROUP BY clause and aggregate functions.

```
SELECT p.product_id, f.bug_id AS fixed, o.bug_id AS open
FROM BugsProducts p
JOIN Bugs f ON (p.bug_id = f.bug_id AND f.status = 'FIXED')
JOIN Bugs o ON (p.bug_id = o.bug_id AND o.status = 'OPEN')
WHERE p.product_id = 1;
```

The only relationships expressed in that query are between the Bugs-Products table and each subset of Bugs. No conditions restrict every *FIXED* bug from matching with every *OPEN* bug, and the default is that they do. The result produces twelve times seven rows.

It's all too easy to produce an unintentional Cartesian product when you try to make a query do double-duty like this. If you try to do more unrelated tasks with a single query, the total could be multiplied by yet another Cartesian product.

As Though That Weren't Enough. . .

Besides the fact that you can get the wrong results, it's important to consider that these queries are simply hard to write, hard to modify, and hard to debug. You should expect to get regular requests for incremental enhancements to your database applications. Managers want more complex reports and more fields in a user interface. If you design intricate, monolithic SQL queries, it's more costly and time-consuming to make enhancements to them. Your time is worth something, both to you and to your project.

There are runtime costs, too. An elaborate SQL query that has to use many joins, correlated subqueries, and other operations is harder for the SQL engine to optimize and execute quickly than a more straight-forward query. Programmers have an instinct that executing fewer SQL queries is better for performance. This is true assuming the SQL queries in question are of equal complexity. On the other hand, the cost of a single monster query can increase exponentially, until it's much more economical to use several simpler queries.

18.3   How to Recognize the Antipattern

If you hear the following statements from members of your project, it could indicate a case of the Spaghetti Query antipattern:

- "Why are my sums and counts impossibly large?"

    An unintended Cartesian product has multiplied two different joined data sets.

- "I've been working on this monster SQL query all day!"

  SQL isn't this difficult—really. If you've been struggling with a single query for too long, you should reconsider your approach.

- "We can't add anything to our database report, because it will take too long to figure out how to recode the SQL query."

  The person who coded the query will be responsible for maintaining that code forever, even if they have moved on to other projects. That person could be you, so don't write overly complex SQL that no one else can maintain!

- "Try putting another DISTINCT into the query."

  Compensating for the explosion of rows in a Cartesian product, programmers reduce duplicates using the DISTINCT keyword as a query modifier or an aggregate function modifier. This hides the evidence of the malformed query but causes extra work for the RDBMS to generate the interim result set only to sort it and discard duplicates.

Another clue that a query might be a Spaghetti Query is simply that it has an excessively long execution time. Poor performance could be symptomatic of other causes, but as you investigate such a query, you should consider that you may be trying to do too much in a single SQL statement.

18.4   Legitimate Uses of the Antipattern

The most common reason that you might need to run a complex task with a single query is that you're using a programming framework or a visual component library that connects to a data source and presents data in an application. Simple business intelligence and reporting tools also fall into this category, although more sophisticated BI software can merge results from multiple data sources.

A component or reporting tool that assumes its data source is a single SQL query may have a simpler usage, but it encourages you to design monolithic queries to synthesize all the data for your report. If you use one of these reporting applications, you may be forced to write a more complex SQL query than if you had the opportunity to write code to process the result set.

If the reporting requirements are too complex to be satisfied by a single SQL query, it might be better to produce multiple reports. If your boss

doesn't like this, remind him or her of the relationship between the report's complexity and the hours it takes to produce it.

Sometimes, you may want to produce a complex result in one query because you need all the results combined in sorted order. It's easy to specify a sort order in an SQL query. It's likely to be more efficient for the database to do that than for you to write custom code in your application to sort the results of several queries.

18.5 Solution: Divide and Conquer

The quote from William of Ockham at the beginning of this chapter is also known as the *law of parsimony*:

> **The Law of Parsimony**
>
> When you have two competing theories that make exactly the same predictions, the simpler one is the better.

What this means to SQL is that when you have a choice between two queries that produce the same result set, choose the simpler one. We should keep this in mind when straightening out instances of this antipattern.

One Step at a Time

If you can't see a logical join condition between the tables involved in an unintended Cartesian product, that could be because there simply is no such condition. To avoid the Cartesian product, you have to split up a Spaghetti Query into several simpler queries. In the simple example shown earlier, we need only two queries:

Download Spaghetti-Query/soln/split-query.sql

```sql
SELECT p.product_id, COUNT(f.bug_id) AS count_fixed
FROM BugsProducts p
LEFT OUTER JOIN Bugs f ON (p.bug_id = f.bug_id AND f.status = 'FIXED')
WHERE p.product_id = 1
GROUP BY p.product_id;

SELECT p.product_id, COUNT(o.bug_id) AS count_open
FROM BugsProducts p
LEFT OUTER JOIN Bugs o ON (p.bug_id = o.bug_id AND o.status = 'OPEN')
WHERE p.product_id = 1
GROUP BY p.product_id;
```

The results of these two queries are 12 and 7, as expected.

You may feel slight regret at resorting to an "inelegant" solution by splitting this into multiple queries, but this should quickly be replaced by relief as you realize this has several positive effects for development, maintenance, and performance:

- The query doesn't produce an unwanted Cartesian product, as shown in the earlier examples, so it's easier to be sure the query is giving you accurate results.

- When new requirements are added to the report, it's easier to add another simple query than to integrate more calculations into an already-complicated query.

- The SQL engine can usually optimize and execute a simple query more easily and reliably than a complex query. Even if it seems like the work is duplicated by splitting the query, it may nevertheless be a net win.

- In a code review or a teammate training session, it's easier to explain how several straightforward queries work than to explain one intricate query.

Look for the UNION Label

You can combine the results of several queries into one result set with the UNION operation. This can be useful if you really want to submit a single query and consume a single result set, for instance because the result needs to be sorted.

Download Spaghetti-Query/soln/union.sql
```sql
(SELECT p.product_id, f.status, COUNT(f.bug_id) AS bug_count
 FROM BugsProducts p
 LEFT OUTER JOIN Bugs f ON (p.bug_id = f.bug_id AND f.status = 'FIXED')
 WHERE p.product_id = 1
 GROUP BY p.product_id, f.status)

UNION ALL

(SELECT p.product_id, o.status, COUNT(o.bug_id) AS bug_count
 FROM BugsProducts p
 LEFT OUTER JOIN Bugs o ON (p.bug_id = o.bug_id AND o.status = 'OPEN')
 WHERE p.product_id = 1
 GROUP BY p.product_id, o.status)

ORDER BY bug_count;
```

The result of the query is the result of each subquery, concatenated together. This example has two rows, one for each subquery. Remember

to include a column to distinguish the results of one subquery from the other, in this case the status column.

Use the UNION operation only when the columns in both subqueries are compatible. You can't change the number, name, or data type of columns midway through a result set, so be sure that the columns apply to all the rows consistently and sensibly. If you catch yourself defining a column alias like bugcount_or_customerid_or_null, you're probably using UNION to combine query results that are not compatible.

Solving Your Boss's Problem

How could you have solved the urgent request for statistics about your project? Your boss said, "I need to know how many products we work on, how many developers fixed bugs, the average bugs fixed per developer, and how many of our fixed bugs were reported by customers."

The best solution is to split up the work:

- How many products:

Download Spaghetti-Query/soln/count-products.sql

```
SELECT COUNT(*) AS how_many_products
FROM Products;
```

- How many developers fixed bugs:

Download Spaghetti-Query/soln/count-developers.sql

```
SELECT COUNT(DISTINCT assigned_to) AS how_many_developers
FROM Bugs
WHERE status = 'FIXED';
```

- Average number of bugs fixed per developer:

Download Spaghetti-Query/soln/bugs-per-developer.sql

```
SELECT AVG(bugs_per_developer) AS average_bugs_per_developer
FROM (SELECT dev.account_id, COUNT(*) AS bugs_per_developer
      FROM Bugs b JOIN Accounts dev
        ON (b.assigned_to = dev.account_id)
      WHERE b.status = 'FIXED'
      GROUP BY dev.account_id) t;
```

- How many of our fixed bugs were reported by customers:

Download Spaghetti-Query/soln/bugs-by-customers.sql

```
SELECT COUNT(*) AS how_many_customer_bugs
FROM Bugs b JOIN Accounts cust ON (b.reported_by = cust.account_id)
WHERE b.status = 'FIXED' AND cust.email NOT LIKE '%@example.com';
```

Some of these queries are tricky enough by themselves. Trying to combine them all into a single pass would be a nightmare.

Writing SQL Automatically—with SQL

When you split up a complex SQL query, the result may be many similar queries, perhaps varying slightly depending on data values. Writing these queries is a chore, so it's a good application of code generation.

Code generation is the technique of writing code whose output is new code you can compile or run. This can be worthwhile if the new code is laborious to write by hand. A code generator can eliminate repetitive work for you.

Multitable Updates

During a consulting job, I was called to solve an urgent SQL problem for a manager in another department.

I went to the manager's office and found a harried-looking fellow who was clearly at the end of his rope. We had barely exchanged greetings when he began sharing with me his woes. "I sure hope you can solve this problem quickly; our inventory system has been offline *all day.*" He was no amateur with SQL, but he told me he had been working for hours on a statement that could update a large set of rows.

His problem was that he couldn't use a consistent SQL expression in his UPDATE statement for all values of rows. In fact, the value he needed to set was different on each row. His database tracked inventory for a computer lab and the usage of each computer. He wanted to set a column called last_used to the most recent date each computer had been used.

He was too focused on solving this complex task in a single SQL statement, another example of the Spaghetti Query antipattern. In the hours he had been struggling to write the perfect UPDATE, he could have made the changes manually.

Instead of writing one SQL statement to solve his complex update, I wrote a script to generate a set of simpler SQL statements that had the desired effect:

Download Spaghetti-Query/soln/generate-update.sql

```sql
SELECT CONCAT('UPDATE Inventory '
  ' SET last_used = ''', MAX(u.usage_date), '''',
  ' WHERE inventory_id = ', u.inventory_id, ';') AS update_statement
FROM ComputerUsage u
GROUP BY u.inventory_id;
```

The output of this query is a series of UPDATE statements, complete with semicolons, ready to run as an SQL script:

update_statement
UPDATE Inventory SET last_used = '2002-04-19' WHERE inventory_id = 1234;
UPDATE Inventory SET last_used = '2002-03-12' WHERE inventory_id = 2345;
UPDATE Inventory SET last_used = '2002-04-30' WHERE inventory_id = 3456;
UPDATE Inventory SET last_used = '2002-04-04' WHERE inventory_id = 4567;
. . .

With this technique, I solved in minutes what that manager had been struggling with for hours.

Executing so many SQL queries or statements may not be the most efficient way to accomplish a task. But you should balance the goal of efficiency against the goal of getting the task done.

Although SQL makes it seem possible to solve a complex problem in a single line of code, don't be tempted to build a house of cards.

How can I tell what I think till I see what I say?
► E. M. Forster

Chapter 19

Implicit Columns

A PHP programmer asked for help troubleshooting the confusing result of a seemingly straightforward SQL query against his library database:

Download Implicit-Columns/intro/join-wildcard.sql

```
SELECT * FROM Books b JOIN Authors a ON (b.author_id = a.author_id);
```

This query returned all book titles as null. Even stranger, when he ran a different query without joining to the Authors, the result included the real book titles as expected.

I helped him find the cause of his trouble: the PHP database extension he was using returned each row resulting from the SQL query as an associative array. For example, he could access the Books.isbn column as $row["isbn"]. In his tables, both Books and Authors had a column called title (the latter was for titles like *Dr.* or *Rev.*). A single-result array element $row["title"] can store only one value; in this case, Authors.title occupied that array element. Most authors in the database had no title, so the result was that $row["title"] appeared to be null. When the query skipped the join to Authors, no conflict existed between column names, and the book title occupied the array element as expected.

I told the programmer that the solution was to declare a column alias to give one or the other title column a different name so that each would have a separate entry in the array.

Download Implicit-Columns/intro/join-alias.sql

```
SELECT b.title, a.title AS salutation
FROM Books b JOIN Authors a ON (b.author_id = a.author_id);
```

His second question was, "How do I give one column an alias but also request other columns?" He wanted to continue using the wildcard (SELECT *) but apply an alias to one column covered by the wildcard.

19.1 Objective: Reduce Typing

Software developers don't seem to like to type, which in a way makes their choice of career ironic, like the twist ending in an O. Henry story.

One example that programmers cite as requiring too much typing is when writing all the columns used in an SQL query:

Download Implicit-Columns/obj/select-explicit.sql

```sql
SELECT bug_id, date_reported, summary, description, resolution,
  reported_by, assigned_to, verified_by, status, priority, hours
FROM Bugs;
```

It's no surprise that software developers gratefully use the SQL *wildcard* feature. The * symbol means *every column*, so the list of columns is implicit rather than explicit. This helps make queries more concise.

Download Implicit-Columns/obj/select-implicit.sql

```sql
SELECT * FROM Bugs;
```

Likewise, when using INSERT, it seems smart to take advantage of the default: the values apply to all the columns in the order they're defined in the table.

Download Implicit-Columns/obj/insert-explicit.sql

```sql
INSERT INTO Accounts (account_name, first_name, last_name, email,
  password, portrait_image, hourly_rate) VALUES
  ('bkarwin', 'Bill', 'Karwin', 'bill@example.com', SHA2('xyzzy'), NULL, 49.95);
```

It's shorter to write the statement without listing the columns.

Download Implicit-Columns/obj/insert-implicit.sql

```sql
INSERT INTO Accounts VALUES (DEFAULT,
  'bkarwin', 'Bill', 'Karwin', 'bill@example.com', SHA2('xyzzy'), NULL, 49.95);
```

19.2 Antipattern: a Shortcut That Gets You Lost

Although using wildcards and unnamed columns satisfies the goal of less typing, this habit creates several hazards.

Breaking Refactoring

Suppose you need to add a column to the Bugs table, such as date_due for scheduling purposes.

Download **Implicit-Columns/anti/add-column.sql**

```sql
ALTER TABLE Bugs ADD COLUMN date_due DATE;
```

Your INSERT statement now results in an error, because you listed eleven values instead of the twelve the table now expects.

Download **Implicit-Columns/anti/insert-mismatched.sql**

```sql
INSERT INTO Bugs VALUES (DEFAULT, CURDATE(), 'New bug', 'Test T987 fails...',
    NULL, 123, NULL, NULL, DEFAULT, 'Medium', NULL);

-- SQLSTATE 21S01: Column count doesn't match value count at row 1
```

In an INSERT statement that uses implicit columns, you must give values for all columns in the same order that columns are defined in the table. If the columns change, the statement produces an error—or even assigns values to the wrong columns.

Suppose you run a SELECT * query, and since you don't know the column names, you reference columns based on their ordinal position:

Download **Implicit-Columns/anti/ordinal.php**

```php
<?php
$stmt = $pdo->query("SELECT * FROM Bugs WHERE bug_id = 1234");
$row = $stmt->fetch();
$hours = $row[10];
```

But unknown to you, another person on the team dropped a column:

Download **Implicit-Columns/anti/drop-column.sql**

```sql
ALTER TABLE Bugs DROP COLUMN verified_by;
```

The hours column is no longer at position 10. Your application is using the value in another column by mistake. As columns are renamed, added, or dropped, your query result could change in ways your code doesn't support. You can't predict how many columns your query returns if you use a wildcard.

These errors can propagate through your code, and by the time you notice the problem in the output of the application, it's hard to trace back to the line where the mistake occurred.

Hidden Costs

The convenience of using wildcards in queries can harm performance and scalability. The more columns your query fetches, the more data must travel over the network between your application and the database server.

You probably have many queries running concurrently in your production application environment. They compete for access to the same network bandwidth. Even a gigabit network can be saturated by a hundred application clients querying for thousands of rows at a time.

Object-relational mapping (ORM) techniques such as Active Record often use SELECT * by default to populate the fields of an object representing a row in a database. Even if the ORM offers the means to override this behavior, most programmers don't bother.

You Asked for It, You Got It

One of the most common questions I see from programmers using the SQL wildcard is, "Is there a shortcut to request all columns, except a few that I specify?" Perhaps these programmers are trying to avoid the resource cost of fetching bulky TEXT columns that they don't need, but they do want the convenience of using a wildcard.

The answer is no, SQL does not support any syntax, which means, "all the columns I want but none that I don't want." Either you use the wildcard to request all columns from a table, or else you have to list the columns you want explicitly.

19.3   How to Recognize the Antipattern

The following scenarios may indicate that your project is using implicit columns inappropriately, and it's causing trouble:

- "The application broke because it's still referencing columns in the database result set by the old column names. We tried to update all the code, but I guess we missed some."

  You've changed a table in the database—adding, deleting, renaming, or changing the order of columns—but you failed to change your application code that references the table. It's laborious to track down all these references.

- "It took us days to track down our network bottleneck, and we finally narrowed it down to excessive traffic to the database server. According to our statistics, the average query fetches more than 2MB of data but displays less than a tenth of that."

You're fetching a lot of data you don't need.

19.4 Legitimate Uses of the Antipattern

A well-justified use of wildcards is in ad hoc SQL when you're writing quick queries to test a solution or as a diagnostic check of current data. A single-use query benefits less from maintainability.

The examples in this book use wildcards to save space and to avoid distracting from the more interesting parts of the example queries. I rarely use SQL wildcards in production application code.

If your application needs to run a query that adapts when columns are added, dropped, renamed, or repositioned, you may find it best to use wildcards. Be sure to plan for the extra work it takes to troubleshoot the pitfalls described earlier.

You can use wildcards for each table individually in a join query. Prefix the wildcard with the table name or alias. This allows you to specify a short list of specific columns you need from one table, while using the wildcard to fetch all columns from the other table. For example:

Download Implicit-Columns/legit/wildcard-one-table.sql

```sql
SELECT b.*, a.first_name, a.email
FROM Bugs b JOIN Accounts a
  ON (b.reported_by = a.account_id);
```

Keying in a long list of column names can be time-consuming. For some people, development efficiency is more important than runtime efficiency. Likewise, you might place a priority on writing queries that are shorter and therefore more readable. Using wildcards does reduce keystrokes and result in a shorter query, so if this is your priority, then use wildcards.

I've heard a developer claim that a long SQL query passing from the application to the database server causes too much network overhead. In theory, query length could make a difference in some cases. But it's more common that the rows of data that your query returns use much more network bandwidth than your SQL query string. Use your judgment about exception cases, but don't sweat the small stuff.

19.5 Solution: Name Columns Explicitly

Always spell out all the columns you need, instead of relying on wild-cards or implicit column lists.

Download **Implicit-Columns/soln/select-explicit.sql**

```sql
SELECT bug_id, date_reported, summary, description, resolution,
  reported_by, assigned_to, verified_by, status, priority, hours
FROM Bugs;
```

Download **Implicit-Columns/soln/insert-explicit.sql**

```sql
INSERT INTO Accounts (account_name, first_name, last_name, email,
  password_hash, portrait_image, hourly_rate)
VALUES ('bkarwin', 'Bill', 'Karwin', 'bill@example.com',
  SHA2('xyzzy'), NULL, 49.95);
```

All this typing seems burdensome, but it's worth it in several ways.

Mistake-Proofing

Remember poka-yoke?[1] You make your SQL queries more resistant to the errors and confusion described earlier when you specify the columns in the select-list of the query.

- If a column has been repositioned in the table, it doesn't change position in a query result.

- If a column has been added in the table, it doesn't appear in the query result.

- If a column has been dropped from the table, your query raises an error—but it's a good error, because you're led directly to the code that you need to fix, instead of left to hunt for the root cause.

You get similar benefits when you specify columns in INSERT statements. The order of columns you specify overrides the order in the table definition, and values are assigned to the columns you intend. Newly added columns you haven't named in your statement are given default values or null. If you reference a column that has been deleted, you get an error, but troubleshooting is easier.

This is an example of the *fail early* principle.

---

1. The practice from the Japanese industry of designing mistake-proof systems. See Chapter 5, *Keyless Entry*, on page 65.

You Ain't Gonna Need It

If you're concerned about the scalability and throughput of your software, you should look for possible wasteful use of network bandwidth. The bandwidth of an SQL query can seem harmless during software development and testing, but it bites you when your production environment is running thousands of SQL queries per second.

Once you abandon the SQL wildcard, you're naturally motivated to leave out unneeded columns—it means less typing. This promotes more efficient use of bandwidth too.

Download **Implicit-Columns/soln/yagni.sql**

```sql
SELECT date_reported, summary, description, resolution, status, priority
FROM Bugs;
```

You Need to Give Up Wildcards Anyway

When you buy a bag of M&M's candies from the vending machine, the wrapper is a convenience, making it easy to carry the package of candies back to your desk. Once you open the bag, however, you need to treat M&M's as individuals. They roll, slide, and bounce all over the place. If you're not careful, some may fall under your desk and attract bugs. But there's no way to eat one until you tear open the bag.

In an SQL query, as soon as you want to apply an expression to a column or use a column alias or exclude columns for the sake of efficiency, you need to break open the "container" provided by the wildcard. You lose the convenience of treating the collection of columns as a single package, but you gain access to all of its contents.

You'll inevitably need to treat some columns in a query individually by employing a column alias or a function or removing a column from the list. If you skip the use of wildcards from the beginning, it'll be easier to change your query later.

Take all you want, but eat all you take.

Part IV

Application Development Antipatterns

The enemy knows the system.
► Shannon's maxim

Chapter 20

Readable Passwords

Suppose you receive a phone call from a man using one of the applications you support. The caller is having trouble logging in.

"This is Pat Johnson in Sales. I must have forgotten my password. Can you just look it up and tell me what it is?" Pat sounds a bit sheepish but also strangely in a hurry.

"I'm sorry, I'm not supposed to do that," you answer. "I can reset your account, and that'll send an email to the address you registered for your account. You can use the instructions in that email to set a new password."

The man on the phone becomes more impatient and assertive. "That's ridiculous," he says. "At my last company the support staff could look up my password. Are you unable to do your job? Do you want me to call your manager?"

Naturally, you want to preserve a smooth relationship with your users, so you run an SQL query to look up the plain-text password for Pat Johnson's account and read it to him over the phone.

The man hangs up. You comment to your co-worker, "That was a close call. I almost had an escalation from Pat Johnson. I hope he doesn't complain."

Your co-worker looks puzzled. "*He?* Pat Johnson in Sales is a woman. I think you just gave her password to a con artist."

20.1   Objective: Recover or Reset Passwords

It's a sure bet that in any application that has passwords, a user will forget his password. Most modern applications handle this by giving the user a chance to recover or reset his password through an email

feedback mechanism. This solution depends on the user having access to the email address associated with the user profile in the application.

20.2 Antipattern: Store Password in Plain Text

The frequent mistake in these kinds of password-recovery solutions is that the application allows the user to request an email containing his password in clear text. This is a dire security flaw related to the database design, and it leads to several security risks that could allow unauthorized people to gain privileged access to the application.

Let's explore these risks in the following sections, assuming our example bug-tracking database has a table Accounts, where each user's account is stored as a row in this table.

Storing Passwords

A password is typically stored in the Accounts table as a string attribute column:

Download **Passwords/anti/create-table.sql**

```sql
CREATE TABLE Accounts (
  account_id    SERIAL PRIMARY KEY,
  account_name  VARCHAR(20) NOT NULL,
  email         VARCHAR(100) NOT NULL,
  password      VARCHAR(30) NOT NULL
);
```

You can create an account simply by inserting one row and specifying the password as a string literal:

Download **Passwords/anti/insert-plaintext.sql**

```sql
INSERT INTO Accounts (account_id, account_name, email, password)
  VALUES (123, 'billkarwin', 'bill@example.com', 'xyzzy');
```

It's not secure to store a password in clear text or even to pass it over the network in the clear. If an attacker can read the SQL statement you use to insert a password, they can see the password plainly. This is also true for SQL statements to change a password or verify that user input matches a stored password. Hackers have several opportunities to steal a password, including the following:

- Intercepting network packets as the SQL statement is sent from the application client to the database server. This is easier than it sounds; there are free software tools such as Wireshark.[1]

---

1. Wireshark (formerly known as Ethereal) is available at http://www.wireshark.org/.

- Searching SQL query logs on the database server. The attacker needs access to the database server host, but assuming they have that, they can access log files that may include a record of SQL statements executed by that database server.

- Reading data from database backup files on the server or on backup media. Are your backup media kept safe? Do you erase backup media destructively before they are recycled or disposed of?

Authenticating Passwords

Later, when the user tries to log in, your application compares the user's input to the password string stored in the database. This comparison is done as plain text, since the password itself is stored in plain text. For example, you can use a query like the following to return a 0 (false) or 1 (true), indicating whether the user's input matches the password in the database:

Download **Passwords/anti/auth-plaintext.sql**

```sql
SELECT CASE WHEN password = 'opensesame' THEN 1 ELSE 0 END
  AS password_matches
FROM Accounts
WHERE account_id = 123;
```

In the previous example, the password the user entered, *opensesame*, is incorrect, and the query returns a zero value.

Like in the earlier section on storing passwords, interpolating the user's input string into the SQL query in plain text exposes it to discovery by an attacker.

Don't Lump Together Two Different Conditions

Most of the time, I see the authentication query place conditions for both the account_id and password columns in the WHERE clause:

Download **Passwords/anti/auth-lumping.sql**

```sql
SELECT * FROM Accounts
WHERE account_name = 'bill' AND password = 'opensesame';
```

This query returns an empty result set if the account doesn't exist or if the user gave the wrong password. Your application can't separate the two causes for failed authentication. It's better to use a query that can treat the two cases as distinct. Then you can handle the failure appropriately.

For example, you may want to lock an account temporarily if you detect many failed logins, because this may indicate an attempted intrusion. However, you can't detect this pattern if you can't tell the difference between a wrong account name and a wrong password.

Sending Passwords in Email

Since the password is stored in plain text in the database, retrieving the password in your application is simple:

Download Passwords/anti/select-plaintext.sql

```sql
SELECT account_name, email, password
FROM Accounts
WHERE account_id = 123;
```

Your application can then send to a user's email address on request. You've probably seen one of these emails as part of the password reminder feature of any number of websites you use. An example of this kind of email is shown here:

Example of Password Recovery Email:

```
From: daemon
To: bill@example.com
Subject: password request

You requested a reminder of the password for your account "bill".
Your password is "xyzzy".

Click the link below to log in to your account:

http://www.example.com/login
```

Sending an email with the password in plain text is a serious security risk. Email can be intercepted, logged, and stored in multiple ways by hackers. It's not good enough that you use a secure protocol to view mail or that the sending and receiving mail servers are managed by responsible system administrators. Since email is routed across the Internet, it can be intercepted at other sites. Secure protocols for email aren't necessarily widespread or under your control.

20.3 How to Recognize the Antipattern

Any application that can recover your password and send it to you must be storing it in plain text or at least with some reversible encoding. This is the antipattern. If your application can read a password for a legitimate purpose, then it's possible that a hacker can read the password illicitly.

20.4 Legitimate Uses of the Antipattern

Your application may need to use a password to access another third-party service—that is, your application can be a client. In this case,

> ### Ethics of Software Development
>
> If you're developing an application that supports passwords and you're asked to design a password recovery feature, you should push back respectfully, warn the project decision makers about the risks, and offer an alternative solution that provides similar value.
>
> Just as an electrician should recognize and correct a wiring design that poses an unsafe fire risk, it's your responsibility as a software engineer to be aware of safety issues and to promote safer software.
>
> A good book you should read is *19 Deadly Sins of Software Security* (HLV05). Another good resource is the Open Web Application Security Project (http://owasp.org).

you must store that password in a readable format. Preferably, you use some encoding that your application can reverse, instead of using plain text in the database.

You can make a distinction between *identification* and *authentication*. A user can identify himself as anyone he wants, but authentication is proving he is who he says he is. Passwords are the most common way of doing this.

If you can't enforce security strong enough to defeat skilled and determined attackers, then you effectively have an identification mechanism but not a reliable authentication mechanism. But this isn't necessarily a deal-breaker.

Not every software application is at risk for attack, and not every application contains sensitive information that must be protected. For example, an intranet application may be accessed by only a few people who are known to be honest and cooperative. In this case, an identification mechanism may be enough for the application to work, and in those informal environments, a simpler login design may be adequate. The additional work necessary to create a strong authentication system may not be justified.

Be careful, though—applications have a tendency to evolve beyond their original environment or role. Before you make your quaint little intranet application available outside your company firewall, you should get a qualified security expert to evaluate it.

20.5 Solution: Store a Salted Hash of the Password

The chief problem in this antipattern is that the original form of the password is readable. But you can authenticate the user's input against a password without reading it. This section describes how to implement this kind of secure password storage in an SQL database.

Understanding Hash Functions

Encode the password using a one-way *cryptographic hash function*. This function transforms its input string into a new string, called the *hash*, that is unrecognizable. Even the length of the original string is obscured, because the hash returned by a hash function is a fixed-length string. For example, the SHA-256 algorithm converts our example password, *xyzzy*, to a 256-bit string of bits, usually represented as a 64-character string of hexadecimal digits:

```
SHA2('xyzzy') = '184858a00fd7971f810848266ebcecee5e8b69972c5ffaed622f5ee078671aed'
```

Another characteristic of a hash is that it's not reversible. You can't recover the input string from its hash because the hashing algorithm is designed to "lose" some information about the input. A good hashing algorithm should take as much work to crack as it would to simply guess the input through trial and error.

A popular algorithm in the past has been SHA-1, but researchers have recently proved this 160-bit hashing algorithm to have insufficient cryptographic strength; a technique exists to infer the input from a hash string. This technique is very time-consuming but nevertheless takes less time than it would take to guess the password by trial and error. The National Institute of Standards and Technology (NIST) has announced a plan to phase out SHA-1 as an approved secure hashing algorithm after 2010 in favor of these stronger variants: SHA-224, SHA-256, SHA-384, and SHA-512.[2] Whether you need to comply with NIST standards or not, it's a good idea to use at least SHA-256 for passwords.

MD5() is another popular hash function, producing hash strings of 128 bits. MD5() has also been shown to be cryptographically weak, so you shouldn't use it for encoding passwords. Weaker algorithms still have uses but not for sensitive information like passwords.

---

2. http://csrc.nist.gov/groups/ST/toolkit/secure_hashing.html

Using a Hash in SQL

The following is a redefinition of the Accounts table. The SHA-256 pass-word hash is always 64 characters long, so define the column as a fixed-length CHAR column of that length.

Download **Passwords/soln/create-table.sql**

```sql
CREATE TABLE Accounts (
  account_id     SERIAL PRIMARY KEY,
  account_name   VARCHAR(20),
  email          VARCHAR(100) NOT NULL,
  password_hash  CHAR(64) NOT NULL
);
```

Hashing functions aren't part of the standard SQL language, so you may need to rely on your database brand to support hashing as an extension. For example, MySQL 6.0.5 with SSL support includes a function SHA2(), which returns a 256-bit hash by default.

Download **Passwords/soln/insert-hash.sql**

```sql
INSERT INTO Accounts (account_id, account_name, email, password_hash)
  VALUES (123, 'billkarwin', 'bill@example.com', SHA2('xyzzy'));
```

You can validate a user's input by applying the same hash function to it and comparing the result to the value stored in the database.

Download **Passwords/soln/auth-hash.sql**

```sql
SELECT CASE WHEN password_hash = SHA2('xyzzy') THEN 1 ELSE 0 END
  AS password_matches
FROM Accounts
WHERE account_id = 123;
```

You can lock an account easily by changing the value in the password hash to a string that the hash function can't return. For example, the string *noaccess* contains letters that aren't hexadecimal digits.

Adding Salt to Your Hash

If you store hashes instead of passwords and the attacker gains access to your database (by searching your trash for a CDROM backup, for example), he can still attempt to guess passwords by trial and error. Guessing each password may take a long time, but he can prepare his own database of hashes of likely passwords against which to compare the hash strings he finds in your database. If only one user chose a password that is a word in the dictionary, it's easy for an attacker to find it by searching your password database for hashes that match his prepared table of hashes. He can even do this with SQL:

```
CREATE TABLE DictionaryHashes (
  password        VARCHAR(100),
  password_hash   CHAR(64)
);

SELECT a.account_name, h.password
FROM Accounts AS a JOIN DictionaryHashes AS h
  ON a.password_hash = h.password_hash;
```

One way to defeat this kind of "dictionary attack" is by including a *salt* in your password-encoding expression. A salt is a string of meaningless bytes you concatenate with the user's password, before passing the resulting string to the hash function. Even if the user chose a word in the dictionary as their password, the hash produced from a salted password won't match the hash in the attacker's hash database. For example, if the password is the word *password*, you can see that the hash of this word is different from a hash of the word with a few random bytes appended:

```
SHA2('password')
  = '5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8'

SHA2('password-0xT!sp9')
  = '7256d8d7741f740ee83ba7a9b30e7ac11fcd9dbd7a0147f4cc83c62dd6e0c45b'
```

Each password should use a different salt value to make an attacker have to generate a new dictionary table of hashes for each password. Then he's back to square one, because cracking passwords in your database takes as much time as guessing them with trial and error.[3]

```
CREATE TABLE Accounts (
  account_id      SERIAL PRIMARY KEY,
  account_name    VARCHAR(20),
  email           VARCHAR(100) NOT NULL,
  password_hash   CHAR(32) NOT NULL,
  salt            BINARY(8) NOT NULL
);

INSERT INTO Accounts (account_id, account_name, email,
    password_hash, salt)
  VALUES (123, 'billkarwin', 'bill@example.com',
    SHA2('xyzzy' || '-0xT!sp9'), '-0xT!sp9');
```

---

3. A related, and more sophisticated, technique to recover passwords from their hashes is called a *rainbow table*. Employing a salt defends against this technique too.

```
SELECT (password_hash = SHA2('xyzzy' || salt)) AS password_matches
FROM Accounts
WHERE account_id = 123;
```

A good length for a salt is 8 bytes. You should generate the salt randomly for each password. The previous examples show a salt string containing printable characters, but remember you can make a salt using any random, unprintable bytes.

Hiding the Password from SQL

Now that you're using a strong hashing function to encode the password before you store it and you use a salt to thwart dictionary attacks, you would think this is enough to ensure security. But the password still appears in plain text in the SQL expression, which means that it's readable if an attacker can intercept network packets or if SQL queries are logged and the log files fall into the wrong hands.

You can protect against this kind of exposure if you don't put the plain-text password into the SQL query. Instead, compute the hash in your application code, and use only the hash in the SQL query. It does an attacker little good to intercept the hash, because he can't reverse it to get the password.

You do need the salt before you can compute the hash.

The following is a PHP example using the PDO extension to get the salt, compute a hash, and run a query to validate the password against the salted hash stored in the database:

Download Passwords/soln/auth-salt.php

```php
<?php

$password = 'xyzzy';

$stmt = $pdo->query(
    "SELECT salt
      FROM Accounts
      WHERE account_name = 'bill'");

$row = $stmt->fetch();
$salt = $row[0];

$hash = hash('sha256', $password . $salt);

$stmt = $pdo->query("
  SELECT (password_hash = '$hash') AS password_matches;
  FROM Accounts AS a
  WHERE a.acct_name = 'bill'");
```

```
$row = $stmt->fetch();
if ($row === false) {
  // account 'bill' does not exist
} else {
  $password_matches = $row[0];
  if (!$password_matches) {
    // password given was incorrect
  }
}
```

The hash() function is guaranteed to return only hexadecimal digits, so there's no risk of SQL injection (see Chapter 21, *SQL Injection*, on page 234).

In web applications, there's another place where attackers have an opportunity to intercept data on the network: between the user's browser and the web application server. When the user submits a login form, the browser sends his password in plain text to the server, where it's used to compute a hash as described earlier. You could protect against this by encoding the password into a hash in the user's browser before sending the form data. But this is awkward because you need to retrieve the salt associated with that password before you can compute the correct hash. A good compromise is to use a secure HTTP connection whenever sending a password from browser to the application.

Resetting the Password Instead of Recovering the Password

Now that the password is stored in a more secure way, you still need to solve the original objective: help users who have forgotten their password. You can't recover their password, because now your database stores a hash instead of the password. You can't reverse the hash any more easily than an attacker could. But you can allow a user access in other ways. Two sample implementations are described here.

The first alternative is that when a user who has forgotten his password requests help, instead of emailing his password to him, your application can send an email with a temporary password generated by the application. For additional security, the application may expire the temporary password after a short time, so if the email is intercepted, it's more likely that it will not allow unauthorized access. Also, the application should be designed so that the user is forced to change the password as his first action when he logs in.

Example of Email with a System-Generated Temporary Password

```
From: daemon
To: bill@example.com
Subject: password reset

You requested to reset your password for your account.

Your temporary password is "p0trz3b1e".
This password will cease to allow access after one hour.

Click the link below to log in to your account and
set your new password:

http://www.example.com/login
```

In a second alternative, instead of including a new password in an email, the request is logged in a database table and assigned a unique token as an identifier:

Download Passwords/soln/reset-request.sql

```
CREATE TABLE PasswordResetRequest (
  token        CHAR(32) PRIMARY KEY,
  account_id   BIGINT UNSIGNED NOT NULL,
  expiration   TIMESTAMP NOT NULL,
  FOREIGN KEY (account_id) REFERENCES Accounts(account_id)
);

SET @token = MD5('billkarwin' || CURRENT_TIMESTAMP);

INSERT INTO PasswordResetRequest (token, account_id, expiration)
  VALUES (@token, 123, CURRENT_TIMESTAMP + INTERVAL 1 HOUR);
```

Then you include the token in an email. You could also send the token in some other message, such as SMS, as long as it's an address that's already associated with the account requesting a password reset. That way, if a stranger requests a password reset illicitly, it sends a spurious email only to the actual owner of the account.

Example of Email with a Temporary Link to a Password Reset Page

```
From: daemon
To: bill@example.com
Subject: password reset

You requested to reset your password for your account.

Click the link below within one hour to change your password.
After one hour, the link below will no longer work and your
password will remain unchanged.

http://www.example.com/reset_password?token=f5cabff22532bd0025118905bdea50da
```

When the application receives a request for the special reset_password screen, the value in the token parameter must match a row in the PasswordResetRequest table, and the expiration timestamp on this row must still be upcoming, not past. The account_id on this row references the Accounts table, so the token is restricted to enable a password reset of only one specific account.

Of course, it would be harmful if the wrong people could access this page. Simple restrictions reduce this risk, such as giving the special screen a short expiration period and making sure the screen does not show the account for which the password is being set.

The state of cryptography is constantly advancing, trying to stay ahead of attack technology. The techniques in this chapter will improve a great number of typical applications, but if you need to develop very secure systems, you should move on to more advanced techniques such as the following:

- PBKDF2 (http://tools.ietf.org/html/rfc2898) is a widely used *key strengthening* standard.

- Bcrypt (http://bcrypt.sourceforge.net/) implements an *adaptive hashing* function.

If you can read passwords, so can a hacker.

Quote me as saying I was misquoted.
  ► Groucho Marx

Chapter 21

SQL Injection

In March 2010, serial computer hacker Albert Gonzalez was convicted for his role in the largest identity theft in history. He acquired an estimated 130 million credit and debit card numbers by hacking into ATM machines and payment systems of several major retail store chains and the credit-card processing companies that serve them.

Gonzales broke the previous record, which he also held, for stealing 45.6 million credit and debit card numbers in 2006. He performed that earlier crime by exploiting vulnerable wireless networks.

How did Gonzalez nearly triple his own record? We imagine a daring plot from a James Bond movie, with black-clad agents rappelling down elevators shafts, using supercomputers to crack state-of-the-art encrypted passwords, or sabotaging electrical power to an entire city.

The indictment describes a more mundane reality. Gonzalez exploited a vulnerability that is one of the most common security weaknesses on the Internet. He was able to use an attack technique called *SQL Injection* to gain privileged access to upload files to the corporate victims' servers. After Gonzalez and his coconspirators gained this access, the indictment states:[1]

Executing the Attacks: The Malware

…they would install "sniffer" programs that would capture credit and debit card numbers, corresponding Card Data, and other information on a real-time basis as the information moved through the Corporate Victims' credit and debit card processing networks, and then periodically transmit that information to the coconspirators.

---

1. http://voices.washingtonpost.com/securityfix/heartlandIndictment.pdf

The retailers whose websites Gonzalez attacked have said that they've made changes to correct these security holes. However, they've plugged only one hole, while new web applications are created every day that contain other holes. SQL Injection attacks remain an easy target for hackers, because software developers don't understand the nature of the vulnerability or how to write code to prevent it.

21.1 Objective: Write Dynamic SQL Queries

SQL is intended to be used in concert with application code. When you build SQL queries as strings and combine application variables into the string, this is commonly called *dynamic SQL*.[2]

Download SQL-Injection/obj/dynamic-sql.php

```php
<?php
$sql = "SELECT * FROM Bugs WHERE bug_id = $bug_id";
$stmt = $pdo->query($sql);
```

This simple example shows interpolating a PHP variable into a string. We intend that $bug_id is an integer so that by the time the database receives the query, the value of $bug_id is part of the query.

Dynamic SQL queries are a natural way to get the most out of a database. When you use application data to specify how you want to query a database, you're using SQL as a two-way language. Your application is having a kind of dialogue with the database.

However, it's not too hard to make your software do tasks that you want it to do—the harder challenge is making your software secure so it doesn't allow actions that you don't want it to do. Software defects resulting from SQL Injection are failures to satisfy the latter.

21.2 Antipattern: Execute Unverified Input As Code

SQL injection happens when you interpolate some content into an SQL query string and the content modifies the syntax of your query in ways you didn't intend. In the classic example of SQL Injection, the value you interpolate into your string finishes the SQL statement and executes a second complete statement. For instance, if the value of the $bug_id

---

2. Technically, any query parsed at runtime is dynamic SQL, but in common usage, it describes SQL that includes variable data.

Figure 21.1: Exploits of a mom

variable is *1234; DELETE FROM Bugs*, the resulting SQL shown earlier would look like this:

Download SQL-Injection/anti/delete.sql

```
SELECT * FROM Bugs WHERE bug_id = 1234; DELETE FROM Bugs
```

This type of SQL Injection can be spectacular, as shown in Figure 21.1.[3] Usually these flaws are more subtle—but still dangerous.

Accidents May Happen

Suppose you are writing a web interface to view the bugs database and one page allows you to view a project based on its name:

Download SQL-Injection/anti/ohare.php

```php
<?php
$project_name = $_REQUEST["name"];
$sql = "SELECT * FROM Projects WHERE project_name = '$project_name'";
```

The trouble begins when your team is hired to develop software for O'Hare International Airport in Chicago. You naturally give the project a name like "O'Hare." How do you submit a request to view the project in your web application?

http://bugs.example.com/project/view.php?name=O'Hare

---

3.   Cartoon by Randall Munroe, used with permission (http://xkcd.com/327/).

Your PHP code takes the value of that request parameter and interpolates it into the SQL query, but it produces a query that neither you nor the user intended:

Download SQL-Injection/anti/ohare.sql

```sql
SELECT * FROM Projects WHERE project_name = 'O'Hare'
```

Because a string is terminated by the first quote character it finds, the resulting expression contains a short string, 'O', followed by some extra characters, Hare', that make no sense in this context. The database can only report this as a syntax error. This is an honest accident. The risk of anything bad happening is low, because a statement with a syntax error can't execute. The greater risk is that the statement executes without error but does something you didn't intend.

The Top Web Security Threat

SQL Injection becomes a greater threat when an attacker can use this to manipulate your SQL statements. For example, your application may allow a user to change his or her password:

Download SQL-Injection/anti/set-password.php

```php
<?php
$password = $_REQUEST["password"];
$userid = $_REQUEST["userid"];
$sql = "UPDATE Accounts SET password_hash = SHA2('$password')
    WHERE account_id = $userid";
```

A clever attacker who can guess how the request parameters are used in your SQL statement can send a carefully chosen string to exploit it:

http://bugs.example.com/setpass?password=xyzzy&userid=123 OR TRUE

After interpolating the string from the userid parameter into your SQL expression, the string has changed the syntax of the statement. Now it changes the password for *every* account in the database, not for one specific account:

Download SQL-Injection/anti/set-password.sql

```sql
UPDATE Accounts SET password_hash = SHA2('xyzzy')
WHERE account_id = 123 OR TRUE;
```

This is key to understanding SQL Injection and also how to combat it: SQL Injection works by changing the syntax of the SQL statement before the statement is parsed. As long as you insert dynamic portions to the statement before it's parsed, you have a risk of SQL Injection.

There are countless ways a maliciously chosen string can alter the behavior of your SQL statements. It's limited only by the imagination of the attacker and your ability to protect your SQL statements.

The Quest for a Cure

Now that we know the threat of SQL Injection, the next natural question is, what do we need to do to protect code from being exploited? You may have read a blog or an article that described some single technique and claimed it's the universal remedy against SQL Injection. In reality, none of these techniques is proof against all forms of SQL Injection, so you need to use all of them in different cases.

Escaping Values

The oldest way to protect SQL queries from accidental unmatched quote characters is to *escape* any quote characters to prevent them from becoming the end of the quoted string. In standard SQL, you can use two quote characters to make one literal quote character:

Download SQL-Injection/anti/ohare-escape.sql

```
SELECT * FROM Projects WHERE project_name = 'O''Hare'
```

Most brands of database also support the backslash to escape the following quote character, just like most other programming languages do:

Download SQL-Injection/anti/ohare-escape.sql

```
SELECT * FROM Projects WHERE project_name = 'O\'Hare'
```

The idea is that you transform application data before you interpolate it into SQL strings. Most SQL programming interfaces provide a convenience function. For example, in PHP's PDO extension, use the quote() function to both delimit a string with quote characters and escape any literal quote characters within the string.

Download SQL-Injection/anti/ohare-escape.php

```php
<?php
$project_name = $pdo->quote($_REQUEST["name"]);
$sql = "SELECT * FROM Projects WHERE project_name = $project_name";
```

This technique can reduce the risk of SQL Injection resulting from unmatched quote characters within the dynamic content. But it doesn't work as well for nonstring content.

Download **SQL-Injection/anti/set-password-escape.php**

```php
<?php
$password = $pdo->quote($_REQUEST["password"]);
$userid = $pdo->quote($_REQUEST["userid"]);
$sql = "UPDATE Accounts SET password_hash = SHA2($password)
    WHERE account_id = $userid";
```

Download **SQL-Injection/anti/set-password-escape.sql**

```sql
UPDATE Accounts SET password_hash = SHA2('xyzzy')
WHERE account_id = '123 OR TRUE'
```

You can't compare a numeric column directly to a string containing digits in all brands of database. Some databases may implicitly cast the string to a sensible numeric equivalent, but in standard SQL you have to use the CAST() function deliberately to convert a string to a numeric data type.

There are also obscure corner cases where strings in non-ASCII character sets can pass through a function intended to escape the quote characters but leave unescaped quote characters intact.[4]

Query Parameters

The solution most frequently cited as a panacea to SQL Injection is to use *query parameters*. Instead of interpolating dynamic values into your SQL string, leave *parameter placeholders* in the string as you prepare the query. Then provide a parameter value as you execute the prepared query.

Download **SQL-Injection/anti/parameter.php**

```php
<?php
$stmt = $pdo->prepare("SELECT * FROM Projects WHERE project_name = ?");
$params = array($_REQUEST["name"]);
$stmt->execute($params);
```

Many programmers recommend this solution because you don't have to escape dynamic content or worry about flawed escaping functions. In fact, query parameters are a very strong defense against SQL Injection. But parameters aren't a universal solution because the value of a query parameter is always interpreted as a single literal value.

---

4. See http://bugs.mysql.com/8378 for an example.

- No lists of values can be a single parameter:

Download **SQL-Injection/anti/parameter.php**

```php
<?php
$stmt = $pdo->prepare("SELECT * FROM Bugs WHERE bug_id IN ( ? )");
$stmt->execute(array("1234,3456,5678"));
```

This works as though you provided a single string value composed of digits and commas, which doesn't work the same as a series of integers:

Download **SQL-Injection/anti/parameter.sql**

```sql
SELECT * FROM Bugs WHERE bug_id IN ( '1234,3456,5678' )
```

- No table identifier can be a parameter:

Download **SQL-Injection/anti/parameter.php**

```php
<?php
$stmt = $pdo->prepare("SELECT * FROM ? WHERE bug_id = 1234");
$stmt->execute(array("Bugs"));
```

This works as though you had entered a string literal in place of the table name, which is simply a syntax error:

Download **SQL-Injection/anti/parameter.sql**

```sql
SELECT * FROM 'Bugs' WHERE bug_id = 1234
```

- No column identifier can be a parameter:

Download **SQL-Injection/anti/parameter.php**

```php
<?php
$stmt = $pdo->prepare("SELECT * FROM Bugs ORDER BY ?");
$stmt->execute(array("date_reported"));
```

In this example, the sort is a no-op, because the expression is a constant string, the same on every row:

Download **SQL-Injection/anti/parameter.sql**

```sql
SELECT * FROM Bugs ORDER BY 'date_reported';
```

- No SQL keyword can be a parameter:

Download **SQL-Injection/anti/parameter.php**

```php
<?php
$stmt = $pdo->prepare("SELECT * FROM Bugs ORDER BY date_reported ?");
$stmt->execute(array("DESC"));
```

> ### What Was My Complete Query?
>
> Many people think that using SQL query parameters is a way to quote values into an SQL statement automatically. This isn't accurate, and thinking about query parameters this way leads to misunderstanding about how they work.
>
> The RDBMS server parses your SQL as you *prepare* the query. After this, nothing can change the syntax of that SQL query.
>
> You provide values as you *execute* a prepared query. Each value you provide is used for each placeholder, one for one.
>
> You can execute a prepared query again, substituting new parameter values for the old values. So, the RDBMS must keep track of the query and the parameter values separately. This is good for security.
>
> This means that if you retrieve the prepared SQL query string, it doesn't contain any parameter values. It would be handy to see the SQL statement including parameter values if you're debugging or logging queries, but these values are never combined with the query in its human-readable SQL form.
>
> The best way to debug your dynamic SQL statements is to log both the statement with parameter placeholders at prepare time and the parameter values at execute time.

The parameter is interpreted as a literal string, not an SQL keyword. In this example, the result is a syntax error.

Download SQL-Injection/anti/parameter.sql

```
SELECT * FROM Bugs ORDER BY date_reported 'DESC'
```

Stored Procedures

Use of stored procedures is another method that many software developers claim is proof against SQL Injection vulnerabilities. Typically, stored procedures contain fixed SQL statements, parsed when you define the procedure.

However, it's possible to use dynamic SQL in stored procedures unsafely. In the following example, the input_userid argument is interpolated into the SQL query verbatim, which is unsafe.

```
CREATE PROCEDURE UpdatePassword(input_password VARCHAR(20),
  input_userid VARCHAR(20))
BEGIN
  SET @sql = CONCAT('UPDATE Accounts
    SET password_hash = SHA2(', QUOTE(input_password), ')
    WHERE account_id = ', input_userid);
  PREPARE stmt FROM @sql;
  EXECUTE stmt;
END
```

Using dynamic SQL in a stored procedure is no more and no less safe than using dynamic SQL in application code. The input_userid argument can contain harmful content and produce an unsafe SQL statement:

```
UPDATE Accounts SET password_hash = SHA2('xyzzy')
WHERE account_id = 123 OR TRUE;
```

Data Access Frameworks

You might see advocates of data access frameworks claim that their library protects your code from SQL Injection risks. This is a false claim for any framework that allows you to write SQL statements as strings.

> **Practice Good Hygiene**
>
> After I gave a presentation on a PHP data access framework that I had developed, a member of the audience approached me and asked, "Does your framework prevent SQL Injection?" I answered that it provides functions for quoting strings and using query parameters.
>
> The young man looked puzzled. "But can it prevent SQL Injection?" he repeated. He was looking for an automatic way to ensure that he doesn't make a mistake that he doesn't know how to recognize himself.
>
> I told him the framework prevents SQL Injection like a toothbrush prevents cavities. You have to use it consistently to get the benefit.

No framework can force you to write safe SQL code. A framework may provide convenience functions to help you, but it's easy to bypass these functions and instead use common string manipulation to build an SQL statement unsafely.

21.3 How to Recognize the Antipattern

Practically every database application builds SQL statements dynamically. If you build any portion of an SQL statement by concatenating

> **Rule #31: Check the Back Seat**
>
> If you like to watch monster movies, you know that creatures like to hide behind the driver seat of your car and grab you after you get in. The lesson is that you shouldn't assume there's no danger inside a familiar space like your car.
>
> SQL Injection can take indirect forms. Even if you insert user-supplied data safely using query parameters, you might use that data later as you form dynamic SQL queries:
>
> ```php
> <?php
> $sql1 = "SELECT last_name FROM Accounts WHERE account_id = 123";
> $row = $pdo->query($sql1)->fetch();
> $sql2 = "SELECT * FROM Bugs WHERE MATCH(description) AGAINST ('"
>   . $row["last_name"] .  "')";
> ```
>
> What would happen in the previous query if the user had spelled their name as *O'Hara*—or if they had deliberately entered their name to contain SQL syntax?

strings together or interpolating variables into strings, then the statement potentially exposes your application to SQL Injection attacks.

SQL Injection vulnerabilities are so common that you should assume that you have some in any application that uses SQL, unless you've just completed a code review specifically to find and correct these issues.

21.4 Legitimate Uses of the Antipattern

This antipattern is different from most of the others in this book, in that there aren't any legitimate reasons for allowing your application to have a security vulnerability because of SQL Injection. It's your responsibility as a software developer to write code defensively and to help your peers to do so as well. Software is only as secure as its weakest link—make sure you're not responsible for that weakest link!

21.5 Solution: Trust No One

There is no single technique for securing your SQL code. You should learn all of the following techniques and use them in appropriate cases.

Filter Input

Instead of wondering whether some input contains harmful content, you should strip away any characters that aren't valid for that input. That is, if you need an integer, use only the part of the content that comprises an integer. The best way to do this depends on your programming language; for example, in PHP, use the filter extension:

Download SQL-Injection/soln/filter.php

```php
<?php
$bugid = filter_input(INPUT_GET, "bugid", FILTER_SANITIZE_NUMBER_INT);
$sql = "SELECT * FROM Bugs WHERE bug_id = {$bugid}";
$stmt = $pdo->query($sql);
```

You can use type casting functions for simple cases like numbers:

Download SQL-Injection/soln/casting.php

```php
<?php
$bugid = intval($_GET["bugid"]);
$sql = "SELECT * FROM Bugs WHERE bug_id = {$bugid}";
$stmt = $pdo->query($sql);
```

You can also use regular expressions to match safe substrings, filtering out illegitimate content:

Download SQL-Injection/soln/regexp.php

```php
<?php
$sortorder = "date_reported"; // default

if (preg_match("/[_[:alnum:]]+/", $_GET["order"], $matches)) {
  $sortorder = $matches[1];
}

$sql = "SELECT * FROM Bugs ORDER BY {$sortorder}";
$stmt = $pdo->query($sql);
```

Parameterize Dynamic Values

When the dynamic parts of your query are simple values, you should use query parameters to separate them from SQL expressions.

Download SQL-Injection/soln/parameter.php

```php
<?php
$sql = "UPDATE Accounts SET password_hash = SHA2(?) WHERE account_id = ?";
$stmt = $pdo->prepare($sql);
$params = array($_REQUEST["password"], $_REQUEST["userid"]);
$stmt->execute($params);
```

We saw examples in the "Antipattern" section that a parameter can substitute only for a single value. If you add the parameter values after

the RDBMS parses the SQL statement, no SQL Injection attack can change the syntax of a parameterized query. Even if an attacker tries to use a malicious parameter value such as *123 OR TRUE*, the RDBMS interprets the parameter as a value. At worst, the query fails to apply to any rows; it's not likely to apply to the wrong rows. The malicious value would result in a relatively safe SQL statement equivalent to the following:

Download **SQL-Injection/soln/parameter.sql**

```
UPDATE Accounts SET password_hash = SHA2('xyzzy')
WHERE account_id = '123 OR TRUE'
```

You should use query parameters when you need to combine application variables as literal values in SQL expressions.

Quoting Dynamic Values

Query parameters are usually the best solution, but in rare cases a query with parameter placeholders causes the query optimizer to make odd decisions about which indexes to use.

For example, suppose you have a column in the Accounts table called is_active. This column stores a true value for 99 percent of the rows, giving it an uneven distribution of values. A query that searches for is_active = false would benefit from an index, but it would be a waste to read the index for a query searching for is_active = true. However, if you used a parameter in the expression is_active = ?, the optimizer can't know which value you will supply when you execute the prepared query, so it's liable to choose the wrong optimization plan.

In exotic cases like this, it could be better to interpolate values directly into the SQL statement, in spite of the general recommendation to use query parameters. If you do this, you should quote the strings carefully.

Download **SQL-Injection/soln/interpolate.php**

```php
<?php
$quoted_active = $pdo->quote($_REQUEST["active"]);
$sql = "SELECT * FROM Accounts WHERE is_active = {$quoted_active}";
$stmt = $pdo->query($sql);
```

Make sure you use a function that is mature and well-tested against obscure SQL security issues. Most data access libraries include such a string-quoting function. For example, in PHP, use PDO::quote(). Don't try to implement your own quoting function unless you have studied the security risks thoroughly.

Parameterizing an IN() Predicate

We've seen that you can't pass a comma-separated string in a single parameter. You need as many parameters as the number of items in your list.

For example, say you need to query six bugs by their primary keys, which you have in an array variable $bug_list:

```php
<?php
$sql = "SELECT * FROM Bugs WHERE bug_id IN (?, ?, ?, ?, ?, ?)");
$stmt = $pdo->prepare($sql);
$stmt->execute($bug_list);
```

This works only if you have exactly six items in $bug_list, matching the number of parameter placeholders. You should build the SQL IN() predicate dynamically, using a number of placeholders equal to the number of items in $bug_list.

The following example in PHP uses some built-in array functions to produce an array of placeholders the same length as $bug_list and then joins that array with comma separators before interpolating it into the SQL expression.

```php
<?php
$sql = "SELECT * FROM Bugs WHERE bug_id IN ("
    . join(",", array_fill(0, count($bug_list), "?")) . ")";
$stmt = $pdo->prepare($sql);
$stmt->execute($bug_list);
```

Use this technique to parameterize a list of values.

Isolate User Input from Code

Query parameters and escaping techniques help you combine literal values into SQL expressions, but they don't help with other parts of a statement, such as table or column identifiers or SQL keywords. You need another solution to make these parts of a query dynamic.

Suppose your users want to choose how to sort lists of bugs, for instance by status or by date created. They also want to choose the direction of sorting.

Download SQL-Injection/soln/orderby.sql

```sql
SELECT * FROM Bugs ORDER BY status ASC
SELECT * FROM Bugs ORDER BY date_reported DESC
```

In the following example, a PHP script accepts request parameters order and dir, and your code interpolates these user choices into the SQL query to be a column name and a keyword.

Download SQL-Injection/soln/mapping.php

```php
<?php
$sortorder = $_REQUEST["order"];
$direction = $_REQUEST["dir"];
$sql = "SELECT * FROM Bugs ORDER BY $sortorder $direction";
$stmt = $pdo->query($sql);
```

The script assumes that order contains the name of a column and that dir contains either *ASC* or *DESC*. This is not a safe assumption, because a user can send any parameter values in a web request.

Instead, you can use the request parameters to look up predefined values and then use these values in your SQL query.

1. Declare a $sortorders array that maps user choices as keys and SQL column names as values. Declare a $directions array that maps user choices as keys and SQL keywords ASC and DESC as values.

   Download SQL-Injection/soln/mapping.php

   ```php
   $sortorders = array( "status" => "status", "date" => "date_reported" );
   $directions = array( "up" => "ASC", "down" => "DESC" );
   ```

2. Set variables $sortorder and $dir to default values in case the user's choices aren't in the arrays.

   Download SQL-Injection/soln/mapping.php

   ```php
   $sortorder = "bug_id";
   $direction = "ASC";
   ```

3. If the user's choices match array keys you declared in $sortorders and $directions, use the corresponding values.

   Download SQL-Injection/soln/mapping.php

   ```php
   if (array_key_exists($_REQUEST["order"], $sortorders)) {
     $sortorder = $sortorders[ $_REQUEST["order"] ];
   }

   if (array_key_exists($_REQUEST["dir"], $directions)) {
     $direction = $directions[ $_REQUEST["dir"] ];
   }
   ```

4. Now it's safe to use the $sortorder and $direction variables in your SQL query, because they can contain only values you declared in your code.

```php
$sql = "SELECT * FROM Bugs ORDER BY {$sortorder} {$direction}";
$stmt = $pdo->query($sql);
```

Using this technique has several advantages:

- You never combine user input with your SQL query, so you reduce the risk of SQL Injection.

- You can make any part of an SQL statement dynamic, including identifiers, SQL keywords, and even entire expressions.

- You have an easy and efficient way to validate user choices.

- You decouple the internal details of your database queries from the user interface.

The choices are hard-coded in your application, but this is appropriate for table names, column names, and SQL keywords. Choices over the full range of strings or numbers are typical for data values, but not for identifiers or syntax.

Get a Buddy to Review Your Code

The best way to catch flaws is to get another pair of eyes to look at it. Ask a teammate who is familiar with SQL Injection risks to help you inspect your code. Don't let pride or ego keep you from doing the right thing—you may be embarrassed now over missing a coding mistake, but would you rather have to admit responsibility later for a security flaw that allowed hackers to exploit your website?

In an inspection for SQL Injection, use the following guidelines:

1. Find SQL statements that are formed using application variables, string concatenation, or replacement.

2. Trace the origin of all dynamic content used in your SQL statements. Find any data that comes from an external source, such as user input, files, environment, web services, third-party code, or even a string fetched from the database.

3. Assume any external content is potentially hazardous. Use filters, validators, and mapping arrays to transform untrusted content.

4. Combine external data into your SQL statements using query parameters or robust escaping functions.

5. Don't forget to inspect your stored procedures and other places where you may find dynamic SQL statements.

Code inspection is the most accurate and economical way to find SQL Injection flaws. You should budget your time for this and treat it as a mandatory activity. You can also return the favor by inspecting your teammates' code.

Let users input values, but never let users input code.

Chapter 22

Pseudokey Neat-Freak

Your manager approaches you, holding two report printouts. "The bean counters are saying we have discrepancies between this quarter's report and last quarter's. I'm looking at them, and they're absolutely right. Most of the later assets have disappeared. What happened?"

You look at the reports, and the pattern of discrepancies rings a bell. "No, everything is still there. You asked me to clean up the rows in the database so there are no missing rows. You said the accountants kept asking you questions about missing assets, because of gaps in the numbering.

"So, I renumbered some of the rows to make them all fit into the places where there were missing rows before. There aren't any missing rows now—every number between 1 and about 12,340 is used. They're all still there, but some have just changed number and moved up. You told me to do this."

Your manager shakes his head. "But that's not what I want. The accountants have to track depreciation by the asset numbers. The number for each piece of equipment has to stay the same in each quarterly report. Besides, all the asset ID numbers are printed on labels on each piece. It'd take weeks to relabel everything in the company. Can you please change all the ID numbers back to their original values?"

You want to be cooperative, so you turn back to your keyboard to start working, but suddenly you think of a new problem. "What about new assets we bought this month, after I consolidated the asset IDs? The new assets have been assigned ID values that were in use before I did the renumbering. If I change the asset IDs back to their old values, what should I do about the duplicates?"

22.1 Objective: Tidy Up the Data

There's a certain type of person who is unnerved by a gap in a series of numbers.

bug_id	status	product_name
1	OPEN	Open RoundFile
2	FIXED	ReConsider
4	OPEN	ReConsider

On one hand, it's understandable to be concerned, because it's unclear what happened to the row with bug_id 3. Why didn't the query return that bug? Did the database lose it? What was in that bug? Was the bug reported by one of our important customers? Am I going to be held responsible for the lost data?

The objective of one who practices the *Pseudokey Neat-Freak* antipattern is to resolve these troubling questions. This person is accountable for data integrity issues, but typically they don't have enough understanding of or confidence in the database technology to feel confident of the generated report results.

22.2 Antipattern: Filling in the Corners

Most people's first reaction to a perceived gap is naturally to want to seal the gap. There are two ways you might do this.

Assigning Numbers Out of Sequence

Instead of allocating a new primary key value using the automatic pseudokey mechanism, you might want to make any new row use the first unused primary key value. This way, as you insert data, you naturally make gaps fill in.

bug_id	status	product_name
1	OPEN	Open RoundFile
2	FIXED	ReConsider
4	OPEN	ReConsider
3	NEW	Visual TurboBuilder

However, you have to run an unnecessary self-join query to find the lowest unused value:

Download Neat-Freak/anti/lowest-value.sql

```
SELECT b1.bug_id + 1
FROM Bugs b1
LEFT OUTER JOIN Bugs AS b2 ON (b1.bug_id + 1 = b2.bug_id)
WHERE b2.bug_id IS NULL
ORDER BY b1.bug_id LIMIT 1;
```

Earlier in the book, we looked at a concurrency issue when you try to allocate a unique primary key value by running a query such as SELECT MAX(bug_id)+1 FROM Bugs.[1] This has the same flaw when two applications may try to find the lowest unused value at the same time. As both try to use the same value as a primary key value, one succeeds, and the other gets an error. This method is both inefficient and prone to errors.

Renumbering Existing Rows

You might find it's more urgent to make the primary key values be contiguous, and waiting for new rows to fill in the gaps won't fix the issue quickly enough. You might think to use a strategy of updating the key values of existing rows to eliminate gaps and make all the values contiguous. This usually means you find the row with the highest primary key value and update it with the lowest unused value. For example, you could update the value 4 to 3:

Download Neat-Freak/anti/renumber.sql

```
UPDATE Bugs SET bug_id = 3 WHERE bug_id = 4;
```

bug_id	status	product_name
1	NEW	Open RoundFile
2	FIXED	ReConsider
3	DUPLICATE	ReConsider

To accomplish this, you need to find an unused key value using a method similar to the previous one for inserting new rows. You also need to run the UPDATE statement to reassign the primary key value. Either one of these steps is susceptible to concurrency issues. You need to repeat the steps many times to fill a wide gap in the numbers.

You must also propagate the changed value to all child records that reference the rows you renumber. This is easiest if you declared for-

---

1. See the sidebar on page 60.

eign keys with the ON UPDATE CASCADE option, but if you didn't, you would have to disable constraints, update all child records manually, and restore the constraints. This is a laborious, error-prone process that can interrupt service in your database, so if you feel you want to avoid it, you're right.

Even if you do accomplish this cleanup, it's short-lived. When a pseudokey generates a new value, the value is greater than the last value it generated (even if the row with that value has since been deleted or changed), *not* the highest value currently in the table, as some database programmers assume. Suppose you update the row with the greatest bug_id value 4 to the lower unused value to fill a gap. The next row you insert using the default pseudokey generator will allocate 5, leaving a new gap at 4.

Manufacturing Data Discrepancies

Mitch Ratcliffe said, "A computer lets you make more mistakes faster than any other human invention in human history. . . with the possible exception of handguns and tequila."[2]

The story at the beginning of this chapter describes some hazards of renumbering primary key values. If another system external to your database depends on identifying rows by their primary keys, then your updates invalidate the data references in that system.

It's not a good idea to reuse the row's primary key value, because a gap could be the result of deleting or rolling back a row for a good reason. For example, suppose a user with account_id 789 is barred from your system for sending offensive emails. Your policies require you to delete the offender's account, but if you recycle primary keys, you would subsequently assign 789 to another user. Since some offensive emails are still waiting to be read by some recipients, you could get further complaints about *account 789*. Through no fault of his own, the poor user who now has that number catches the blame.

Don't reallocate pseudokey values just because they seem to be unused.

---

2. *MIT Technology Review*, April 1992.

22.3 How to Recognize the Antipattern

The following quotes can be hints that someone in your organization is about to use the Pseudokey Neat-Freak antipattern.

- "How can I reuse an autogenerated identity value after I roll back an insert?"

  Pseudokey allocation doesn't roll back; if it did, the RDBMS would have to allocate pseudokey values within the scope of a transaction. This would cause either race conditions or blocking when multiple clients are inserting data concurrently.

- "What happened to bug_id 4?"

  This is an expression of misplaced anxiety over unused numbers in the sequence of primary keys.

- "How can I query for the first unused ID?"

  The reason to do this search is almost certainly to reassign the ID.

- "What if I run out of numbers?"

  This is used as a justification for reallocating unused ID values.

22.4 Legitimate Uses of the Antipattern

There's no reason to change the value of a pseudokey, since the value should have no significance anyway. If the values in the primary key column carry some meaning, then this column is a *natural key*, not a pseudokey. It's not unusual to change values in a natural key.

22.5 Solution: Get Over It

The values in any primary key must be unique and non-null so you can use them to reference individual rows, but that's the only rule—they don't have to be consecutive numbers to identify rows.

Numbering Rows

Most pseudokey generators return numbers that look almost like row numbers, because they're *monotonically increasing* (that is, each successive value is one greater than the preceding value), but this is only a coincidence of their implementation. Generating values in this way is a convenient way to ensure uniqueness.

Don't confuse row numbers with primary keys. A primary key identifies one row in one table, whereas row numbers identify rows in a result set. Row numbers in a query result set don't correspond to primary key values in the table, especially when you use query operations like JOIN, GROUP BY, or ORDER BY.

There are good reasons to use row numbers, for example to return a subset of rows from a query result. This is often called *pagination*, like a page of an Internet search. To select a subset in this way, you need to use true row numbers that are increasing and consecutive, regardless of the form of the query.

SQL:2003 specifies *window functions* including ROW_NUMBER(), which returns consecutive numbers specific to a query result set. A common use of row numbering is to limit the query result to a range of rows:

Download Neat-Freak/soln/row_number.sql

```
SELECT t1.* FROM
  (SELECT a.account_name, b.bug_id, b.summary,
     ROW_NUMBER() OVER (ORDER BY a.account_name, b.date_reported) AS rn
   FROM Accounts a JOIN Bugs b ON (a.account_id = b.reported_by)) AS t1
WHERE t1.rn BETWEEN 51 AND 100;
```

These functions are currently supported by many leading brands of database, including Oracle, Microsoft SQL Server 2005, IBM DB2, PostgreSQL 8.4, and Apache Derby.

MySQL, SQLite, Firebird, and Informix don't support SQL:2003 window functions, but they have proprietary syntax you can use in the scenario presented in this section. MySQL and SQLite support a LIMIT clause, and Firebird and Informix support a query option with keywords FIRST and SKIP.

Using GUIDs

You could also generate random pseudokey values, as long as you don't use any number more than once. Some databases support a *globally unique identifier* (GUID) for this purpose.

A GUID is a pseudorandom number of 128 bits (usually represented by 32 hexadecimal digits). For practical purposes, a GUID is unique, so you can use it to generate a pseudokey.

Are Integers a Nonrenewable Resource?

Another misconception related to the Pseudokey Neat-Freak antipattern is the idea that a monotonically increasing pseudokey generator eventually exhausts the set of integers, so you must take precautions not to waste values.

At first glance, this seems sensible. In mathematics, the set of integers is countably infinite, but in a database, any data type has a finite number of values. A 32-bit integer can represent a maximum of 2^{32} distinct values. It's true that each time you allocate a value for a primary key, you're one step closer to the last one.

But do the math: if you generate unique primary key values as you insert 1,000 rows per second, 24 hours per day, you can continue for 136 years before you use all values in an unsigned 32-bit integer.

If that doesn't meet your needs, then use a 64-bit integer. Now you can use *1 million* integers per second continuously for 584,542 years.

It's very unlikely that you will run out of integers!

The following example uses Microsoft SQL Server 2005 syntax:

Download **Neat-Freak/soln/uniqueidentifier-sql2005.sql**

```sql
CREATE TABLE Bugs (
  bug_id UNIQUEIDENTIFIER DEFAULT NEWID(),
  -- . . .
);

INSERT INTO Bugs (bug_id, summary)
VALUES (DEFAULT, 'crashes when I save');
```

This creates a row like the following:

bug_id	summary
0xff19966f868b11d0b42d00c04fc964ff	Crashes when I save

You gain at least two advantages over traditional pseudokey generators when you use GUIDs:

- You can generate pseudokeys on multiple database servers concurrently without using the same values.

- No one will complain about gaps—they'll be too busy complaining about typing thirty-two hex digits for primary key values.

The latter point leads to some of the disadvantages:

- The values are long and hard to type.

- The values are random, so you can't infer any pattern or rely on a greater value indicating a more recent row.

- Storing a GUID requires 16 bytes. This takes more space and runs more slowly than using a typical 4-byte integer pseudokey.

The Most Important Problem

Now that you know the problems caused by renumbering pseudokeys and some alternative solutions for related goals, you still have one big problem to solve: how do you fend off an order from a boss who wants you to tidy up the database by closing the gaps in a pseudokey? This is a problem of communication, not technology. Nevertheless, you might need to *manage your manager* to defend the data integrity of your database.

- *Explain the technology.* Honesty is usually the best policy. Be respectful and acknowledge the feeling behind the request. For example, tell your manager this:

  "The gaps do look strange, but they're harmless. It's normal for rows to be skipped, rolled back, or deleted from time to time. We allocate a new number for each new row in the database, instead of writing code to figure out which old numbers we can reuse safely. This makes our code cheap to develop, makes it faster to run, and reduces errors."

- *Be clear about the costs.* Changing the primary key values seems like a trivial task, but you should give realistic estimates for the work it will take to calculate new values, write and test code to handle duplicate values, cascade changes throughout the database, investigate the impact to other systems, and train users and administrators to manage the new procedures.

Most managers prioritize based on cost of a task, and they should back down from requesting frivolous, micro-optimizing work when they're confronted with the real cost.

- *Use natural keys.* If your manager or other users of the database insist on interpreting meaning in the primary key values, then let there be meaning. Don't use pseudokeys—use a string or a number that encodes some identifying meaning. Then it's easier to explain any gaps within the context of the meaning of these natural keys.

  You can also use both a pseudokey and another attribute column you use as a natural identifier. Hide the pseudokey from reports if gaps in the numeric sequence make readers anxious.

Use pseudokeys as unique row identifiers; they're not row numbers.

Chapter 23

See No Evil

"I found *another* bug in your product," the voice on the phone said.

I got this call while working as a technical support engineer for an SQL RDBMS in the 1990s. We had one customer who was well-known for making spurious reports against our database. Nearly all of his reports turned out to be simple mistakes on his part, not bugs.

"Good morning, Mr. Davis. Of course, we'd like to fix any problem you find," I answered. "Can you tell me what happened?"

"I ran a query against your database, and nothing came back." Mr. Davis said sharply. "But I know the data is in the database—I can verify it in a test script."

"Was there any problem with your query?" I asked. "Did the API return any error?"

Davis replied, "Why would I look at the return value of an API function? The function should just run my SQL query. If it returns an error, that indicates your product has a bug in it. If your product didn't have bugs, there would be no errors. I shouldn't have to work around your bugs."

I was stunned, but I had to let the facts speak for themselves. "OK, let's try a test. Copy and paste the *exact* SQL query from your code into the query tool, and run it. What does it say?" I waited for him.

"Syntax error at SELCET." After a pause, he said, "You can close this issue," and he hung up abruptly.

Mr. Davis was the sole developer for an air traffic control company, writing software that logged data about international airplane flights. We heard from him every week.

23.1 Objective: Write Less Code

Everyone wants to write *elegant code*. That is, we want to do cool work with little code. The cooler the work is and the less code it takes us, the greater the ratio of elegance. If we can't make our work cooler, it stands to reason that at least we can improve the elegance ratio of coolness to code volume by doing the same work with less code.

That's a superficial reason, but there are more rational reasons to write concise code:

- We'll finish coding a working application more quickly.

- We'll have less code to test, to document, or to have peer-reviewed.

- We'll have fewer bugs if we have fewer lines of code.

It's therefore an instinctive priority for programmers to eliminate any code they can, especially if that code fails to increase coolness.

23.2 Antipattern: Making Bricks Without Straw

Developers commonly practice the See No Evil antipattern in two forms: first, ignoring the return values of a database API; and second, reading fragments of SQL code interspersed with application code. In both cases, developers fail to use information that is easily available to them.

Diagnoses Without Diagnostics

Download See-No-Evil/anti/no-check.php

```php
<?php
❶   $pdo = new PDO("mysql:dbname=test;host=db.example.com",
        "dbuser", "dbpassword");
    $sql = "SELECT bug_id, summary, date_reported FROM Bugs
        WHERE assigned_to = ? AND status = ?";
❷   $stmt = $dbh->prepare($sql);
❸   $stmt->execute(array(1, "OPEN"));
❹   $bug = $stmt->fetch();
```

This code is concise, but there are several places in this code where status values returned from functions could indicate a problem, but you'll never know about it if you ignore the return values.

Probably the most common error from a database API occurs when you try to create a database connection, for example at ❶. You could accidentally mistype the database name or server hostname or you could get the user or password wrong, or the database server could

your users will see this utterly blank screen; then you will get the phone calls
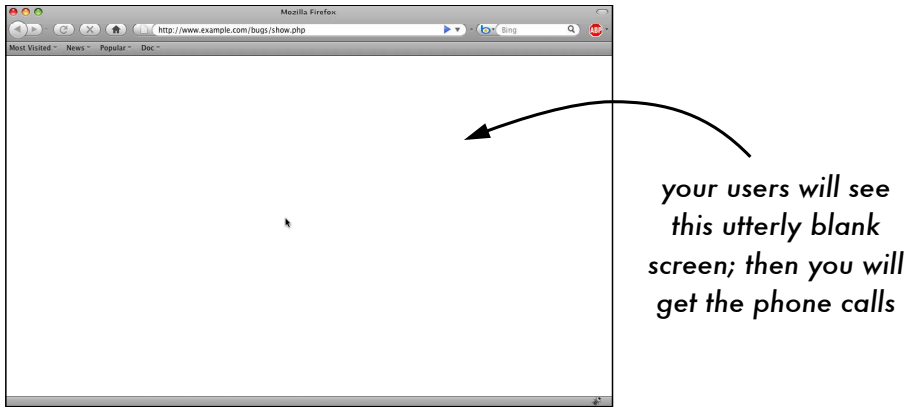
Figure 23.1: A fatal error in PHP results in a blank screen

be unreachable. An error with instantiating a PDO connection throws an exception, which would terminate the example script shown previously.

The call to prepare() at ❷ could return false if you have a simple syntax error caused by a typo or an imbalanced parenthesis or a misspelled column name. If this happens, the attempt to call execute() as a method of $stmt at ❸ would be a fatal error because the value false isn't an object.

```
PHP Fatal error:  Call to a member function execute() on a non-object
```

The call to execute() could also fail, for example, because the statement violates a constraint or exceeds access privileges. The method also returns false on error.

The call to fetch() at ❹ would return false if any other error occurs, such as if the connection to the RDBMS fails.

Programmers with attitudes like Mr. Davis aren't uncommon. They may feel that checking return values and exceptions adds nothing to their code, because those cases aren't supposed to happen anyway. Also, the extra code is repetitive and makes an application ugly and hard to read. It definitely adds no coolness.

But users don't see the code; they only see the output. When a fatal error goes unhandled, the user may see only a blank white screen, as in Figure 23.1, or else an incomprehensible exception message. When

this happens, it's little consolation that the application code is tidy and concise.

Lines Between the Reading

Another common bad habit that fits the See No Evil antipattern is to debug by staring at application code that builds an SQL query as a string. This is difficult because it's hard to visualize the resulting SQL string after you build it with application logic, string concatenation, and extra content from application variables. Trying to debug in this way is like trying to solve a jigsaw puzzle without looking at the photo on the box.

For a simple example, let's look at a type of question I see frequently from developers. The following code builds a query conditionally by concatenating a WHERE clause if the script needs to search for a specific bug instead of a collection of bugs.

Download See-No-Evil/anti/white-space.php

```php
<?php
$sql  = "SELECT * FROM Bugs";
if ($bug_id) {
    $sql .= "WHERE bug_id = " . intval($bug_id);
}
$stmt = $pdo->prepare($sql);
```

Why would the query in this example give an error? The answer is clearer if you look at the full $sql string resulting from the concatenation:

Download See-No-Evil/anti/white-space.sql

```sql
SELECT * FROM BugsWHERE bug_id = 1234
```

There's no whitespace between Bugs and WHERE, which gives the query invalid syntax, as though it were reading a table called BugsWHERE, followed by an SQL expression in an invalid context. The code concatenated the strings with no space between them.

Developers waste an unbelievable amount of time and energy trying to debug problems like this by looking at the code that builds the SQL, instead of looking at the SQL itself.

23.3 How to Recognize the Antipattern

Though you might think that the absence of code is by nature difficult to spot, many modern IDE products highlight instances in your

code where you ignore a return value from a function that returns one or where your code calls a function but neglects to handle a checked exception.[1] You could also encounter the See No Evil antipattern if you hear phrases like the following:

- "My program crashes after I query the database."

  Often the crash happens because your query failed, and you tried to use the result in an illegal manner, such as calling a method on a nonobject or dereferencing a null pointer.

- "Can you help me find my SQL error? Here's my code. . ."

  First, start by looking at the SQL, not the code that builds it.

- "I don't bother cluttering up my code with error handling."

  Some computer scientists have estimated that up to 50 percent of the lines of code in a robust application are devoted to handling error cases. This may seem like a lot, unless you think of all the steps that you could include under error handling: detecting, classifying, reporting, and compensating. It's important for any software to be able to do all that.

23.4 Legitimate Uses of the Antipattern

You can omit error checking when there's really nothing for you to do in response to the error. For example, the close() function for a database connection returns a status, but if your application is about to finish and exit anyway, it's likely that the resources for that connection will be cleaned up regardless.

Exceptions in object-oriented languages allow you to trigger an exception without being responsible for handling it. Your code trusts that whatever code called yours is the code that's responsible for handling the exception. Your code therefore can allow the exception to pass back up the calling stack.

---

1. A checked exception is one that a function's signature declares, so you know that the function might throw that exception type.

23.5 Solution: Recover from Errors Gracefully

Anyone who enjoys dancing knows that missteps are inevitable. The secret to remaining graceful is to know how to recover. Give yourself a chance to notice the cause of the mistake. Then you can react quickly and seamlessly, getting back into rhythm before anyone has noticed your gaffe.

Maintain the Rhythm

Checking return status and exceptions from database API calls is the best way to ensure that you haven't missed a step. The following example shows code that checks the status after each call that could cause an error:

Download See-No-Evil/soln/check.php

```php
<?php
try {
    $pdo = new PDO("mysql:dbname=test;host=localhost",
        "dbuser", "dbpassword");
❶ } catch (PDOException $e) {
    report_error($e->getMessage());
    return;
}

$sql = "SELECT bug_id, summary, date_reported FROM Bugs
    WHERE assigned_to = ? AND status = ?";

❷ if (($stmt = $pdo->prepare($sql)) === false) {
    $error = $pdo->errorInfo();
    report_error($error[2]);
    return;
}

❸ if ($stmt->execute(array(1, "OPEN")) === false) {
    $error = $stmt->errorInfo();
    report_error($error[2]);
    return;
}

❹ if (($bug = $stmt->fetch()) === false) {
    $error = $stmt->errorInfo();
    report_error($error[2]);
    return;
}
```

The code at ❶ catches the exception that is thrown if a database connection fails. The other functions return false when there's a problem.

After checking for a problem at ❷, ❸, and ❹, you can get more information from the database connection object or the statement object.

Retrace Your Steps

It's also important to use the actual SQL query to debug a problem, instead of the code that produces an SQL query. Many simple mistakes, such as misspellings or imbalanced quotes or parentheses, are apparent instantly, even though they're obscure and puzzling otherwise.

- Build your SQL query in a variable, instead of building it ad hoc in the arguments of the API method to prepare the query. This gives you the opportunity to examine the variable before you use it.

- Choose a place to output SQL that is not part of your application output, such as a log file, an IDE debugger console, or a browser extension to show diagnostic output.[2]

- Do not print the SQL query within HTML comments of a web application's output. Any user can view your page source. Reading the SQL query gives hackers a lot of knowledge about your database structure.

Using an object-relational mapping (ORM) framework that builds and executes SQL queries transparently can make debugging complicated. If you don't have access to the content of the SQL query, how can you observe it for debugging? Some ORM frameworks solve this by sending generated SQL to a log.

Finally, most database brands provide their own logging mechanism on the database servers instead of in application client code. If you can't enable SQL logging in the application, you can still monitor queries as the database server executes them.

Troubleshooting code is already hard enough.
        Don't hinder yourself by doing it blind.

---

2. Firebug (http://getfirebug.com/) is a good example.

Chapter 24

Diplomatic Immunity

One of my earliest jobs gave me a lesson in the importance of using software engineering best practices, after a tragic accident left me responsible for an important database application.

I interviewed for a contract job at Hewlett-Packard to develop and maintain an application on UNIX, written in C with HP ALLBASE/SQL. The manager and staff interviewing me told me sadly that their programmer who had worked on that application was killed in a traffic accident. No one else in their department knew how to use UNIX or anything about the application.

After I started the job, I found that the developer had never written documentation or tests for this application, and he never used a source code control system or even code comments. All his code resided in a single directory, including code that was part of the live system, code that was under development, and code that was no longer used.

This project had high *technical debt*—a consequence of using shortcuts instead of best practices.[1] Technical debt causes risk and extra work in a project until you pay it off by refactoring, testing, and documenting.

I worked for six months to organize and document the code for what was really a fairly modest application, because I had to spend a lot of my time supporting its users and continuing development.

There was obviously no way that I could ask my predecessor to help me come up to speed on the project. The experience really demonstrated the impact of letting technical debt get out of control.

---

1. Ward Cunningham coined this metaphor in his experience report for OOPSLA 1992 (http://c2.com/doc/oopsla92.html).

24.1   Objective: Employ Best Practices

Professional programmers strive to use good software engineering habits in their projects, such as the following:

- Keeping application source code under revision control using tools such as Subversion or Git.
- Developing and running automated unit tests or functional tests for applications.
- Writing documentation, specifications, and code comments to record the requirements and implementation strategies of an application.

The time you take to develop software using best practices is a net win, because it reduces a lot of needless or repetitive work. Most experienced developers know that sacrificing these practices for the sake of expediency is a recipe for failure.

24.2   Antipattern: Make SQL a Second-Class Citizen

Even among developers who accept best practices when developing application code, there's a tendency to think of database code as exempt from these practices. I call this antipattern *Diplomatic Immunity* because it assumes that the rules of application development don't apply to database development.

Why would developers make this assumption? The following are some possible reasons:

- The role of software engineer and database administrator are separate in some companies. The DBA typically works with several teams of programmers, so there's a perception that she's not a full-time member of any one of these teams. She's treated like a visitor, and she's not subject to the same responsibilities as the software engineers.
- The SQL language used for relational databases is different from conventional programming. Even the way we invoke SQL statements as a specialized language within application code suggests a kind of guest-like status.
- Advanced IDE tools are popular for application code languages, making editing, testing, and source control quick and painless. But tools for database development are not as advanced, or at least not as widely used. Developers can code applications with

best practices easily, but applying these practices to SQL feels clumsy by comparison. Developers tend to find other things to do.

- In IT, it's ordinary for knowledge and operation of the database to be focused on one person—the DBA. Because the DBA is the only one who has access to the database server, she serves as a living knowledge base and source control system.

The database is the foundation of an application, and quality matters. You know how to develop application code with high quality, but you may be building your application on top of a database that has failed to solve the needs of the project or that no one understands. The risk is that you're developing an application only to find that you have to scrap it.

24.3 How to Recognize the Antipattern

You might think it's hard to show evidence of not doing something, but that isn't always true. The following are some telltale signs of cutting corners:

- "We are adopting the new engineering process—that is, a lightweight version of it."

  Lightweight in this context means that the team intends to skip some tasks that the engineering process calls for. Some of these may be legitimate to skip, but it could also be a euphemism for not following important best practices.

- "We don't need the DBA staff to attend training for our new source control system, since they don't use it anyway."

  Excluding some technical team members from training (and probably access) *ensures* that they won't use those tools.

- "How can I track usage of tables and columns in the database? There are some elements we don't know the purpose of, and we'd like to eliminate them if they're obsolete."

  You are not using the project documentation for the database schema. The document may be out-of-date, may be inaccessible, or may never have existed at all. Even if you don't know the purpose of some tables or columns, they might be important to someone, and you can't remove them.

- "Is there a tool to compare two database schema, report the differences, and create a script to alter one to match the other?"

If you don't follow a process of deploying changes to database schema, they can get out of sync, and then it's a complicated task to bring them back into order.

24.4 Legitimate Uses of the Antipattern

I do write documentation and tests, and I use source control and other good habits for any code I want to use more than once. But I also write code that is truly ad hoc, such as a one-time test of an API function to remind myself how to use it or an SQL query I write to answer a user's question.

A good guideline for whether code is really temporary is to delete it immediately after you've used it. If you can't bring yourself to do that, it's probably worth keeping. That's OK, but that means it's worth storing in source control and writing at least some brief notes about what the code is for and how to use it.

24.5 Solution: Establish a Big-Tent Culture of Quality

Quality is simply testing to most software developers, but that's only *quality control*—only part of the story. The full life cycle of software engineering involves *quality assurance*, which includes three parts:

1. Specify project requirements clearly and in writing.
2. Design and develop a solution for your requirements.
3. Validate and test that your solution matches the requirements.

You need to do all three of these to perform QA correctly, although in some software methodologies, you don't necessarily have to do them in that order.

You can achieve quality assurance in database development by following best practices in *documentation*, *source code control*, and *testing*.

Exhibit A: Documentation

There's no such thing as self-documenting code. Although it's true that a skilled programmer can decipher most code through a combination of careful analysis and experimentation, this is laborious.[2] Also, code can't tell you about missing features or unsolved problems.

---

2. If code were readable, why would we call it *code*?

You should document the requirements and implementation of a database just as you do application code. Whether you're the original designer of the database or you're inheriting a database designed by someone else, use the following checklist to document a database:

Entity-relationship diagram: The single most important piece of documentation for a database is an ER diagram showing the tables and their relationships. Several chapters in this book use a simple form of ER diagrams. More complex ER diagrams have notation for columns, keys, indexes, and other database objects.

Some diagramming software packages include elements for ER diagram notation. Some tools can even reverse-engineer an SQL script or a live database and produce an ER diagram.

One caveat is that databases can be complex and have so many tables that it's impractical to use a single diagram. In this case, you should decompose it into several diagrams. Usually you can choose natural subgroups of tables so each diagram is readable enough to be useful and not overwhelming to the reader.

Tables, columns, and views: You also need written documentation for your database, because an ER diagram isn't the right format to describe the purpose and usage of each table, column, and other object.

Tables need a description of what type of entity the table models. For example, Bugs, Products, and Accounts are pretty clear, but what about a lookup table like BugStatus or an intersection table like BugsProducts or a dependent table like Comments? Also, how many rows do you anticipate each table to have? What queries against this table do you expect? What indexes exist in this table?

Columns each have a name and a data type, but that doesn't tell the reader what the column's values mean. What values make sense in that column (it's rarely the full range of the data type)? For columns storing a quantitative value, what is the unit of measurement? Does the column allow nulls or not, and why? Does it have a unique constraint, and if so, why?

Views store frequently used queries against one or more tables. What made it worthwhile to create a given view? What application or user is expected to use the view? Was the view intended to abstract a complex relationship of tables? Does it exist as a way

to allow unprivileged users to query a subset of rows or columns in a privileged table? Is the view updatable?

Relationships: Referential integrity constraints implement dependencies between tables, but this might not tell everything that you intend the constraints to model. For example, `Bugs.reported_by` is not nullable, but `Bugs.assigned_to` is nullable. Does that mean a bug can be fixed before it's assigned? If not, what are the business rules for when the bug must be assigned?

In some cases, you may have implicit relationships but no constraints for them. Without documentation, it's hard to know where these relationships exist.

Triggers: Data validation, data transformation, and logging database changes are examples of tasks for a trigger. What business rules are you implementing in triggers?

Stored procedures: Document your stored procedures like an API. What problem is the procedure solving? Does a procedure perform any changes to data? What are the data types and meanings of the input and output parameters? Do you intend the procedure to replace a certain type of query to eliminate a performance bottleneck? Do you use the procedure to grant unprivileged users access to privileged tables?

SQL Security: What database users do you define for applications to use? What access privileges do each of these users have? What SQL roles do you provide, and which users can use them? Are any users designated for specific tasks, such as backups or reports? What system-level security provisions do you use, such as if the client must reach the RDBMS server via SSL? What measures do you take to detect and block attempts at illicit authentication, such as brute-force password guessing? Have you done a thorough code review for SQL Injection vulnerabilities?

Database infrastructure: This information is chiefly used by IT staff and DBAs, but developers need to know some of it too. What RDBMS brand and version do you operate? What is your database server hostname? Do you use multiple database servers, replication, clusters, proxies, and so on? What is your network organization and the port number used by the database server? What connection options do client applications need to use? What are

the database user passwords? What are your database backup policies?

Object-relational mapping: Your project may implement some database-handling logic in application code, as part of an layer of ORM-based code classes. What business rules are implemented in this way? Data validation, data transformation, logging, caching, or profiling?

Developers don't like to maintain engineering documentation. It's hard to write, it's hard to keep up-to-date, and it's dispiriting when few people read what you do write. But even battle-hardened, extreme programmers know that they need to document the database, even if they document no other part of their software.[3]

Trail of Evidence: Source Code Control

If your database server failed completely, how would you re-create a database? What's the best way to track a complex upgrade to your database design? How would you back out a change?

We know how we would use a source control system to manage application code, solving similar problems of software development. A project under source control should include *everything* you need to rebuild and redeploy the project if your existing deployment explodes. Source control also serves as a history of changes and an incremental backup so you can reverse any of these changes.

You can use source control with your database code and get similar benefits for development.

You should check into source control the files related to your database development, including the following:

Data definition scripts: All brands of database provide ways to execute *SQL scripts* containing `CREATE TABLE` and other statements that define the database objects.

Triggers and procedures: Many projects supplement application code with routines stored in the database. Your application probably won't work without these routines, so they count as part of your project's code.

---

3. For example, Jeff Atwood and Joel Spolsky see little value in documenting code, *except* for the database, in StackOverflow podcast #80, http://blog.stackoverflow.com/2010/01/podcast-80/.

Schema Evolution Tools

Your code is under source control, but your database isn't. Ruby on Rails popularized a technique called *migrations* to manage upgrades to a database instance under source control. Let's briefly see an example of an upgrade:

Write a script with code to upgrade a database by one step, based on Rails' abstract class for making database changes. Also write a downgrade function that reverses the changes from those in the upgrade function.

```ruby
class AddHoursToBugs < ActiveRecord::Migration
  def self.up
    add_column :bugs, :hours, :decimal
  end

  def self.down
    remove_column :bugs, :hours
  end
end
```

The Rails tool that runs migrations automatically creates a table to record the revision or revisions that apply to your current database instance. Rails 2.1 introduced changes to make this system more flexible, and subsequent versions of Rails may also change the way migrations work.

Create a new migration script for each schema alteration in the database. You accumulate a set of these migration scripts; each one can upgrade or downgrade the database schema one step. If you need to change your database to version 5, specify an argument to the migration tool.

```
$ rake db:migrate VERSION=5
```

There's a lot more to learn about migrations in *Agile Web Development with Rails, Third Edition* (RTH08) or http://guides.rubyonrails.org/migrations.html.

Most other web development frameworks, including Doctrine for PHP, Django for Python, and Microsoft ASP.NET, support features similar to Rails' migrations, either included with the framework or available as a community project.

Migrations automate a lot of tedious work of synchronizing a database instance with the structure expected in a given revision of your project under source code control. But they aren't perfect. They handle only a few simple types of schema changes, and they basically implement a revision system on top of your conventional source control.

Bootstrap data: Lookup tables may contain some set of data that represents an initial state of your database, before any users enter new data. You should keep bootstrap data to help if you need to re-create a database from your project source. Also called *seed data.*

ER diagrams and documentation: These files aren't code, but they're closely tied to the code, describing database requirements, implementation, and integration with the application. As the project evolution results in changes to both the database and the application, you should keep these files up-to-date. Make sure the documents describe the current designs.

DBA scripts: Most projects have a collection of data-handling jobs that run outside the application. These include tasks for import/export, synchronization, reporting, backups, validation, testing, and so on. These may be written as SQL scripts, not part of a conventional application programming language.

Make sure your database code files are associated with the application code that uses that database. Part of the benefit of using source control is that if you check out your project from source control given a certain revision number, date, or milestone, the files should work together. Use the same source control repository for both application code and database code.

Burden of Proof: Testing

The final part of quality assurance is quality control—validating that your application does what it set out to do. Most professional developers are familiar with techniques to write automated tests to validate application code behavior. One important principle of testing is *isolation*, testing only one part of the system at a time so that if a defect exists, you can narrow down where it exists as precisely as possible.

We can extend the practice of isolation testing to the database by validating the database structure and behavior independently from your application code.

The following example shows a unit test script using the PHPUnit test framework:[4]

Download Diplomatic_immunity/DatabaseTest.php

```php
<?php
require_once "PHPUnit/Framework/TestCase.php";

class DatabaseTest extends PHPUnit_Framework_TestCase
{
    protected $pdo;

    public function setUp()
    {
        $this->pdo = new PDO("mysql:dbname=bugs", "testuser", "xxxxxx");
    }

    public function testTableFooExists()
    {
        $stmt = $this->pdo->query("SELECT COUNT(*) FROM Bugs");
        $err = $this->pdo->errorInfo();
        $this->assertType("object", $stmt, $err[2]);
        $this->assertEquals("PDOStatement", get_class($stmt));
    }

    public function testTableFooColumnBugIdExists()
    {
        $stmt = $this->pdo->query("SELECT COUNT(bug_id) FROM Bugs");
        $err = $this->pdo->errorInfo();
        $this->assertType("object", $stmt, $err[2]);
        $this->assertEquals("PDOStatement", get_class($stmt));
    }

    static public function main()
    {
        $suite  = new PHPUnit_Framework_TestSuite(__CLASS__);
        $result = PHPUnit_TextUI_TestRunner::run($suite);
    }

}

DatabaseTest::main();
```

You can use the following checklist for tests that validate your database:

Tables, columns, views: You should test that tables and views you ex-
pect to exist in the database do exist. Each time you enhance
the database with a new table, view, or column, add a new test

---

4.   See http://www.phpunit.de/. Admittedly, testing database functionality isn't strictly *unit
testing*, but you still can use this tool to organize and automate the tests.

that confirms that the object is present. You can also use *negative tests* to confirm that a table or column you removed in the current revision of your project is in fact no longer present.

Constraints: This is another use of negative testing. Try to execute INSERT, UPDATE, or DELETE statements that should result in an error because of a constraint. For example, try to violate not-null, unique constraints, or foreign keys. If the statement doesn't return an error, then your constraint isn't working. You can catch many bugs early by identifying these failures.

Triggers: Triggers can enforce constraints too. Triggers can perform cascading effects, transform values, log changes, and so on. You should test these scenarios by executing a statement that spawns the trigger and then querying to confirm that the trigger performed the action you intended.

Stored procedures: Testing procedures in the database is closest to conventional unit testing of application code. A stored procedure has input parameters, which could throw errors if you try to pass values outside the range of valid input. Logic within the body of the procedure could allow multiple execution paths. The procedure could return a single value or a query result set, depending on the inputs and the state of data in the database. Also, the procedure could have *side effects* in the form of updating the database. You can test all of these features of procedures.

Bootstrap data: Even a supposedly empty database typically needs some initial data, such as in lookup tables. You can run queries to validate that the initial data is present.

Queries: Application code is laced with SQL queries. You can execute queries in a test environment to validate syntax and results. Confirm that the result set includes the column names and data types you expect, just like testing tables and views.

ORM classes: Like triggers, ORM classes contain logic, including validation, transformation, or monitoring. You should test your ORM-based database abstraction code as you would any other application code. Confirm that these classes do the expected actions with input and also that they reject invalid input.

If any of your tests fail, your application could be using the wrong database instance. Always double-check that you're connecting to the right

database—the mistake is frequently simply a matter of connecting to the wrong instance. Edit the configuration if needed and try again. If you're sure you're connection is proper but you need to alter the database, then you can run a *migration script* (see the sidebar on page 273) to synchronize this database instance to match what your application expects.

Case Load: Working in Multiple Branches

While you develop your application, you could work on multiple revisions of the code. You might even work on different revisions in the same day. For example, you could fix an urgent bug in the branch of the application currently deployed and then moments later resume working on long-term development in the main branch.

But the database your application uses isn't under revision control. It's not practical to set up and tear down a database on a moment's notice, even if the database brand you use is relatively agile and easy to use.

Ideally, create a separate instance of your database for each revision of the application you need to develop, test, stage, or deploy. Also, each developer in your project team needs a separate database instance so they can work without interfering with the rest of the team.

Make your application support a configurable means to specify database connection parameters so that whichever application revision you work on, you can specify which database to use without overwriting code.

Today every RDBMS brand, both commercial and open source, offers a free solution for development and testing. Platform virtualization technology such as VMware Workstation, Xen, and VirtualBox allow every developer to run a clone of the server infrastructure at little cost. There is no reason that software developers can't develop and test in a fully functional environment that matches the production environment.

Use software development best practices, including documentation, testing, and source control, for your database as well as your application.

Explanations exist; they have existed for all time; there is always a well-known solution to every human problem—neat, plausible, and wrong.

▶ H. L. Mencken

Chapter 25

Magic Beans

"Why is it taking so long to add one little feature?" Your manager had assigned your team to enhance the bug-tracking application to show a count of how many comments a bug had received. You've been working on this task for four weeks.

Your group of software developers in the meeting room looks reluctant to answer the question. As project lead, you answer. "We've had a couple of false starts," you explain. "It seemed simple at first, until we realized there were some other screens in the application where we needed to show the comment count."

"And designing the screens took four weeks?" your manager asks.

"Well, no, that's just a little bit of HTML, and that's pretty easy since we use the framework that separates code from presentation," you go on. "But each time we added this element to a screen, we had to duplicate code to fetch data in the screen's back-end code. And that meant each back-end class needed a new suite of tests."

"Don't we use a testing framework?" your manager asks. "How long does it take to code a few more tests?"

"Writing tests isn't as simple as writing the code," another engineer says hesitantly. "We also created scripts for test data. Then we needed to reload data on the test database for each test. We needed to test the front end too, with all the permutations of the new feature combined with old scenarios."

Your manager's eyes are now starting to glaze over, but your co-worker continues, "We now have 600 tests for the front end, and each one runs an instance of a browser emulator. It just takes time to run through all those tests." He shrugs, "There's nothing we can do about that."

Your manager takes a deep breath, and says, "OK. . . I didn't follow all that; I just want to know why this is so complicated to add one simple feature. Wasn't your object-oriented framework supposed to make it quicker and easier to add features?"

Good question.

25.1 Objective: Simplify Models in MVC

Web application frameworks make it faster and easier to add features and code to an application. The greatest contributor to the cost of a software project is development time. So, the more we can reduce developer time, the less expensive it is to produce software.

Robert L. Glass found that "eighty percent of software work is intellectual. A fair amount of it is creative. Little of it is clerical."[1]

One way we assist the intellectual part of software development is to adopt the terminology and conventions of design patterns. When we say *Singleton* or *Facade* or *Factory*, the other developers on our team know what we mean. That saves a lot of time.

Much of the code in any application is repetitive—practically boiler-plate. Frameworks help improve coding productivity by giving us reusable components and code generation tools. We can produce working software applications while writing less original code.

Design patterns and software frameworks come together when we use the *Model View Controller* (MVC) architecture. This is a technique for separating concerns in an application, as illustrated in Figure 25.1, on the following page.

- The *controllers* accept user input, define the work the application should do in response, delegate work to the appropriate models, and send results to the view.
- The *models* handle everything else; they are the heart of the application and include input validation, business logic, and database interaction.
- The *views* present information in the user interface.

It's easy to understand what the controller and the view do. But the purpose of the model is more vague. There's a great desire in the software developer community to simplify and generalize what a model is,

---

1. *Facts and Fallacies of Software Engineering* [Gla92], p.60.
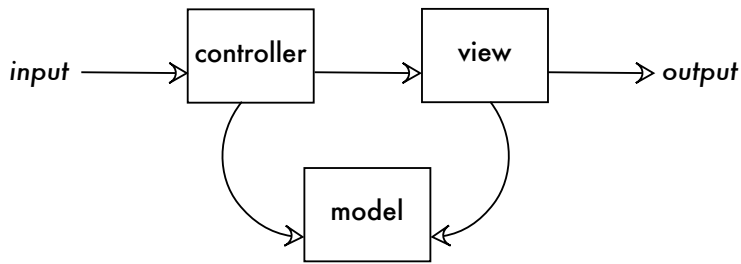
Figure 25.1: Model View Controller

with the goal of reducing the complexity of software design. But often that goal leads them to oversimplify by assuming the model is only a data access object.

25.2 Antipattern: The Model Is an Active Record

In simple applications, you don't need much custom logic in a model. It's relatively straightforward to match the fields of a model object to the columns of a single table in a database. This is a type of *object-relational mapping.* All you need the object to do is know how to create a row in the table, read the row, and update and delete it—the basic *CRUD* operations.

Martin Fowler described a design pattern to support this mapping, called *Active Record.*[2] Active Record is a data access pattern. You define a class corresponding to a table or view in your database. You can call a class method find() that returns an object instance of the class, corresponding to an individual row in that table or view. You can also use the class constructor to create a new row. Calling save() on this object either inserts a new row or updates the existing row.

Download Magic-Beans/anti/doctrine.php

```php
<?php
$bugsTable = Doctrine_Core::getTable('Bugs');
$bugsTable->find(1234);

$bug = new Bugs();
$bug->summary = "Crashes when I save";
$bug->save();
```

---

2. See Active Record in *Patterns of Enterprise Application Architecture* [Fow03], p.160.

Leaky Abstractions

Joel Spolsky coined the term *leaky abstractions* in 2002.* An abstraction simplifies the internal workings of some technology and makes it easier to use. But when you have to know the internals anyway to be productive, that's when it's called a leaky abstraction.

The use of the Active Record pattern as a model in MVC is a good example of a leaky abstraction. In very simple cases, Active Record works like magic. But if you try to use it for all database access, you find many operations such as JOIN or GROUP BY that are simple to express in SQL are awkward in Active Record.

Some frameworks try to enhance Active Record to support a variety of SQL clauses. The more these enhancements expose the fact that the class uses SQL internally, the more you feel like you might be better off using SQL directly.

The abstraction has failed to hide its secrets, like Toto exposing the Wizard of Oz as an ordinary man behind a curtain.

---

*. See *The Law of Leaky Abstractions* (Spo02).

Ruby on Rails popularized Active Record for web development frameworks in 2004, and now most web application frameworks use this pattern as the *de facto* data access object (DAO). There's nothing wrong with using Active Record; it's a fine pattern that provides a simple interface to individual rows in a single table. The antipattern is the convention that all model classes in an MVC application inherit from the base Active Record class. This is an example of the *Golden Hammer* antipattern: if the only tool you have is a hammer, treat everything as if it were a nail.

It's tempting to embrace any convention that simplifies software design. We can make our work easier if we're willing to sacrifice some flexibility, and if we never really needed the flexibility to begin with, that's even better.

But this is a fairy tale, like *Jack and the Beanstalk*. Jack believed that his magic beans would grow into a mighty beanstalk while he slept. It worked out all right in Jack's story, but we may not always be so lucky. Let's look at the consequences of the *Magic Beans* antipattern.

Active Record Couples Models to the Schema

Active Record is a simple pattern, because a plain Active Record class represents a single table or view in the database. The fields of each Active Record object match the columns in one corresponding table. If you have sixteen tables, you define sixteen model subclasses.

This means that if you need to refactor your database to represent a new structure of data, your model classes need to change, as well as any code in your application that uses the model classes. Likewise, if you add a controller to handle a new screen in your application, you may have to duplicate code that queries your models.

Active Record Exposes CRUD Functions

The next problem you may run into is that other programmers who use your model class can bypass your intended usage, updating data directly using CRUD functions.

For example, you might add a method assignUser() to a bug model, because you need to send an email to that engineer after updating the bug.

Download Magic-Beans/anti/crud.php

```php
<?php
class CustomBugs extends BaseBugs
{
  public function assignUser(Accounts $a)
  {
    $this->assigned_to = $a->account_id;
    $this->save();
    mail($a->email, "Assigned bug",
      "You are now responsible for bug #{$this->bug_id}.");
  }
}
```

However, another programmer working on the bug application bypasses your method and assigns the bug manually without sending the email.

Download Magic-Beans/anti/crud.php

```php
$bugsTable = Doctrine_Core::getTable('Bugs');
$bugsTable->find(1234);
$bug->assigned_to = $user->account_id;
$bug->save();
```

Your requirement was to have an email notification sent whenever the assignment changes. This allows that step to be bypassed. Does it make sense for your derived model class to expose the CRUD methods of the base Active Record class? How can you prevent other programmers

from using these methods inappropriately? How can you exclude the base Active Record interface from your model class's generated documentation and code completion in programming editors?

Active Record Encourages an Anemic Domain Model

A closely related point is that a model frequently has no behavior *except* generic CRUD methods. Many developers extend the base Active Record class without adding any new methods related to the work the model should do.

Treating models as simple data access objects encourages you to code your business logic outside the model, usually spread over multiple controller classes and reducing cohesion of the model's behavior. Martin Fowler calls this antipattern the *Anemic Domain Model* in his blog.[3] For example, you might have separate Active Record classes corresponding to the Bugs, Accounts, and Products tables. But you need data from all three of these tables in many application tasks.

Let's look at a simple code example for our bug-tracking application that implements bug assignment, data entry, bug display, and bug search tasks. It uses a PHP framework called Doctrine to provide a simple active record interface, and it uses the Zend Framework for the MVC architecture.

Download Magic-Beans/anti/anemic.php

```php
<?php
class AdminController extends Zend_Controller_Action
{
  public function assignAction()
  {
    $bugsTable = Doctrine_Core::getTable("Bugs");
    $bug = $bugsTable->find($_POST["bug_id"]);
    $bug->Products[] = $_POST["product_id"];
    $bug->assigned_to = $_POST["user_assigned_to"];
    $bug->save();
  }
}

class BugController extends Zend_Controller_Action
{
  public function enterAction()
  {
    $bug = new Bugs();
    $bug->summary = $_POST["summary"];
```

---

3. http://www.martinfowler.com/bliki/AnemicDomainModel.html

```php
      $bug->description = $_POST["summary"];
      $bug->status = "NEW";

      $accountsTable = Doctrine_Core::getTable("Accounts");
      $auth = Zend_Auth::getInstance();
      if ($auth && $auth->hasIdentity()) {
        $bug->reported_by = $auth->getIdentity();
      }
      $bug->save();
    }

    public function displayAction()
    {
      $bugsTable = Doctrine_Core::getTable("Bugs");
      $this->view->bug = $bugsTable->find($_GET["bug_id"]);

      $accountsTable = Doctrine_Core::getTable("Accounts");
      $this->view->reportedBy = $accountsTable->find($bug->reported_by);
      $this->view->assignedTo = $accountsTable->find($bug->assigned_to);
      $this->view->verifiedBy = $accountsTable->find($bug->verified_by);

      $productsTable = Doctrine_Core::getTable("Products");
      $this->view->products = $bug->Products;
    }
}

class SearchController extends Zend_Controller_Action
{
  public function bugsAction()
  {
    $q = Doctrine_Query::create()
      ->from("Bugs b")
      ->join("b.Products p")
      ->where("b.status = ?", $_GET["status"])
      ->andWhere("MATCH(b.summary, b.description) AGAINST (?)", $_GET["search"]);
    $this->view->searchResults = $q->fetchArray();
  }
}
```

Code that uses Active Record in controller classes expands to become a procedural approach to organizing application logic. If the database schema or the desired application behavior ever changes, you need to update many places in the code. Likewise, if you add a controller, you need to write new code even if your queries against the model are similar to those in other controllers.

The class interaction diagram (shown in Figure 25.2, on the next page) is messy and hard to read; it only gets worse as we add more controllers and DAO classes. This should be a strong clue that the code
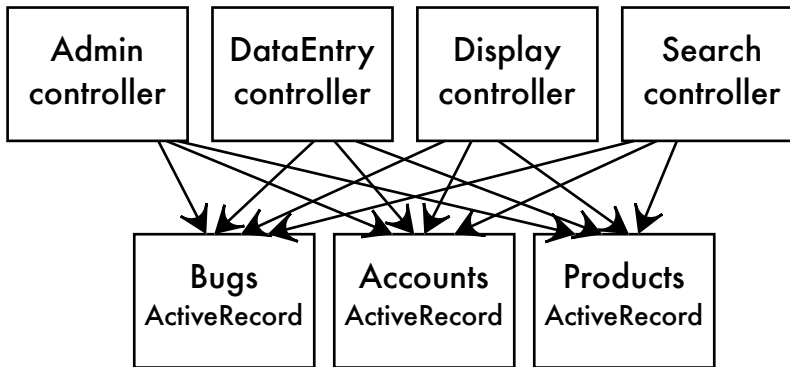
Figure 25.2: Using Magic Beans leads to vinelike tangles.

that uses different models together is duplicated across controllers. You need to use another approach to simplify and encapsulate part of your application.

Unit Testing Magic Beans Is Hard

When you employ the Magic Beans antipattern, you find that testing each of the layers in MVC is harder.

- *Testing the model:* Since you've made the model the same class as the Active Record, you can't test model behavior separately from data access. To test the model, you have to execute queries against a live database.

  Many people use *database fixtures*. A database fixture loads data into a test database to ensure that each test runs against a baseline state. Doing this much complex setup and teardown work makes testing models slow and error-prone, as well as requiring a live database for tests.

- *Testing the view:* Testing views involves rendering the view into HTML and parsing the result to verify that dynamic HTML elements provided by models appear in the output. Even if your framework simplifies the assertions in your test scripts, the framework has to run complex and time-consuming code to perform the rendering and then parse the HTML for specific elements.

- *Testing the controller:* You also find that testing the controller is complex, because a model that is a data access object leads to repetitions of the same code in multiple controllers, all of which need to be tested.

  To test a controller, you need to create a fake HTTP request. The output of a web application is an HTTP response header and body. To verify the test, you have to pick apart the HTTP response the controller returns. This needs a lot of setup code to test business logic, and it makes tests run slowly.

If you could separate business logic from the database access and separate business logic from presentation, it would help to meet the goals of MVC, and it would make testing simpler too.

25.3 How to Recognize the Antipattern

The following clues may mean that you have Magic Beans:

- "How can I pass a custom SQL query to a model?"

  The question suggests that you're using a database access class as a model class. You shouldn't have to pass SQL queries to the model—the model class should encapsulate any queries it needs.

- "Should I copy complex model queries to all my controllers, or should I code them once in an abstract controller?"

  Neither of these solutions gives you the stability or simplicity you are looking for. You should code complex queries within the model, exposed as part of the interface of the model. That way, you follow the *Don't Repeat Yourself* (DRY) principle, and you make the model's usage simpler.[4]

- "I have to write more database fixtures to unit test my models."

  If you're using database fixtures, you're testing database access, not business logic. You should be able to unit test a model in isolation from the database.

---

4. DRY was coined in *The Pragmatic Programmer* [HT00] by Andy Hunt and Dave Thomas.

25.4 Legitimate Uses of the Antipattern

There's nothing wrong with the Active Record design pattern per se. It's a convenient pattern for simple CRUD operations. In most applications, you have some cases where you need only a simple data access object for simple operations on individual rows of a table. You can simplify these cases by defining the model as coincident with its DAO.

Another good use of Active Record is for prototyping code. When writing code quickly is more important than writing code that's testable and maintainable, shortcuts are important. Showing a working prototype early and often is a great way to refine the project with active feedback. Anything you can do to speed up development of the prototype is helpful in these circumstances, and using simple application frameworks can help in this case.

Just be sure to budget some time for refactoring the code to pay back the technical debt you gather by writing coding in a prototype mode.

25.5 Solution: The Model Has an Active Record

Controllers handle application input and views handle application output, both relatively simple and well-defined tasks. Frameworks are best at helping you put these together quickly. But it's hard for a framework to provide a one-size-fits-all solution for models, because models comprise the rest of the object-oriented design for your application.

This is where you actually need to think hard about what the objects are in your application and what data and behavior those objects have. Remember Robert L. Glass's estimate that the majority of software development is intellectual and creative?

Grasping the Model

Fortunately, there's a lot of wisdom in the field of object-oriented design to guide you. Craig Larman's book *Applying UML and Patterns* [Lar04], for example, describes guidelines called the General Responsibility Assignment Software Patterns (GRASP). Some of these guidelines are especially relevant to separating models from their data access objects:

Information Expert

The object responsible for an operation should have *all* the data needed to fulfill that operation. Since some operations in your application involve multiple tables (or no tables) and Active Record is good at working

with only one table at a time, we need another class to aggregate several database access objects together and use them for the composite operation.

The relationship between a model and a DAO like Active Record should be *HAS-A* (aggregation) instead of *IS-A* (inheritance). Most frameworks that rely on Active Record assume the IS-A solution. If your model uses DAOs instead of inheriting from the DAO class, then you can design the model to contain all data and code for the domain it's supposed to model—even if it takes multiple database tables to represent it.

Creator

How the model persists its data in a database should be an internal implementation detail. A domain model that aggregates its DAOs should have the responsibility to create those objects.

The controllers and views in your application should use the domain model interface, without being aware of what kind of database interaction is necessary for the model to fetch or store data. This makes it easy to change the database queries later, in one place in your application.

Low Coupling

It's important to decouple logically independent blocks of code. This gives you the flexibility to change the implementation of a class without affecting its consumers. You can't simplify the requirements of the application; some complexity has to reside somewhere in your code. But you can make the best choice about where you implement that complexity.

High Cohesion

The interface for the domain model class should reflect its intended usage, not the physical database structure or CRUD operations. Generic methods of the Active Record interface like find(), first(), insert(), or even save() don't tell you much about how they apply to application requirements. Methods like assignUser() are more descriptive, and your controller code is easier to understand.

When you decouple a model class from the DAO it uses, you can even design more than one model class for the same DAO. This is better for cohesion than trying to combine all work related to the given tables into a single class extending Active Record.

Putting the Domain Model into Action

In *Domain-Driven Design: Tackling Complexity in the Heart of Software* [Eva03], Eric Evans describes a better solution: the *domain model.*

A model in the original MVC sense—not in the opinionated software sense—is an object-oriented representation of a *domain* in your application, that is, the business rules in your application and the data for those business rules. The model is where you implement business logic for the application; storing it in a database is an internal implementation detail of a model.

Once we have the model designed around concepts in our application, instead of database layout, we can start to implement database operations completely hidden within our model classes. Let's look at a possible refactoring of our earlier example code:

Download   Magic-Beans/soln/domainmodel.php

```php
<?php

class BugReport
{
  protected $bugsTable;
  protected $accountsTable;
  protected $productsTable;

  public function __construct()
  {
    $this->bugsTable = Doctrine_Core::getTable("Bugs");
    $this->accountsTable = Doctrine_Core::getTable("Accounts");
    $this->productsTable = Doctrine_Core::getTable("Products");
  }

  public function create($summary, $description, $reportedBy)
  {
    $bug = new Bugs();
    $bug->summary = $summary;
    $bug->description = $description;
    $bug->status = "NEW";
    $bug->reported_by = $reportedBy;
    $bug->save();
  }

  public function assignUser($bugId, $assignedTo)
  {
    $bug = $bugsTable->find($bugId);
    $bug->assigned_to = $assignedTo"];
    $bug->save();
  }
```

```php
  public function get($bugId)
  {
    return $bugsTable->find($bugId);
  }

  public function search($status, $searchString)
  {
    $q = Doctrine_Query::create()
      ->from("Bugs b")
      ->join("b.Products p")
      ->where("b.status = ?", $status)
      ->andWhere("MATCH(b.summary, b.description) AGAINST (?)", $searchString]);
    return $q->fetchArray();
  }
}


class AdminController extends Zend_Controller_Action
{
  public function assignAction()
  {
    $this->bugReport->assignUser(
      $this->_getParam("bug"),
      $this->_getParam("user"));
  }
}


class BugController extends Zend_Controller_Action
{
  public function enterAction()
  {
    $auth = Zend_Auth::getInstance();
    if ($auth && $auth->hasIdentity()) {
      $identity = $auth->getIdentity();
    }
    $this->bugReport->create(
      $this->_getParam("summary"),
      $this->_getParam("description"),
      $identity);
  }


  public function displayAction()
  {
    $this->view->bug = $this->bugReport->get(
      $this->_getParam("bug"));
  }
}
```

```
class SearchController extends Zend_Controller_Action
{
  public function bugsAction()
  {
    $this->view->searchResults = $this->bugReport->search(
      $this->_getParam("status", "OPEN"),
      $this->_getParam("search"));
  }
}
```

You should be able to notice several improvements:

- The class interaction diagram (shown in Figure 25.3, on the following page) is much simpler and easier to read, indicating an improvement in decoupling classes.

- By decoupling the model's interface from its underlying database structure, we've reduced and simplified the code in the controller.

- Each model class creates the objects to interact with one or more tables. The controllers do not need to know which tables are involved.

- The model classes encapsulate and hide the database queries. The controller is concerned only with retrieving user inputs and invoking higher-level tasks through the model API.

- In some cases, a query is too complex to do easily through a DAO, and writing custom SQL is needed. Using plain SQL seems less scary when it's safely encapsulated inside a model class.

Testing Plain Objects

Ideally, you should be able to test your model without connecting to a live database. If you decouple your model from its DAO, then you can create *stub* and *mock* DAOs to help unit test your model.

Likewise, you can test the interface of a domain model just like any other object-oriented testing: call methods of the object, and then validate the method's return value. This is faster and easier than creating fake HTTP requests to feed to a controller and parsing the resulting HTTP response.

You still test your controllers with fake HTTP requests, but because the controller code is simpler, you don't need to test as many logical paths.

If you separate models and controllers and separate data access components from models, then you can unit test all these classes more simply
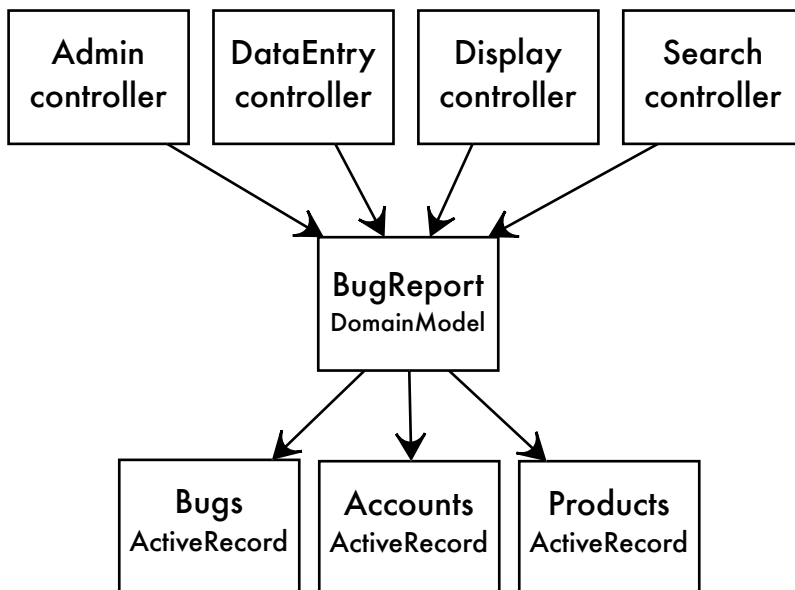
Figure 25.3: Untangling the vines by decoupling

and with better isolation. This makes it easier to diagnose defects when they occur. Isn't this the point of unit tests?

Getting Down to Earth

You can use a data access object productively in any software development framework, even one that encourages the Magic Beans antipattern. However, developers who don't learn how to employ object-oriented design principles are doomed to write spaghetti code.

The basics of domain modeling described and cited in this chapter will help you choose the best design to support testing and code maintenance. You'll finally be able to achieve great productivity developing database-driven applications.

Decouple your models from your tables.

Part V

Appendixes

Young man, in mathematics you don't understand things.
You just get used to them.
▶ John von Neumann

Appendix A

Rules of Normalization

Relational database design isn't arbitrary or mysterious. You can use a number of well-defined rules to design a data storage strategy that avoids redundancy and helps make your application mistake-proof, like the poka-yoke ideas mentioned earlier in this book. You've probably heard other metaphors for the same idea, such as *defensive design* or *fail early*.

The rules of normalization aren't complicated, but they are subtle. Developers often misunderstand how they work, perhaps because they expect the rules to be harder than they are.

Another possibility is that people are turned off by having to follow rules at all. Rules are the bête noire of developers who value newness, creativity, and innovation. Rules are in a way the opposite of freedom.

Software developers continually make trade-offs between simplicity and flexibility. You can make a lot of work for yourself by reinventing the wheel and developing custom data management software for every application. Or you can take advantage of existing knowledge and technology if you can conform to a relational design.

I've described the antipatterns in this book using their own merits (or faults) to avoid being too academic or theoretical. But in this appendix, we'll see that theory can also be practical.

A.1 What Does Relational Mean?

This term *relational* doesn't refer to relationships between tables. It refers to the table itself, or rather, the relationship between columns within a table. In a way, it refers to both.

Mathematicians define a *relation* as the combination of two sets of values from different domains, with some condition applied that gives us a subset of all the possible combinations.

For example, one set is the names of baseball teams, and the other set is cities. The combination of every team to every city is a long list of pairings. But we're interested in a particular subset of this list: the teams paired with their home city. Valid pairs include *Chicago/White Sox*, *Chicago/Cubs*, or *Boston/Red Sox*, but not *Miami/Red Sox*.

The word *relation* is used in two ways: as a rule ("this city is the home city of that team") and as the subset of pairings that comply with the rule. In SQL, we can store that result in a table with two columns, and one row per pair.

Of course, relations support more than two columns. You can combine any number of domains, one per column, into a relation. Also, you can use domains like the set of 32-bit integers or the set of text strings of a specific length.

Before we can begin normalizing tables, we need to be sure that they are proper relations. They have to meet a few criteria.

Rows Have No Order from Top to Bottom

In SQL, a query returns results in an unpredictable order, unless you use an ORDER BY clause to specify the order. But apart from the order, the set of rows is the same.

Columns Have No Order from Left to Right

Whether we ask Steven to test the product Open RoundFile against bug #1234 or whether we need to know if bug #1234 can be verified in product Open RoundFile by Steven, the result should be the same.

This is related to the antipattern in Chapter 19, *Implicit Columns*, on page 214, where we would use columns by their position instead of by their name.

Duplicate Rows Are Not Allowed

Once you know a fact, stating it again doesn't make it any more true. Given the name of a baseball team, your data dictates the city. We say the city *depends on* the team name.

To prevent duplicates, we have to be able to tell one row from another and to address individual rows. To ensure this in SQL, we declare a

primary key constraint for a column or set of columns, whatever is needed to uniquely identify rows.

We might have duplication among nonkey columns—there are two teams in the city of Boston—but the row as a whole is still unique because the team names are different.

Every Column Has One Type, and One Value per Row

A relation has a *header* that defines the names and data types of the columns. Every row must have the same columns as those in the header, and a given column must have the same meaning on all rows.

We saw an antipattern break this rule in two ways in Chapter 6, *Entity-Attribute-Value*, on page 73. First, the EAV table models an entity that can have a custom set of attributes for every instance, so the entity is not bound by any header that defines its attributes.

Second, the EAV attr_value column contains all the entity's attributes, such as the bug's date reported, the bug's status, the account the bug is assigned to, and so on. A given value like 1234 in this column may be valid for two different attributes but mean something totally different.

The antipattern in Chapter 7, *Polymorphic Associations*, on page 89 also breaks this rule, because a given value like 1234 references the primary key of any of the multiple parent tables. You can't say 1234 on one row means the same thing as 1234 on another row.

Rows Have No Hidden Components

Columns contain data values, not physical storage indicators such as row IDs or object IDs. Above in Chapter 22, *Pseudokey Neat-Freak*, on page 250, we saw that primary keys are unique, but they aren't row numbers.

Some databases bend this rule, giving you access to internal storage details with extensions to SQL (for example, the ROWNUM pseudocolumn in Oracle or OID in PostgreSQL). However, these values aren't properly part of the relation.

A.2   Myths About Normalization

It's hard to find a subject that is so widely misunderstood, despite having a precise definition. You are practically guaranteed to encounter

developers who express with complete confidence untruths such as these:

- "Normalization makes a database slower. Denormalization makes a database faster."

  False. It's true that you may need to use a join to retrieve attributes from separate tables after you apply normalization. If you denormalize data, you can avoid some joins.

  For example, the comma-separated list in Chapter 2, *Jaywalking*, on page 25 stores products for a given bug. But what if you also need a list of bugs for a given product? Denormalization usually helps convenience or performance for one type of query, but at a great cost for some other types of queries.

  There are legitimate uses for denormalization. But you should model your database in a normal form first, before deciding to use denormaliziation. The *MENTOR* guide for indexing in Chapter 13, *Index Shotgun*, on page 148 applies to denormalization too: be sure you measure performance both before and after you implement a change for the sake of efficiency.

- "Normalization says to push the data out to child tables and reference it using a pseudokey."

  False. You can use pseudokeys for the goal of convenience, performance, or storage efficiency, and those reasons are legitimate. But don't believe that it has anything to do with normalization.

- "Normalization is where you separate attributes as much as possible, such as in the Entity-Attribute-Value design."

  False. It's common for developers to use the word *normalization* inaccurately, implying that it makes data less human-readable or less convenient to query. In fact, the opposite is true.

- "No one needs to normalize past the third normal form. The other normal forms are so esoteric that you'll never encounter them."

  False. One study showed that more than 20 percent of business databases contain designs that satisfy the first three normal forms but violate the fourth normal form. This is a minority, but it's far from insignificant. If you learn of a bug that potentially results in data loss and occurs in 20 percent of your applications, wouldn't you want to fix it?
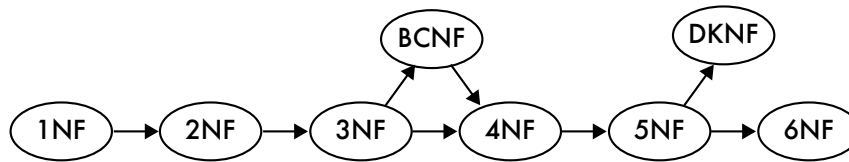
Figure A.1: Progression of normal forms

A.3 What Is Normalization?

The following are the objectives of normalization:

- To represent facts about the real world in a way that we can understand

- To reduce storing facts redundantly and to prevent anomalous or inconsistent data

- To support integrity constraints

Notice that improving database performance is not on this list. Normalization helps us store data *correctly* and avoid getting into trouble. It's practically inevitable that a database that is not normalized becomes a mess. We find ourselves developing a lot more code to clean up inconsistent or duplicate data. We experience delays and expenses to our businesses from faulty data. If you include these scenarios, the benefits to performance from normalizing a database become clearer.

When a table satisfies rules of normalization, we say the table is in *normal form*. There are five traditional normal forms, describing progressive levels of normalization. Each normal form eliminates a specific type of redundancy or anomaly when you design a relation. Generally, if your table satisfies a normal form, the table also satisfies all the preceding normal forms. There are three additional normal forms that researchers have described. The progression of normal forms is shown in Figure A.1.

First Normal Form

The most fundamental requirement for first normal form is that the table must be a relation. If it doesn't meet the criteria for a relation described in the first section, then your table can't be in first normal form or any of the subsequent normal forms.
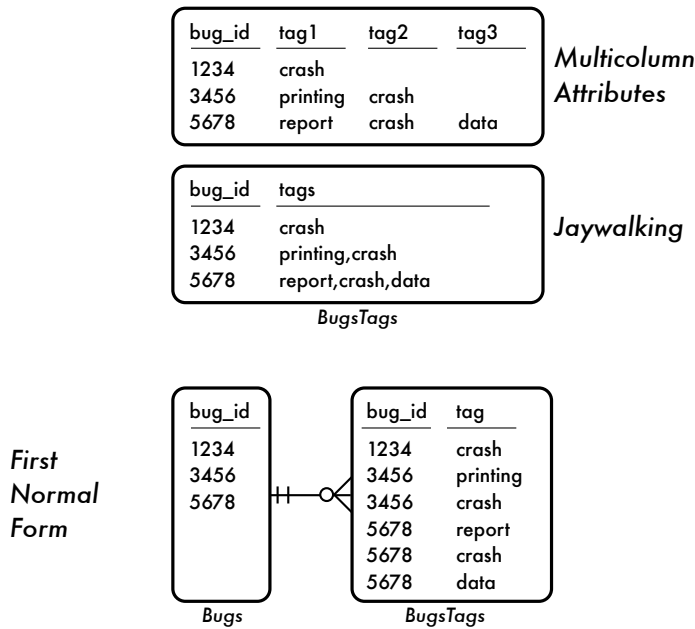
Figure A.2: Repeating groups vs. first normal form

The next requirement is that the table must not have any *repeating groups*. Remember that each row in a relation is a combination between several sets, choosing one value from each set. A repeating group means that one row may have multiple values from the given set.

We saw two antipatterns that create repeating groups:

- Multiple values from the same domain across multiple columns, in Chapter 8, *Multicolumn Attributes*, on page 102

- Multiple values within a single column, in Chapter 2, *Jaywalking*, on page 25

In Figure A.2, we can see repeating groups according to each of these antipatterns. The proper design that satisfies first normal form is to create a separate table. Tags now occupy a single column, and we can support multiple tags by storing one tag per row.

Second Normal Form

The second normal form is identical to the first normal form, unless your table has a compound primary key. In the tagging example, let's keep track of which user chose to apply each given tag to a bug. We're also interested in who first coined a given tag.

Download Normalization/2NF-anti.sql

```
CREATE TABLE BugsTags (
  bug_id  BIGINT NOT NULL,
  tag     VARCHAR(20) NOT NULL,
  tagger  BIGINT NOT NULL,
  coiner  BIGINT NOT NULL,
  PRIMARY KEY (bug_id, tag),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (tagger) REFERENCES Accounts(account_id),
  FOREIGN KEY (coiner) REFERENCES Accounts(account_id)
);
```

In Figure A.3, on the next page, we can see that the identity of the coiner is stored redundantly.[1] This means someone might create an *anomaly* by changing the identity of the coiner on one row for a given tag (*crash*) without changing all rows for the same tag.

To satisfy second normal form, we should store the coiner for a given tag only once. That means we have to define another table, Tags, where the tag is the primary key, so there's bound to be only one row per distinct tag. Then we can store the coiner of that tag in this new table instead of in BugsTags and prevent anomalies.

Download Normalization/2NF-normal.sql

```
CREATE TABLE Tags (
  tag     VARCHAR(20) PRIMARY KEY,
  coiner  BIGINT NOT NULL,
  FOREIGN KEY (coiner) REFERENCES Accounts(account_id)
);

CREATE TABLE BugsTags (
  bug_id  BIGINT NOT NULL,
  tag     VARCHAR(20) NOT NULL,
  tagger  BIGINT NOT NULL,
  PRIMARY KEY (bug_id, tag),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (tag) REFERENCES Tags(tag),
  FOREIGN KEY (tagger) REFERENCES Accounts(account_id)
);
```

---

1. The figure uses names instead of ID numbers for the user identities.
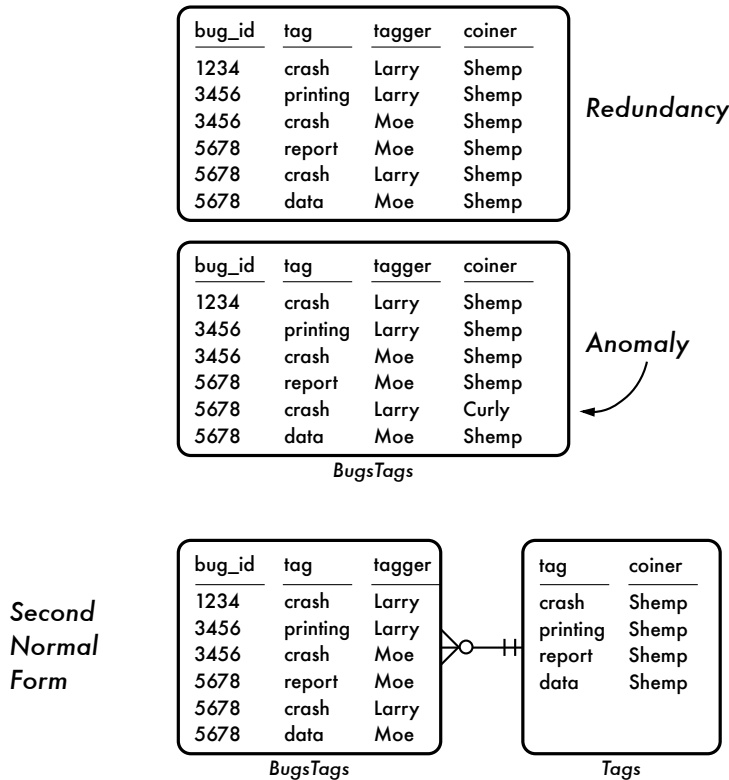
Figure A.3: Redundancy vs. second normal form

Third Normal Form

In the Bugs table, you might want to store the email of the engineer working on the bug.

Download Normalization/3NF-anti.sql

```
CREATE TABLE Bugs (
  bug_id SERIAL PRIMARY KEY
  -- . . .
  assigned_to BIGINT,
  assigned_email VARCHAR(100),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id)
);
```

However, the email is an attribute of the assigned engineer's account; it's not strictly an attribute of the bug. It's redundant to store the email
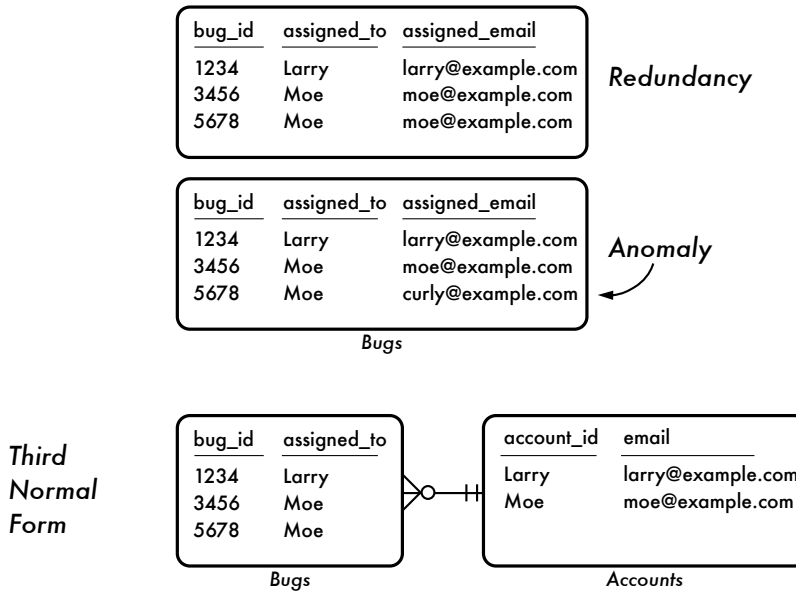
Figure A.4: Redundancy vs. third normal form

in this way, and we risk anomalies like in the table that fails second normal form.

In the example for second normal form the offending column is related to at least *part* of the compound primary key. In this example, that violates third normal form, the offending column doesn't correspond to the primary key at all.

To fix this, we need to put the email address into the Accounts table. See how you can separate the column from the Bugs table in Figure A.4. That's the right place because the email corresponds directly to the primary key of that table, without redundancy.

Boyce-Codd Normal Form

A slightly stronger version of third normal form is called Boyce-Codd normal form. The difference between these two normal forms is that in third normal form, all nonkey attributes must depend on the key of the table. In Boyce-Codd normal form, key columns are subject to this rule as well. This would come up only when the table has multiple sets of columns that *could* serve as the table's key.
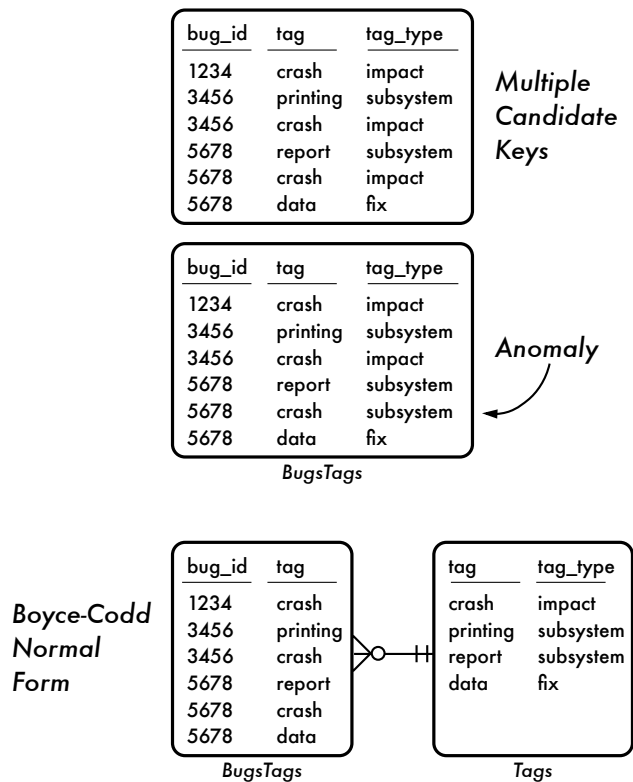
Figure A.5: Third normal form vs. Boyce-Codd normal form

For example, suppose we have three tag types: tags that describe the impact of the bug, tags for the subsystem the bug affects, and tags that describe the fix for the bug. We decide that each bug must have at most one tag of each type. Our candidate key could be bug_id plus tag, but it could also be bug_id plus tag_type. Either pair of columns would be specific enough to address every row individually.

In Figure A.5, we see an example of a table that is in third normal form, but not Boyce-Codd normal form, and how to change it.

Fourth Normal Form

Now let's alter our database to allow each bug to be reported by multiple users, assigned to multiple development engineers, and verified by

multiple quality engineers. We know that a many-to-many relationship deserves an additional table:

Download **Normalization/4NF-anti.sql**

```sql
CREATE TABLE BugsAccounts (
  bug_id       BIGINT NOT NULL,
  reported_by  BIGINT,
  assigned_to  BIGINT,
  verified_by  BIGINT,
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id),
  FOREIGN KEY (verified_by) REFERENCES Accounts(account_id)
);
```

We can't use bug_id alone as the primary key. We need multiple rows per bug so we can support multiple accounts in each column. We also can't declare a primary key over the first two or the first three columns, because that would still fail to support multiple values in the last column. So, the primary key would need to be over all four columns. However, assigned_to and verified_by should be nullable, because bugs can be reported before being assigned or verified, All primary key columns standardly have a NOT NULL constraint.

Another problem is that we may have redundant values when any column contains fewer accounts than some other column. The redundant values are shown in Figure A.6, on the following page.

All the problems shown previously are caused by trying to create an intersection table that does double-duty—or triple-duty in this case. When you try to use a single intersection table to represent multiple many-to-many relationships, it violates fourth normal form.

The figure shows how we can solve this by splitting the table so that we have one intersection table for each type of many-to-many relationship. This solves the problems of redundancy and mismatched numbers of values in each column.

Download **Normalization/4NF-normal.sql**

```sql
CREATE TABLE BugsReported (
  bug_id       BIGINT NOT NULL,
  reported_by  BIGINT NOT NULL,
  PRIMARY KEY (bug_id, reported_by),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id)
);
```

bug_id	reported_by	assigned_to	verified_by
1234	Zeppo	NULL	NULL
3456	Chico	Groucho	Harpo
3456	Chico	Spalding	Harpo
5678	Chico	Groucho	NULL
5678	Zeppo	Groucho	NULL
5678	Gummo	Groucho	NULL

BugsAccounts

Redundancy,
NULLs,
No Primary Key

Fourth
Normal
Form

bug_id	reported_by
1234	Zeppo
3456	Chico
5678	Chico
5678	Zeppo
5678	Gummo

BugsReported

bug_id	assigned_to
3456	Groucho
3456	Spalding
5678	Groucho

BugsAssigned

bug_id	verified_by
3456	Harpo

BugsVerified

Figure A.6: Merged relationships vs. fourth normal form

```
CREATE TABLE BugsAssigned (
  bug_id       BIGINT NOT NULL,
  assigned_to  BIGINT NOT NULL,
  PRIMARY KEY (bug_id, assigned_to),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id)
);

CREATE TABLE BugsVerified (
  bug_id       BIGINT NOT NULL,
  verified_by  BIGINT NOT NULL,
  PRIMARY KEY (bug_id, verified_by),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (verified_by) REFERENCES Accounts(account_id)
);
```

Fifth Normal Form

Any table that meets the criteria of Boyce-Codd normal form and does not have a compound primary key is already in fifth normal form. But to understand fifth normal form, let's work through an example.

Some engineers work only on certain products. We should design our database so that we know the facts of who works on which products and

bug_id	assigned_to	product_id
3456	Groucho	Open RoundFile
3456	Spalding	Open RoundFile
5678	Groucho	Open RoundFile

BugsAssigned

Redundancy, Multiple Facts

Fifth Normal Form

bug_id	assigned_to
3456	Groucho
3456	Spalding
5678	Groucho

BugsAssigned

account_id	product_id
Groucho	Open RoundFile
Groucho	ReConsider
Spalding	Open RoundFile
Spalding	Visual Turbo Builder

EngineerProducts

Figure A.7: Merged relationships vs. fifth normal form

which bugs, with a minimum of redundancy. Our first try at supporting this is to add a column to our BugsAssigned table to show that a given engineer works on a product:

Download Normalization/5NF-anti.sql

```sql
CREATE TABLE BugsAssigned (
  bug_id        BIGINT NOT NULL,
  assigned_to   BIGINT NOT NULL,
  product_id    BIGINT NOT NULL,
  PRIMARY KEY (bug_id, assigned_to),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

This doesn't tell us which products we may assign the engineer to work on; it only tells us which products the engineer is currently assigned to work on. It also stores the fact that an engineer works on a given product redundantly. This is caused by trying to store multiple facts about independent many-to-many relationships in a single table, similar to the problem we saw in the fourth normal form. The redundancy is illustrated in Figure A.7.[2]

---

2. The figure uses names instead of ID numbers for the products.

Our solution is to isolate each relationship into separate tables:

```sql
CREATE TABLE BugsAssigned (
  bug_id       BIGINT NOT NULL,
  assigned_to  BIGINT NOT NULL,
  PRIMARY KEY (bug_id, assigned_to),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);

CREATE TABLE EngineerProducts (
  account_id   BIGINT NOT NULL,
  product_id   BIGINT NOT NULL,
  PRIMARY KEY (account_id, product_id),
  FOREIGN KEY (account_id) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

Now we can record the fact that an engineer is available to work on a given product, independently from the fact that the engineer is working on a given bug for that product.

Further Normal Forms

Domain-Key normal form (DKNF) says that every constraint on a table is a logical consequence of the table's domain constraints and key constraints. Normal forms three, four, five, and Boyce-Codd normal form are all encompassed by DKNF.

For example, you may decide that a bug that has a status of *NEW* or *DUPLICATE* has resulted in no work, so there should be no hours logged, and also it makes no sense to assign a quality engineer in the verified_by column. You might implement these constraints with a trigger or a CHECK constraint. These are constraints between nonkey columns of the table, so they don't meet the criteria of DKNF.

Sixth normal form seeks to eliminate all join dependencies. It's typically used to support a history of changes to attributes. For example, the Bugs.status changes over time, and we might want to record this history in a child table, as well as when the change occurred, who made the change, and perhaps other details.

You can imagine that for Bugs to support sixth normal form fully, nearly every column may need a separate accompanying history table. This

leads to an overabundance of tables. Sixth normal form is overkill for most applications, but some data warehousing techniques use it.[3]

A.4 Common Sense

Rules of normalization aren't esoteric or complicated. They're really just a commonsense technique to reduce redundancy and improve consistency of data.

You can use this brief overview of relations and normal forms as an quick reference to help you design better databases in future projects.

---

3. For example, Anchor Modeling uses it (http://www.anchormodeling.com/).

Bibliography

[BMMM98] William J. Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. *AntiPatterns*. John Wiley and Sons, Inc., New York, 1998.

[Cel04]   Joe Celko. *Joe Celko's Trees and Hierarchies in SQL for Smarties*. Morgan Kaufmann Publishers, San Francisco, 2004.

[Cel05]   Joe Celko. *Joe Celko's SQL Programming Style*. Morgan Kaufmann Publishers, San Francisco, 2005.

[Cod70]   Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[Eva03]   Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, Reading, MA, first edition, 2003.

[Fow03]   Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley Longman, Reading, MA, 2003.

[Gla92]   Robert L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley Professional, Reading, MA, 1992.

[Gol91]   David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, pages 5–48, March 1991. Reprinted http://www.validlab.com/goldberg/paper.pdf.

[GP03]   Peter Gulutzan and Trudy Pelzer. *SQL Performance Tuning.* Addison-Wesley, 2003.

[HLV05]   Michael Howard, David LeBlanc, and John Viega. *19 Deadly Sins of Software Security.* McGraw-Hill, Emeryville, California, 2005.

[HT00]   Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master.* Addison-Wesley, Reading, MA, 2000.

[Lar04]   Craig Larman. *Applying UML and Patterns: an Introduction to Object-Oriented Analysis and Design and Iterative Development.* Prentice Hall, Englewood Cliffs, NJ, third edition, 2004.

[RTH08]   Sam Ruby, David Thomas, and David Heinemeier Hansson. *Agile Web Development with Rails.* The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, third edition, 2008.

[Spo02]   Joel Spolsky. The law of leaky abstractions. http://www.joelonsoftware.com/articles/LeakyAbstractions.html, 2002.

[SZT+08]   Baron Schwartz, Peter Zaitsev, Vadim Tkachenko, Jeremy Zawodny, Arjen Lentz, and Derek J. Balling. *High Performance MySQL.* O'Reilly Media, Inc., second edition, 2008.

[Tro06]   Vadim Tropashko. *SQL Design Patterns.* Rampant Techpress, Kittrell, NC, USA, 2006.

Index

T

The Pragmatic Bookshelf

Available in paperback and DRM-free eBooks, our titles are here to help you stay on top of your game. The following are in print as of June 2010; be sure to check our website at pragprog.com *for newer titles.*

Title	Year	ISBN	Pages
Advanced Rails Recipes: 84 New Ways to Build Stunning Rails Apps	2008	9780978739225	464
Agile Coaching	2009	9781934356432	248
Agile Retrospectives: Making Good Teams Great	2006	9780977616640	200
Agile Web Development with Rails, Third Edition	2009	9781934356166	784
Beginning Mac Programming: Develop with Objective-C and Cocoa	2010	9781934356517	300
Behind Closed Doors: Secrets of Great Management	2005	9780976694021	192
Best of Ruby Quiz	2006	9780976694076	304
Cocoa Programming: A Quick-Start Guide for Developers	2010	9781934356302	450
Core Animation for Mac OS X and the iPhone: Creating Compelling Dynamic User Interfaces	2008	9781934356104	200
Core Data: Apple's API for Persisting Data on Mac OS X	2009	9781934356326	256
Data Crunching: Solve Everyday Problems using Java, Python, and More	2005	9780974514079	208
Debug It! Find, Repair, and Prevent Bugs in Your Code	2009	9781934356289	232
Deploying Rails Applications: A Step-by-Step Guide	2008	9780978739201	280
Design Accessible Web Sites: 36 Keys to Creating Content for All Audiences and Platforms	2007	9781934356029	336
Desktop GIS: Mapping the Planet with Open Source Tools	2008	9781934356067	368
Developing Facebook Platform Applications with Rails	2008	9781934356128	200
Domain-Driven Design Using Naked Objects	2009	9781934356449	375
Enterprise Integration with Ruby	2006	9780976694069	360
Enterprise Recipes with Ruby and Rails	2008	9781934356234	416
Everyday Scripting with Ruby: for Teams, Testers, and You	2007	9780977616619	320
ExpressionEngine 2: A Quick-Start Guide	2010	9781934356524	250
FXRuby: Create Lean and Mean GUIs with Ruby	2008	9781934356074	240
From Java To Ruby: Things Every Manager Should Know	2006	9780976694090	160

Title	Year	ISBN	Pages
GIS for Web Developers: Adding Where to Your Web Applications	2007	9780974514093	275
Google Maps API, V2: Adding Where to Your Applications	2006	PDF-Only	83
Grails: A Quick-Start Guide	2009	9781934356463	200
Groovy Recipes: Greasing the Wheels of Java	2008	9780978739294	264
Interface Oriented Design	2006	9780976694052	240
Land the Tech Job You Love	2009	9781934356265	280
Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages	2009	9781934356456	350
Learn to Program, 2nd Edition	2009	9781934356364	240
Manage It! Your Guide to Modern Pragmatic Project Management	2007	9780978739249	360
Manage Your Project Portfolio: Increase Your Capacity and Finish More Projects	2009	9781934356296	200
Mastering Dojo: JavaScript and Ajax Tools for Great Web Experiences	2008	9781934356111	568
Metaprogramming Ruby: Program Like the Ruby Pros	2010	9781934356470	240
Modular Java: Creating Flexible Applications with OSGi and Spring	2009	9781934356401	260
No Fluff Just Stuff 2006 Anthology	2006	9780977616664	240
No Fluff Just Stuff 2007 Anthology	2007	9780978739287	320
Pomodoro Technique Illustrated: The Easy Way to Do More in Less Time	2009	9781934356500	144
Practical Programming: An Introduction to Computer Science Using Python	2009	9781934356272	350
Practices of an Agile Developer	2006	9780974514086	208
Pragmatic Ajax: A Web 2.0 Primer	2006	9780976694083	296
Pragmatic Project Automation: How to Build, Deploy, and Monitor Java Applications	2004	9780974514031	176
Pragmatic Thinking and Learning: Refactor Your Wetware	2008	9781934356050	288
Pragmatic Unit Testing in C# with NUnit	2007	9780977616671	176
Pragmatic Unit Testing in Java with JUnit	2003	9780974514017	160
Pragmatic Version Control Using Git	2008	9781934356159	200
Pragmatic Version Control using CVS	2003	9780974514000	176
Pragmatic Version Control using Subversion	2006	9780977616657	248
Programming Clojure	2009	9781934356333	304
Programming Cocoa with Ruby: Create Compelling Mac Apps Using RubyCocoa	2009	9781934356197	300
Programming Erlang: Software for a Concurrent World	2007	9781934356005	536

Continued on next page

Title	Year	ISBN	Pages
Programming Groovy: Dynamic Productivity for the Java Developer	2008	9781934356098	320
Programming Ruby: The Pragmatic Programmers' Guide, Second Edition	2004	9780974514055	864
Programming Ruby 1.9: The Pragmatic Programmers' Guide	2009	9781934356081	960
Programming Scala: Tackle Multi-Core Complexity on the Java Virtual Machine	2009	9781934356319	250
Prototype and script.aculo.us: You Never Knew JavaScript Could Do This!	2007	9781934356012	448
Rails Recipes	2006	9780977616602	350
Rails for .NET Developers	2008	9781934356203	300
Rails for Java Developers	2007	9780977616695	336
Rails for PHP Developers	2008	9781934356043	432
Rapid GUI Development with QtRuby	2005	PDF-Only	83
Release It! Design and Deploy Production-Ready Software	2007	9780978739218	368
Scripted GUI Testing with Ruby	2008	9781934356180	192
Ship It! A Practical Guide to Successful Software Projects	2005	9780974514048	224
Stripes ...and Java Web Development Is Fun Again	2008	9781934356210	375
TextMate: Power Editing for the Mac	2007	9780978739232	208
The Definitive ANTLR Reference: Building Domain-Specific Languages	2007	9780978739256	384
The Passionate Programmer: Creating a Remarkable Career in Software Development	2009	9781934356340	200
ThoughtWorks Anthology	2008	9781934356142	240
Ubuntu Kung Fu: Tips, Tricks, Hints, and Hacks	2008	9781934356227	400
Web Design for Developers: A Programmer's Guide to Design Tools and Techniques	2009	9781934356135	300
iPhone SDK Development	2009	9781934356258	576

How to Get It Done

Driving Technical Change

Your co-workers' resistance to new technologies can be baffling. Learn to read users' "patterns of resistance"—and then dismantle their objections. Every developer must master the art of evangelizing. With these techniques and strategies, you'll help your organization adopt your solutions—without selling your soul to organizational politics.

Driving Technical Change: Why People On Your Team Don't Act On Good Ideas, and How to Convince Them They Should
Terrence Ryan
(200 pages) ISBN: 978-1934356-60-9. $32.95
http://pragprog.com/titles/trevan

The Agile Samurai

Faced with a software project of epic proportions? Tired of over-committing and under-delivering? Enter the dojo of the agile samurai, where agile expert Jonathan Rasmusson shows you how to kick-start, execute, and deliver your agile projects. You'll see how agile software delivery really works and how to help your team get agile fast, while having fun along the way.

The Agile Samurai: How Agile Masters Deliver Great Software
Jonathan Rasmusson
(275 pages) ISBN: 9781934356586. $34.95
http://pragprog.com/titles/jtrap
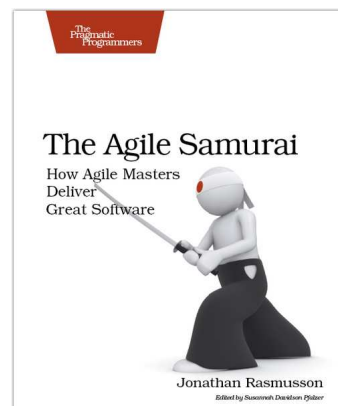
# Expand Your Boundaries

Seven Languages in Seven Weeks

In this book you'll get a hands-on tour of Clojure, Haskell, Io, Prolog, Scala, Erlang, and Ruby. Whether or not your favorite language is on that list, you'll broaden your perspective of programming by examining these languages side-by-side. You'll learn something new from each, and best of all, you'll learn how to learn a language quickly.

Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages
Bruce A. Tate
(300 pages) ISBN: 978-1934356-59-3. $34.95
http://pragprog.com/titles/btlang
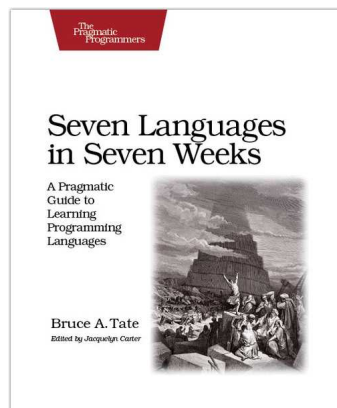
The RSpec Book

RSpec, Ruby's leading Behaviour Driven Development tool, helps you do TDD right by embracing the design and documentation aspects of TDD. It encourages readable, maintainable suites of code examples that not only test your code, they document it as well. *The RSpec Book* will teach you how to use RSpec, Cucumber, and other Ruby tools to develop truly agile software that gets you to market quickly and maintains its value as evolving market trends drive new requirements.

The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends
David Chelimsky, Dave Astels, Zach Dennis, Aslak Hellesøy, Bryan Helmkamp, Dan North
(450 pages) ISBN: 978-1-9343563-7-1. $42.95
http://pragprog.com/titles/achbd
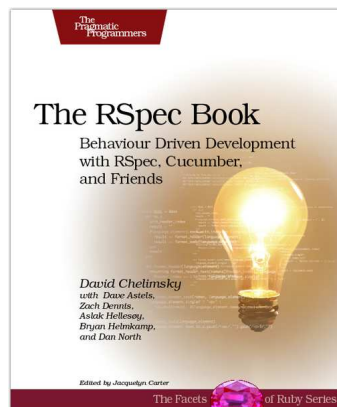
Grow Your Skills

Language Implementation Patterns

Learn to build configuration file readers, data readers, model-driven code generators, source-to-source translators, source analyzers, and interpreters. You don't need a background in computer science—ANTLR creator Terence Parr demystifies language implementation by breaking it down into the most common design patterns. Pattern by pattern, you'll learn the key skills you need to implement your own computer languages.

Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages
Terence Parr
(350 pages) ISBN: 978-1934356-45-6. $34.95
http://pragprog.com/titles/tpdsl
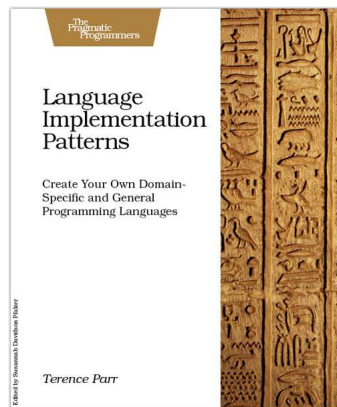
Debug It!

Debug It! will equip you with the tools, techniques, and approaches to help you tackle any bug with confidence. These secrets of professional debugging illuminate every stage of the bug life cycle, from constructing software that makes debugging easy; through bug detection, reproduction, and diagnosis; to rolling out your eventual fix. Learn better debugging whether you're writing Java or assembly language, targeting servers or embedded micro-controllers, or using agile or traditional approaches.

Debug It! Find, Repair, and Prevent Bugs in Your Code
Paul Butcher
(232 pages) ISBN: 978-1-9343562-8-9. $34.95
http://pragprog.com/titles/pbdp
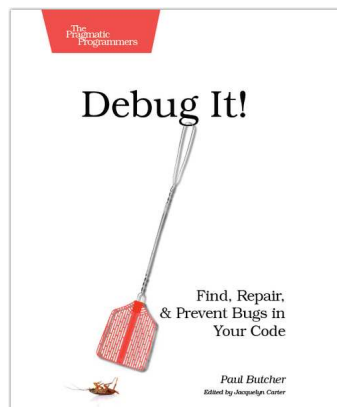
The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Home page for SQL Antipatterns
http://pragprog.com/titles/bksqla
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/bksqla.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	support@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com
Contact us:	1-800-699-PROG (+1 919 847 3884)