Extracted from:

SQL Antipatterns

Avoiding the Pitfalls of Database Programming

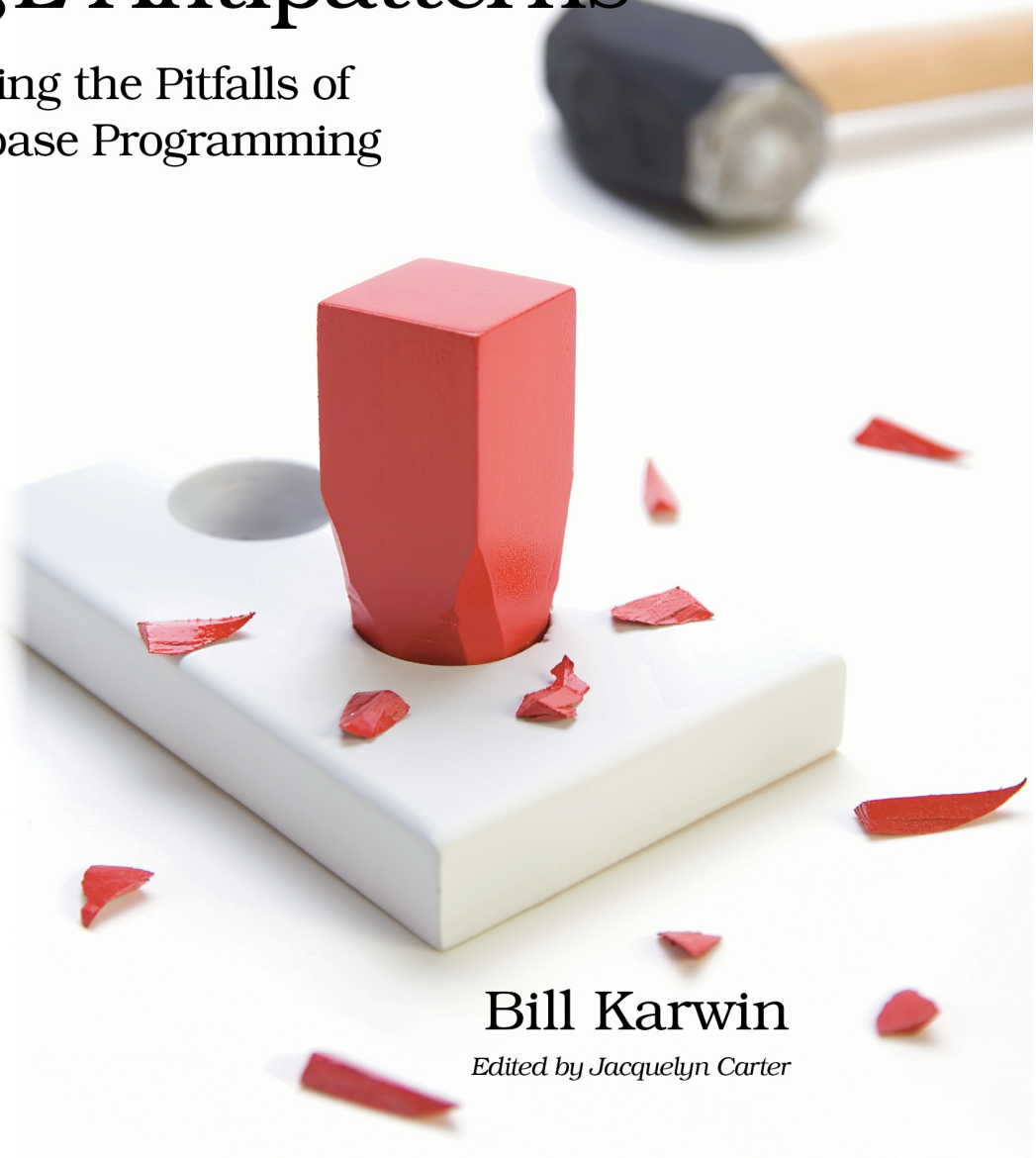
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

SQL Antipatterns

Avoiding the Pitfalls of Database Programming



Bill Karwin

Edited by Jacquelyn Carter

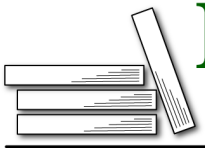
SQL Antipatterns

Avoiding the Pitfalls of Database Programming

Bill Karwin

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

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

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

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 advertisers 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 curiously, 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:

Random/anti/orderby-rand.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.

Random/anti/indexed-sort.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 *Choose Next Higher Key Value*, on page 9).

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.

Random/soln/rand-1-to-max.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.

Random/soln/next-higher.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:

Random/soln/rand-key-from-list.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.

Random/soln/limit-offset.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 MySQL, 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:

Random/soln/row_number.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 `TABLESAMPLE` clause:

Random/soln/tablesample-sql2005.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:

Random/soln/sample-oracle.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.
