

Extracted from:

# SQL Antipatterns

---

## Avoiding the Pitfalls of Database Programming

This PDF file contains pages extracted from SQL Antipatterns, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

**Note:** This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2010 The Pragmatic Programmers, LLC.

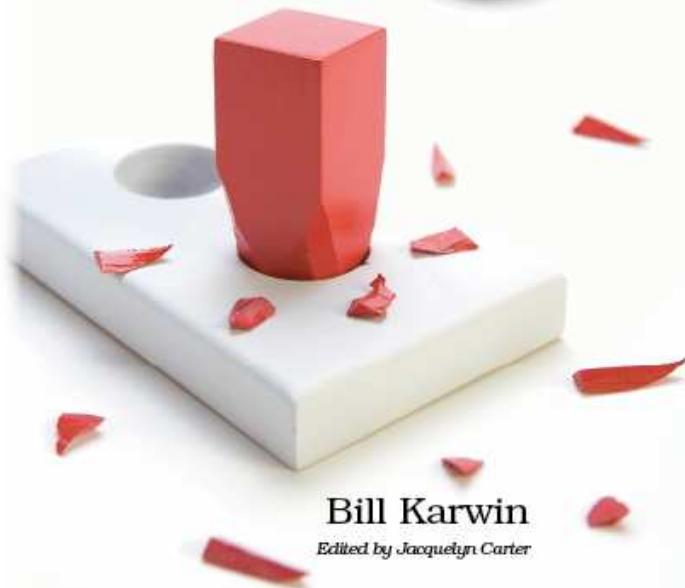
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The  
Pragmatic  
Programmers

# SQL Antipatterns

Avoiding the Pitfalls of  
Database Programming



**Bill Karwin**

*Edited by Jacquelyn Carter*

*A tree is a tree—how many more do you need to look at?*

► Ronald Reagan

## Chapter 3

# 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`

```
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 organized 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. Examples 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 countless 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.

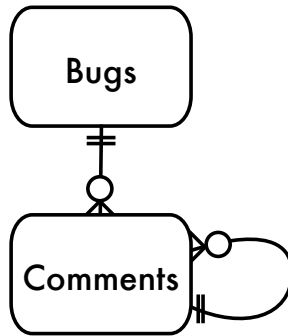
In this chapter, we'll use the threaded discussion example to show the antipattern and its solutions.

### 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.

[Download](#) Trees/anti/adjacency-list.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

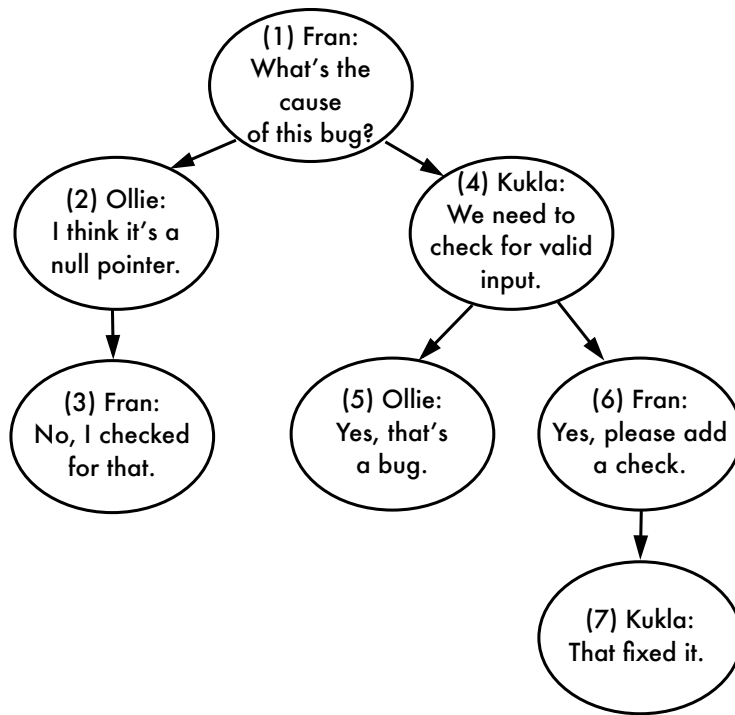
Adjacency List can be an antipattern when it's the default choice of so many developers yet it fails to be a solution for one of the most common tasks you need to do with a tree: query all descendants.

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 descendants 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`

```
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/legil/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.

# 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

---

### SQL Antipattern's Home Page

<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](http://pragprog.com/titles/BKSQLA).

## Contact Us

---

Online Orders:	<a href="http://www.pragprog.com/catalog">www.pragprog.com/catalog</a>
Customer Service:	<a href="mailto:support@pragprog.com">support@pragprog.com</a>
Non-English Versions:	<a href="mailto:translations@pragprog.com">translations@pragprog.com</a>
Pragmatic Teaching:	<a href="mailto:academic@pragprog.com">academic@pragprog.com</a>
Author Proposals:	<a href="mailto:proposals@pragprog.com">proposals@pragprog.com</a>
Contact us:	1-800-699-PROG (+1 919 847 3884)