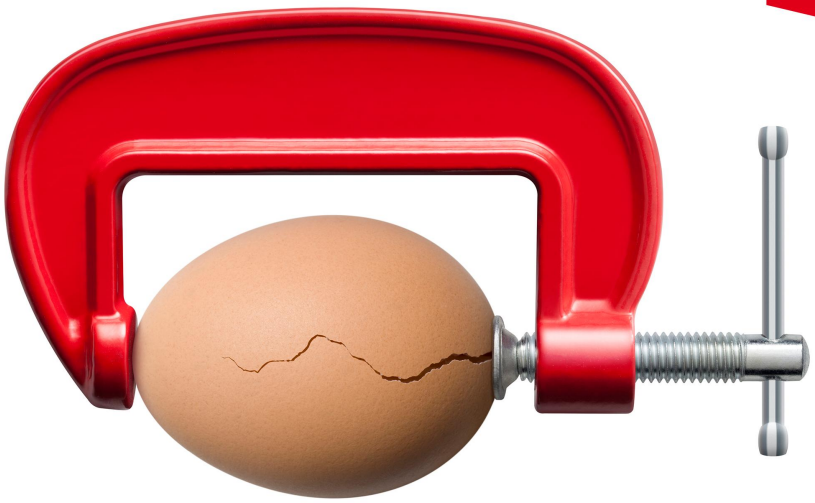
The Pragmatic Programmers

BETA

More SQL Antipatterns



Bill Karwin

edited by
Jacquelyn Carter

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Your manager Paul sends you a request. “We need a report of traffic on the site. Please display the total comments by month, in columns. And make it data-driven, so the report automatically includes future months as they get traffic.”

You know how to write a query that gives subtotals using `COUNT()` and `GROUP BY`, but that outputs in rows, not columns. How can you turn that result on its side, output in columns, like a spreadsheet? And how can you meet your boss’s requirement that the report expands to handle future months?

Objective: Turning the Table

It’s a natural request to display the results of a query in a tabular format, using as many columns as needed to make it easy to read. Effective presentation is important.

SQL query results are like the result of a function; they are raw data, which isn’t necessarily the most readable format. But you still have to present the data in the format your boss requested. They want a column for each month, and they want the report to include new columns for future months:

2022-01	2022-02	2022-03	…	2024-12	2025-01	2025-02
1234576	2048327	6060842		8675309	11041357	12345678

This type of query is called a *pivot table query* or a *crosstab query*.

Antipattern: Using a Single Query

To produce the report format your boss requested, the temptation is to develop a single query that populates “dynamic” columns (that is, additional columns are appended to the result as the query discovers new data). This is where you get stuck searching for a solution. Nothing you try as a single query works.

Wildcard in the Select-List

You know that you can use the wildcard * in an SQL select-list to populate a variable number of columns. This may sound like it’s on the right track, but it doesn’t work to add columns for each distinct value examined. The wildcard is only a shorthand for a fixed set of columns, corresponding to the columns in tables referenced in the `FROM` clause.

You might try to use `GROUP BY`, thinking that it would produce a value for each year and month. But `GROUP BY` doesn’t do what you want.

Pivot/anti/group-by.sql

```
SELECT *
FROM comment
GROUP BY TO_CHAR(created_at, 'YYYY-MM');

SELECT comment_id, comment_text, in_reply_to, user_id, created_at
FROM comment
GROUP BY TO_CHAR(created_at, 'YYYY-MM');
```

In fact, both forms of this query result in an error:

```
ERROR:  column "comment.comment_id" must appear in the GROUP BY clause
or be used in an aggregate function
LINE 1: SELECT comment_id, comment_text, in_reply_to, user_id, creat...
               ^
```

A query with GROUP BY would produce an additional *row* for each distinct value resulting from the grouping expression, not an additional column. Besides that, the values in the rest of the columns becomes ambiguous, and that's the reason for the error. See the chapter "Ambiguous Groups" in *SQL Antipatterns Volume 1 [Kar22]* for more explanation.

Correlated Subqueries In The Select-List

You can use subqueries in a select-list, such that each sub-query has a monthly subtotal. This produces the result you want, if you specify a column for each range of dates. This means you need to know the dates present in the table before you write the query. There's no way to make it expand the list of these expressions automatically as time goes on and traffic occurs in additional months. You'll have to change the query to add a new column every time the next month of traffic is added to the table.

Pivot/anti/subquery.sql

```
SELECT
  (SELECT COUNT(*) FROM comment
   WHERE created_at BETWEEN '2022-01-01' AND '2022-02-01') AS "2022-01",
  (SELECT COUNT(*) FROM comment
   WHERE created_at BETWEEN '2022-02-01' AND '2022-03-01') AS "2022-02",
  (SELECT COUNT(*) FROM comment
   WHERE created_at BETWEEN '2022-03-01' AND '2022-04-01') AS "2022-03",
  -- ...
  (SELECT COUNT(*) FROM comment
   WHERE created_at BETWEEN '2024-12-01' AND '2025-01-01') AS "2024-12",
  (SELECT COUNT(*) FROM comment
   WHERE created_at BETWEEN '2025-01-01' AND '2025-02-01') AS "2025-01",
  (SELECT COUNT(*) FROM comment
   WHERE created_at BETWEEN '2025-02-01' AND '2025-03-01') AS "2025-02";
```

Correlated subqueries in the select-list have a high cost with respect to performance, because most SQL implementations repeat the execution for each subquery.

Conditional Aggregation

Instead of using subqueries, you can use an aggregation function like COUNT(), applied to a subset of rows. This is called *conditional aggregation*. Functions like COUNT() ignore rows where the expression inside the function is NULL. If you use an expression as an argument to COUNT() that evaluates to a non-NULL value on certain rows, and NULL otherwise, then it counts the subset of rows where the expression is non-NULL.

Pivot/anti/conditional-agg.sql

```
SELECT
  COUNT(CASE WHEN created_at BETWEEN '2022-01-01' AND '2022-02-01'
    THEN 1 ELSE NULL END) AS "2022-01",
  COUNT(CASE WHEN created_at BETWEEN '2022-02-01' AND '2022-03-01'
    THEN 1 ELSE NULL END) AS "2022-02",
  COUNT(CASE WHEN created_at BETWEEN '2022-03-01' AND '2022-04-01'
    THEN 1 ELSE NULL END) AS "2022-03",
  -- ...
  COUNT(CASE WHEN created_at BETWEEN '2024-12-01' AND '2025-01-01'
    THEN 1 ELSE NULL END) AS "2024-12",
  COUNT(CASE WHEN created_at BETWEEN '2025-01-01' AND '2025-02-01'
    THEN 1 ELSE NULL END) AS "2025-01",
  COUNT(CASE WHEN created_at BETWEEN '2025-02-01' AND '2025-03-01'
    THEN 1 ELSE NULL END) AS "2025-02"
FROM comment;
```

This also requires you to spell out each column explicitly. There's no syntax to make the select-list populate more columns than those you specify.

If you are using PostgreSQL or SQLite, the FILTER clause (introduced in SQL:2003) assist conditional aggregation. This does the same thing as the previous example, but you may find it makes the code more clear.

Pivot/anti/filter.sql

```
SELECT
  COUNT(*) FILTER (WHERE created_at
    BETWEEN '2022-01-01' AND '2022-02-01') AS "2022-01",
  COUNT(*) FILTER (WHERE created_at
    BETWEEN '2022-02-01' AND '2022-03-01') AS "2022-02",
  COUNT(*) FILTER (WHERE created_at
    BETWEEN '2022-03-01' AND '2022-04-01') AS "2022-03",
  -- ...
  COUNT(*) FILTER (WHERE created_at
    BETWEEN '2024-12-01' AND '2025-01-01') AS "2024-12",
  COUNT(*) FILTER (WHERE created_at
```

```
    BETWEEN '2025-01-01' AND '2025-02-01') AS "2025-01",
  COUNT(*) FILTER (WHERE created_at
    BETWEEN '2025-02-01' AND '2025-03-01') AS "2025-02"
FROM comment;
```

As of this writing, other brands of SQL products haven't implemented the FILTER clause yet, but check the release notes, because they might implement it in a future update.

Vendor-Specific Solutions

This section is about non-standard, proprietary solutions implemented by some SQL vendors. Proprietary solutions are not in the SQL standard, so they are less likely to be adopted by other SQL products. In some cases, a proprietary feature may even be deprecated by the vendor who implemented it, if a similar standard solution becomes mainstream.

PostgreSQL developed a CROSSTAB() function to help programmers write queries that map rows to columns. The crosstab is not truly dynamic, because you must still specify the columns. The following query shows how you can use the CROSSTAB() function for the example of reporting comments per month.

Pivot/anti/postgresql-crosstab.sql

```
-- Enable the tablefunc extension before using CROSSTAB().
CREATE EXTENSION IF NOT EXISTS tablefunc;

SELECT * FROM CROSSTAB(
  'SELECT NULL, TO_CHAR(created_at, ''YYYY-MM''), COUNT(*) FROM comment
  GROUP BY TO_CHAR(created_at, ''YYYY-MM'') ORDER BY 2'
) AS ct(row_name TEXT,
  "2022-01" BIGINT, "2022-02" BIGINT, "2022-03" BIGINT, -- ...
  "2024-12" BIGINT, "2025-01" BIGINT, "2025-02" BIGINT);
```

Several other brands of RDBMS, including Microsoft SQL Server, Oracle, and Snowflake, use a PIVOT keyword. Each brand has implemented it differently.

Pivot/anti/microsoft-pivot.sql

```
SELECT 'Count' AS CommentsPerMonth,
  [2022-01], [2022-02], [2022-03], [2024-12], [2025-01], [2025-02]
FROM (
  SELECT FORMAT([created_at], 'yyyy-MM') AS ym, comment_id
  FROM comment
) AS SourceTable
PIVOT (
  COUNT(comment_id) FOR ym IN
  ([2022-01], [2022-02], [2022-03], [2024-12], [2025-01], [2025-02])
) AS PivotTable;
```

None of the solutions in this antipattern section accomplish the goal of accommodating new data dynamically, except Snowflake's implementation of PIVOT or Oracle's implementation of PIVOT XML.

How to Recognize the Antipattern

- "How can I write a query that returns a dynamic number of columns, one for each value?"

  This question may be asked in different ways, but the answer is always the same—aside from the use of wildcards, SQL cannot return a dynamic set of columns.

Legitimate Uses of the Antipattern

You can use a workaround to simulate a dynamic set of columns, by running a query that returns a single column, which formats a dynamic set of values as a JSON document:

Pivot/soln/json-objectagg.sql

```
SELECT JSONB_OBJECT_AGG(ym, count)
FROM (
  SELECT TO_CHAR(created_at, 'YYYY-MM'), COUNT(*)
  FROM comment
  GROUP BY TO_CHAR(created_at, 'YYYY-MM')
) AS t(ym, count);
```

The result of the preceding query is a single JSON column, with the year-month strings as object keys, and the counts as object values:

```
{
  "2022-01" : 1234576, "2022-02" : 2048327, "2022-03" : 6060842, ...
  "2024-12" : 8675309, "2025-01" : 11041357, "2025-02": 12345678
}
```

Your client application fetches this as a single string, not as individual columns of a query result. To separate the JSON object into a data structure the client application can use, explode the string. For example, in Python, you could use json.loads() to convert a string containing JSON into a dict.