Debugging complex SQL queries with writable CTEs

Gianni Ciolli

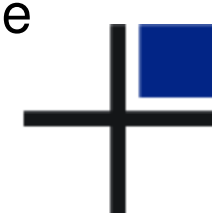
2ndQuadrant Italia

PostgreSQL Conference Europe 2011
October 18-21, Amsterdam

Outline

The problem

- We consider *SQL queries with subqueries*
  - SQL allows to write very complex queries
  - subqueries are represented by some of the vertices in the *query tree*
- The result of a query might not be what you expect
  - maybe you wrote the wrong JOIN condition
  - or you mistyped an expression
  - …
  - difficult to trace the error in the query tree
- Therefore you need to debug your query
  - `EXPLAIN` tells you the shape of the query tree, but not the contents of each node, which is what this talk is about.



Generic example 1

Minimal: only one subquery

```
SELECT ...
FROM
   (
     SELECT ...
     FROM ...
   ) a
```

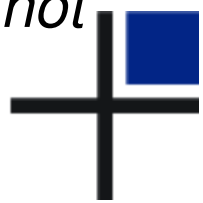
- If the output is *not* what we expect, then *where* is the error?
- SQL gives access the output of the query, but not to the output of the intermediate subquery
- We could say: an SQL query is a *black data box*
- We don't say a *black box*, because the source code is available

Generic example 2

How black is the black data box?

```
SELECT ...
FROM
  (
    SELECT ...
    FROM
      (
        SELECT ...
        FROM ...
        WHERE ...
      ) b1
    LEFT JOIN ... ON ...
  ) a1
GROUP BY ...
```

- Subqueries can be nested, combined with joins, grouped...
- Real-world problems can be represented by complicated SQL queries
- Very hard to understand why the final output is *not* what you expect



Example 3

Find all the other words with the same length

Problem

Given a list of words, to each word assign an array with all the other words having the same number of letters.

Solution (in HL)

Join the list of words with itself, creating a list of pairs of different words with the same length. Then aggregate the right side of each pair to create the array.

Recursive example 4

Greatest Common Divisor (á la Euclid)

Problem

Given two positive integers x and y, find the largest integer that divides both x and y.

Solution (Euclid of Alexandria, about 23 centuries ago)

Let z be the remainder of x when divided by y.
If $z = 0$ then y is the solution.
Otherwise replace x with y, and y with z, and repeat.

Our solution

Overview, from 9.1

- Idea
  - intercept intermediate nodes in the query tree
  - log their output to previously created *debug tables*
  - examine contents of debug tables after executing the query
- Implementation
  - 9.1: rewrite subqueries as CTEs, then add writable CTEs which contain logging statements
  - (in 8.4 or 9.0 we rewrite subqueries as CTEs, then alter them to invoke functions that contain logging statements)
- Impact
  - all in core PostgreSQL 9.1, no need to extend the server
  - no impact on the effects of the query
  - no impact on the resultset of the query
  - small impact on resource consumption, just the logging statement (the execution time will not change much)

Limitations

- Each writable CTE is executed once, therefore it cannot capture the intermediate status of the table if the query is `RECURSIVE`
- There is an impact on the original definition of the query: you need to rewrite it to add logging information; however it is easy to mark the debug code you added so that it doesn't get confused with the original code
- Correlated subqueries are not covered by this technique; they would require "circular" references in `RECURSIVE` CTEs which are unsupported at the moment (thanks to Albe Laurenz for pointing this out!)

Example 3, without CTEs

Neither efficient nor readable

```
SELECT a.word, c.arr
FROM (  SELECT word,length(word) AS n
        FROM words ) a
LEFT JOIN (
    SELECT word, array_agg(word1) AS arr
    FROM (  SELECT a.word, a1.word AS word1
            FROM (  SELECT word,length(word) AS n
                    FROM words ) a
            JOIN (  SELECT word,length(word) AS n
                    FROM words ) a1
            ON a.n = a1.n AND a.word != a1.word ) b
    GROUP BY word ) c
ON a.word = c.word;
```

Example 3, with CTEs

Efficient and more readable, but still a black data box

```
WITH a AS (
        SELECT word, length(word) AS n
        FROM words ) ,
b AS (
        SELECT a.word, a1.word AS word1
        FROM a
        JOIN a AS a1
        ON a.n = a1.n AND a.word != a1.word ) ,
c AS (
        SELECT word, array_agg(word1) AS arr
        FROM b
        GROUP BY word )
SELECT a.word, c.arr
FROM a LEFT JOIN c ON a.word = c.word;
```

Example 3, with CTEs

A *clear* data box

```
CREATE TEMPORARY TABLE debug_table
(id serial, t text, r text);

WITH a AS ( ... ) ,
debug_a AS ( INSERT INTO debug_table(t,r)
             SELECT 'a', ROW(a.*)::text FROM a ),
b AS ( ... ),
debug_b AS ( INSERT INTO debug_table(t,r)
             SELECT 'b', ROW(b.*)::text FROM b ),
c AS ( ... ),
debug_c AS ( INSERT INTO debug_table(t,r)
             SELECT 'c', ROW(c.*)::text FROM c )
SELECT a.word, c.arr
FROM a LEFT JOIN c ON a.word = c.word;
```

Example 3, with CTEs: the output

In case you were wondering...

```
   word    |                                        arr
-----------+-------------------------------------------------------------
 Alexander | {Christoph,Jean-Paul,Guillaume}
 Andreas   | {Stephen,Vincent,Michael,Dimitri}
 Bruce     | {Gavin,Simon,Peter,Steve}
 Cédric    | {Daniel,Selena,Robert,Poojan,Magnus,Stefan,Harald,Gilles,
 Christoph | {Alexander,Guillaume,Jean-Paul}
 Daniel    | {Cédric,Selena,Robert,Poojan,Magnus,Stefan,Harald,Gilles,
 Dave      | {Will,Marc,Greg,Luis}
 Dimitri   | {Michael,Andreas,Stephen,Vincent}
 Ed        |
 Gavin     | {Simon,Peter,Bruce,Steve}
 Gianni    | {Robert,Magnus,Stefan,Harald,Gilles,Daniel,Cédric,Heikki,
 Gilles    | {Selena,Heikki,Cédric,Daniel,Gianni,Harald,Stefan,Magnus,
 Greg      | {Marc,Dave,Will,Luis}
 Guillaume | {Christoph,Alexander,Jean-Paul}
 Harald    | {Gilles,Stefan,Magnus,Poojan,Selena,Robert,Heikki,Cédric,
 Heikki    | {Gianni,Gilles,Daniel,Cédric,Selena,Robert,Poojan,Magnus,
 Jean-Paul | {Guillaume,Christoph,Alexander}
```

Recursive example 4, with CTEs

Greatest Common Divisor (á la Euclid)

```
WITH RECURSIVE a(x,y) AS
    (
        VALUES (:x,:y)
    UNION ALL
        SELECT y, x % y
        FROM
            (
                SELECT *
                FROM a
                WHERE y > 0
                ORDER BY x
                LIMIT 1
            ) b
        WHERE y > 0
    )
SELECT x
FROM a
WHERE y = 0;
```

```
$ psql -v x=1547 \
       -v y=1729 \
       -f gcd-1.sql

 x
----
 91
(1 row)
```

Recursive example 4, logged

Greatest Common Divisor (á la Euclid, explained)

```
CREATE TABLE debug_table
    (
        id serial,
        x numeric,
        y numeric
    );

WITH RECURSIVE a(x,y) AS
    ( ... )
, debug_a AS (
    INSERT INTO debug_table(x,y)
    SELECT * FROM a
)
SELECT x
FROM a
WHERE y = 0;

TABLE debug_table;
```

```
$ psql -v x=1547 \
       -v y=1729 \
       -f gcd-2.sql

CREATE TABLE
 x
----
 91
(1 row)

 id |  x   |  y
----+------+------
  1 | 1547 | 1729
  2 | 1729 | 1547
  3 | 1547 |  182
  4 |  182 |   91
  5 |   91 |    0
(5 rows)
```

Before 9.1

Without writable CTEs

- CTEs were introduced in 8.4, supporting only `SELECT` (read-only)
- Before 8.4 this technique cannot be applied at all
- In 8.4 and 9.0 we can create a bespoke *logging function* which will write logging information behind the scenes, and `SELECT` it
- However, the planner assumes that your logging CTE does not modify the data, so it will skip that CTE unless it is required by other parts of the query

Example 4 on 8.4 (not working) *Joke!*

```
CREATE FUNCTION debug_func
(i_x numeric, i_y numeric)
RETURNS numeric
LANGUAGE plpgsql AS #BODY#
BEGIN
    INSERT INTO debug_table(x,y)
        VALUES (i_x,i_y);
    RETURN NULL;
END;
#BODY#;

WITH RECURSIVE a(x,y) AS
    ( ... )
, debug_a AS (
    SELECT debug_func(x,y)
    FROM a
)
SELECT x
FROM a
WHERE y = 0;
```

This doesn't work... because:

- *"the Amsterdam theme keeps changing all my $ to # (G.S.)"*

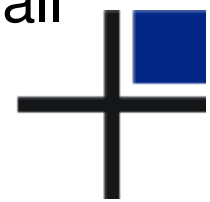
Example 4 on 8.4 (not working)

```
CREATE FUNCTION debug_func
(i_x numeric, i_y numeric)
RETURNS numeric
LANGUAGE plpgsql AS $BODY$
BEGIN
    INSERT INTO debug_table(x,y)
        VALUES (i_x,i_y);
    RETURN NULL;
END;
$BODY$;

WITH RECURSIVE a(x,y) AS
    ( ... )
, debug_a AS (
    SELECT debug_func(x,y)
    FROM a
)
SELECT x
FROM a
WHERE y = 0;
```

This doesn't work, because:

1. the contents of `debug_a` are not needed to compute the result of the query
2. PostgreSQL assumes that `debug_a` is read-only, as it should be
3. therefore there is no reason to compute it at all



Example 4 on 8.4 (hack)

It is not a part that we're proud of

- Solution: *deceive* the planner.
- That is: rewrite the query, so that the contents of `debug_a` *seem* necessary to the planner, while in fact they are not.
- **Warning 1:** deceiving the planner is bad practice. Use it responsibly, and document clearly any usage.
- **Warning 2:** you are altering the original query in a way which might not be easily undone. Make sure you copy the original version before proceeding!

Recursive example 4, on 8.4

Greatest Common Divisor (á la Euclid, explained with a hack)

```
WITH RECURSIVE a(x,y) AS
    ( ... )
, debug_a AS (
    SELECT debug_func(x,y)
    FROM a
)
SELECT x
FROM a
WHERE y = 0
AND -1 != (
    SELECT count(1)
    FROM debug_a
);
```

```
$ psql --cluster 8.4/main \
    -v x=1547 -v y=1729   \
    -f gcd-4.sql

 x
----
 91
(1 row)

 id |  x   |  y
----+------+------
  1 | 1547 | 1729
  2 | 1729 | 1547
  3 | 1547 |  182
  4 |  182 |   91
  5 |   91 |    0
(5 rows)
```

Question time

- **Any questions?**

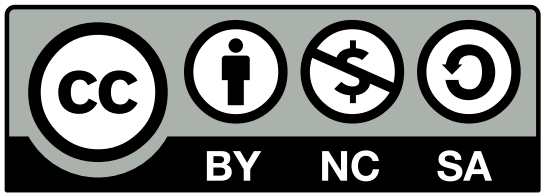
Thank you for your attention!

Feedback

`http://2011.pgconf.eu/feedback`

Licence

- This document is distributed under the **Creative Commons Attribution-Non commercial-ShareAlike 3.0 Unported** licence



- A copy of the licence is available at the URL

  `http://creativecommons.org/licenses/by-nc-sa/3.0/`

  or you can write to

  Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.