Come fare il debug di query SQL complicate usando le CTE scrivibili

Gianni Ciolli

2ndQuadrant Italia

PGDay Italiano 2011
Prato, 25 novembre

Outline

Il problema

- Consideriamo *query SQL che usano sottoquery non correlate*
  - in SQL si possono scrivere query molto complicate
  - le sottoquery sono rappresentate da alcuni dei vertici nell'*albero della query*
- Il risultato di una query può essere inatteso
  - ad esempio, può darsi che la condizione di JOIN sia stata scritta male
  - oppure c'è un errore di battitura in un'espressione
  - ...
  - è difficile localizzare l'errore nell'albero della query
- Occorre quindi fare il debug della query
  - `EXPLAIN` mostra la *forma* dell'albero della query, ma non il contenuto di ogni nodo; questo è l'argomento che tratteremo.

Esempio generico 1

Esempio minimale: una sola sottoquery

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

- Se l'output *non* è quello atteso, allora *dove* è l'errore?
- SQL mette a disposizione l'output finale della query, ma non l'output delle sottoquery intermedie
- Potremmo dire: una query SQL è una *scatola nera di dati*
- Non diciamo *scatola nera* e basta, in quanto il codice sorgente è disponibile

Esempio generico 2

Quanto è nera la scatola?

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

- Le sottoquery possono annidarsi, combinarsi con dei join, essere raggruppate...
- le query SQL che rappresentano modelli provenienti dal mondo reale possono essere arbitrariamente complicate
- è molto difficile capire per quale ragione l'output finale *non* sia quello atteso.

Esempio 3, ricorsivo

Massimo Comun Divisore (á la Euclide)

Problem

Dati due interi positivi x e y, trovare il più grande intero che divide sia x che y.

Soluzione (Euclide di Alessandria, circa 23 secoli fa)

Sia z il resto di x diviso per y.
Se $z = 0$, allora y è la soluzione.
Altrimenti ripeti dall'inizio, sostituendo x con y e y con z.

La nostra soluzione 1/2

Panoramica (dalla 9.1 in poi)

- Idea
  - intercettare i nodi intermedi nell'albero della query
  - copiare (logging) l'output dei nodi intermedi in apposite *tabelle di debug* create in precedenza
  - esaminare i contenuti delle tabelle di debug dopo aver eseguito la query
- Implementazione
  - 9.1: riscrivi le sottoquery come delle CTE, poi aggiungi ulteriori CTE scrivibili che contengono i comandi di logging
  - (8.4 o 9.0: riscriviamo le sottoquery come CTE, poi le modifichiamo leggermente in modo da garantire l'esecuzione di apposite funzioni di logging)

La nostra soluzione 2/2

Panoramica (dalla 9.1 in poi)

- Impatto
  - tutte le tecnologie richieste fanno parte di PostgreSQL 9.1 core; non occorre estendere il server
  - nessun impatto sugli effetti della query
  - nessun impatto sul risultato della query
  - impatto variabile sul consumo di risorse, dovuto essenzialmente all'esecuzione dei comandi di logging (l'impatto dipende dal quantitativo di dati ispezionati)

Limitazioni

- Ogni CTE scrivibile viene eseguita esattamente una volta; pertanto non può fotografare lo stato intermedio della tabella, in caso di query con CTE ricorsive
- C'è un impatto nella definizione originale della query: occorre aggiungere manualmente il codice che implementa il sistema di debug; tuttavia è semplice scrivere tale codice in modo che non si confonda con il codice originale della query
- Le sottoquery correlate non sono coperte da questa tecnica; sarebbe necessario disporre dei riferimenti "circolari" nelle CTE ricorsive, che al momento non sono supportati da PostgreSQL (grazie ad Albe Laurenz per avere sollevato il caso in occasione di `2011.pgconf.eu` !)

Esempio 3 ricorsivo, con le CTE

Massimo Comun Divisore (á la Euclide)

```
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)
```

Esempio 3 ricorsivo + log

Massimo Comun Divisore (á la Euclide, spiegato)

```
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)
```

PostgreSQL pre-9.1

Senza le CTE scrivibili

- Le CTE sono state introdotte in PostgreSQL 8.4, limitatamente a `SELECT` (sola lettura)
- PostgreSQL < 8.4: questa tecnica non si può usare
- Nelle versioni 8.4 e 9.0 è possibile creare delle apposite *funzioni di logging* che scriveranno le informazioni di logging dietro le scene, ed eseguirle tramite `SELECT`
- Tuttavia, il planner parte dal presupposto che tutte le CTE siano in sola lettura, e quindi pensa che non ci siano altri effetti a parte la produzione del loro output; pertanto vengono ignorate le CTE il cui output non sia utilizzato altrove nella query.

Esempio 3 su PostgreSQL 8.4 (non funzionante)

```
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;
```

Così non funziona, poiché:

1. il planner riesce a calcolare il risultato della query senza utilizzare il contenuto di `debug_a`
2. PostgreSQL parte dal presupposto che la query contenuta dentro `debug_a` sia in sola lettura, come dovrebbe essere
3. quindi non c'è ragione di eseguire la query in `debug_a`

Esempio 3 su PostgreSQL 8.4 (*hack*)

It is not a part that we're proud of

- Soluzione: *ingannare* il planner.
- Ossia: riscrivere la query in modo che l'output di `debug_a` *sembri* indispensabile al planner, pur non essendolo in realtà.
- **Avvertenza 1:** ingannare il planner è una cattiva prassi. Va usata con la massima saggezza, ed occorre documentare in dettaglio ogni utilizzo.
- **Avvertenza 2:** la query originale subirà delle modifiche che possono facilmente confondersi col testo originale. Prima di procedere, fai una copia del codice originale della query!

Esempio 3 ricorsivo, sulla 8.4

Massimo Comun Divisore (á la Euclide, spiegato, e con un *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)
```

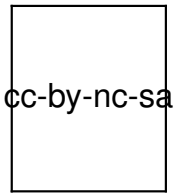
Domande

- **Ci sono domande?**

Grazie per la vostra attenzione!

Licenza

- Questo documento è distribuito secondo la licenza **Creative Commons Attribution-Non commercial-ShareAlike 3.0 Unported**



- Una copia della licenza è disponibile alla URL `http://creativecommons.org/licenses/by-nc-sa/3.0/` oppure scrivendo a *Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA*.