Explain verstehen

Hans-Jürgen Schönig

www.postgresql-support.de

Zielsetzung

EXPLAIN ...

- Was versucht uns PostgreSQL zu sagen?
- Wie kann diese Information genutzt werden?
- Wie erkenne ich Probleme?

Abfragen in PostgreSQL

Mehrstufige Ausführung

- Parser: Syntax wird überprüft
- Rewrite System: Rules, etc.
- Optimizer: Erstellung eines Plans
- Executor: Ausführung des Plans
- EXPLAIN hilft uns, dem Optimizer in die Karten zu schauen

Demo Daten


```
CREATE TABLE t_test (id serial, name text);
INSERT INTO t_test (name) SELECT 'hans'
    FROM    generate_series(1, 2000000);
INSERT INTO t_test (name) SELECT 'paul'
    FROM    generate_series(1, 2000000);
ANALYZE;
```

Ein einfacher Plan

```
test=# explain SELECT count(*) FROM t_test;
                        QUERY PLAN
-----------------------------------------------------
 Aggregate  (cost=71622.00..71622.01 rows=1 width=0)
   ->  Seq Scan on t_test
    (cost=0.00..61622.00 rows=4000000 width=0)
(2 rows)
```

- Die Tabelle wird gelesen und aggregiert

Kosten

- Kosten dienen zur Bewertung einer Query
- Kosten können nicht in Zeit umgerechnet werden
- Jede Hardware liefert die exakt selben Zahlen
- Der Optimizer ist konfigurierbar

seq_page_cost = 1

- Wie teuer ist es, Blöcke sequentiell zu lesen?
- Erhöht man diesen Wert, werden Index Scans wahrscheinlicher
- Senkt man diesen Wert, wird sequential I/O relativ gesehen billiger

random_page_cost = 4

- Auf mechanischen Platten ist Random I/O teurer als sequentielle I/O
- 4 ist "falsch":
  - Sind viele Daten gecacht, ist ein niedriger Wert besser
  - Ist nichts gecacht, ist ein wesentlich höherer Wert besser
- Im Mittel ist 4 nicht so schlecht.

cpu_tuple_cost = 0.01

- Wie teuer ist es, eine Zeile mit der CPU zu bearbeiten?
- Dieser Wert ist von der Breite der Zeile unabhängig.

cpu_operator_cost = 0.0025

- Wie teuer ist es, einen Operator oder eine Funktion aufzurufen?

cpu_index_tuple_cost = 0.005

- Wie teuer ist es, während einem Index Scan einen Index Eintrag zu bearbeiten?

Ein Beispiel (1):


```
test=# SELECT pg_relation_size('t_test') / 8192.0;
      ?column?
--------------------
 21622.000000000000
(1 row)

test=# SELECT 21622*1 + 4000000*0.01;
 ?column?
----------
 61622.00
(1 row)
```

Ein Beispiel (2):


```
test=# SELECT 61622 + 4000000*0.0025;
  ?column?
-------------
 71622.0000
(1 row)
```

Zum Vergleich

```
test=# explain SELECT count(*) FROM t_test;
                        QUERY PLAN
-----------------------------------------------------
 Aggregate  (cost=71622.00..71622.01 rows=1 width=0)
   ->  Seq Scan on t_test
    (cost=0.00..61622.00 rows=4000000 width=0)
(2 rows)
```

- Das sind exakt die Kosten aus dem Plan

Kostenparameter anpassen

- Können zur Laufzeit angepasst werden.
- “Korrekte” Parameter zu finden ist ziemlich schwierig.
- In die Kosten einzugreifen kann risky sein.

Wie liest man einen Plan?

- Immer von “innen nach außen”:

```
explain SELECT * FROM t_test
    WHERE id > (SELECT avg(id) FROM t_test);
                    QUERY PLAN
-----------------------------------------------------
 Seq Scan on t_test  (cost=71622.01..153244.01)
   Filter: ((id)::numeric > $0)
   InitPlan 1 (returns $0)
     -> Aggregate  (cost=71622.00..71622.01 rows=1)
        ->  Seq Scan on t_test t_test_1
            (cost=0.00..61622.00 rows=4000000 width=4)
(5 rows)
```

Startup Costs

- Startup Costs sagen, wie hoch die Kosten sind, bis PostgreSQL die erste Zeile liefern kann.
- Es ist quasi die zu leistende Vorarbeit.
- Für die praktische Arbeit sind total_costs (für mich persönlich) relevanter.

EXPLAIN ANALYZE (1)

- PostgreSQL führt die Query aus
- Mehr Information über die Laufzeit der Abfrage
- Ideal, um “Soll” und “Ist” zu vergleichen
- Tip: Sollte immer verwendet werden, wenn eine Abfrage langsam erscheint.

EXPLAIN ANALYZE (2)

CYBERTEC
The PostgreSQL Database Company

```
Command:     EXPLAIN
Description: show the execution plan of a statement
Syntax:
EXPLAIN [ ( option [, ...] ) ] statement
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement

where option can be one of:

    ANALYZE [ boolean ] VERBOSE [ boolean ]
    COSTS [ boolean ] BUFFERS [ boolean ]
    TIMING [ boolean ]
    FORMAT { TEXT | XML | JSON | YAML }
```

EXPLAIN ANALYZE bei der Arbeit (1)


```
test=# explain (ANALYZE true, VERBOSE true, COSTS true,
                BUFFERS true, TIMING true)
SELECT * FROM t_test ORDER BY id;

 Sort  (cost=636977.37..646977.37 rows=4000000 width=9)
    (actual time=1899.322..2306.699 rows=4000000 loops=1)
   Output: id, name
   Sort Key: t_test.id
   Sort Method: external sort  Disk: 74304kB
   Buffers: shared hit=16005 read=5620,
    temp read=9288 written=9288
```

EXPLAIN ANALYZE bei der Arbeit (2)


```
   ->  Seq Scan on public.t_test
         (cost=0.00..61622.00 rows=4000000)
        (actual time=7.469..396.204 rows=4000000 loops=1)
          Output: id, name
          Buffers: shared hit=16002 read=5620
 Planning time: 0.037 ms
 Execution time: 2530.554 ms
(10 rows)
```

Daten interpretieren

- "external sort Disk"?
  - Ist am work_mem etwas faul?
  - Fehlt ein Index?
- shared hit + read:
  - Wie gut ist die Cache Hit Rate?
  - Hohe "shared read" können bei Index Scans auf teure Random I/O hindeuten
- Stimmen die Schätzungen und die Realität überein?

Fehlschätzungen

- In komplexeren Statements können Fehlschätzungen passieren.
- Wenn der Optimizer den falschen Plan wählt, kann das zu einem Disaster führen.
- Häufige Probleme:
  - Falsch geschätze Funktion (kann weitgehend korrigiert werden)
  - Vollkommen unterschätzte Nested Loops
  - Cross Column Correlation

Ein Beispiel: Funktionen


```
test=# explain SELECT id
    FROM    generate_series(1, 10000000) AS id
    GROUP BY 1;
                        QUERY PLAN
-----------------------------------------------------------
 HashAggregate  (cost=12.50..14.50 rows=200 width=4)
   Group Key: id
   ->  Function Scan on generate_series id
    (cost=0.00..10.00 rows=1000 width=4)
(3 rows)
```

Ein Beispiel: Nested Loops


```
test=# explain analyze SELECT * FROM pg_stats;
                          QUERY PLAN
-----------------------------------------------------------------
 Nested Loop Left Join  (cost=64.28..157.15 rows=6)
      (actual time=2.814..8.408 rows=434)
  ->  Hash Join  (cost=64.15..155.45 rows=6 width=475)
      (actual time=2.532..7.241 rows=434 loops=1)
       Hash Cond: ((a.attrelid = c.oid) AND ...
       Join Filter: has_column_privilege(c.oid, a.attnum,
            'select'::text)
      ->  Seq Scan on pg_attribute a  (... rows=2519 ...)
      (actual time=0.006..4.101 rows=2519 loops=1)
```

Beispiel: Cross Column Correlation

- PostgreSQL hält Statistiken für jede Spalte
- Fragestellung:
  - 10% aller Patienten haben Hodenkrebs
  - 50% aller Patienten sind Männer
  - Wieviele männliche Patienten haben Hodenkrebs?
- Normalerweise müsste man nur die Wahrscheinlichkeiten multiplizieren.
- Die Daten sind jedoch nicht statistisch unabhängig
- Das kann böse Schätzfehler geben

Pläne und Loops (1):

- Eine einfache (ineffiziente) Abfrage:

```
test=# explain analyze SELECT *,
    (SELECT min(id)
        FROM    t_test
        WHERE t_test.id = a.id)
FROM t_test AS a LIMIT 100;
```

Pläne und Loops (2):


```
Limit (actual time=0.298..2.052 rows=100 loops=1)
 ->  Seq Scan on t_test a (... rows=100 loops=1)
     SubPlan 2
      ->  Result (... rows=1 loops=100)
          InitPlan 1 (returns $1)
          ->  Limit (... rows=1 loops=100)
            ->  Seq Scan on t_test
                (... rows=1 loops=100)
                Filter: ((id IS NOT NULL) AND (id = a.id))
                Rows Removed by Filter: 50
```

Finally . . .

Contact us ...

Cybertec Schönig & Schönig GmbH
Gröhrmühlgasse 26
A-2700 Wiener Neustadt, Austria

- More than 15 years of PostgreSQL experience:
  - Training
  - Consulting
  - 24x7 support