Eine Reise durch den PostgreSQL Optimizer

Bernd Helmle, bernd.helmle@credativ.de

11. November 2011

Am Anfang steht SQL

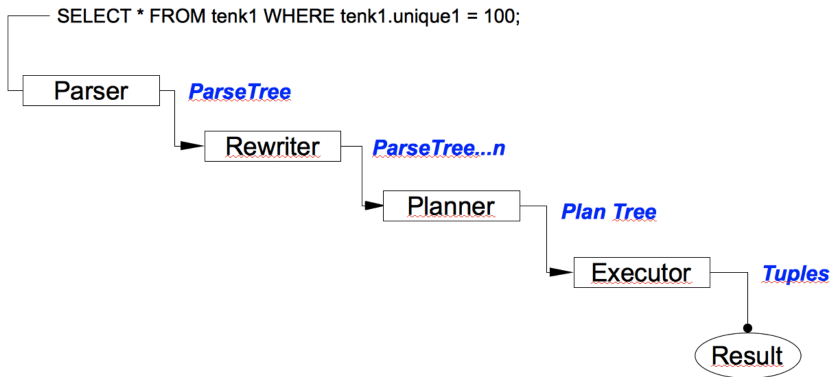
- SQL = *Structured Query Language*
- Eigentlich ein Eigenname
- Standardisiert, stetige Weiterentwicklung (SQL99, SQL 2003, SQL 2008, SQL/MED)
- Deklarativ, Beschreibend
- **KEIN(!)** Programmcode

Optimizer? Planer?

"Finde einen möglichst effizienten Weg um das gewünschte Resultat korrekt zu liefern"

- *Optimizer* ein wenig missverständlich
- Planen **und** "optimieren"
- Liefert Arbeitsanweisungen für den *Executor*
- Fasst diese in einen möglichst effizienten "Plan" zusammen
- Kritischer Faktor: Aufwand um mögliche Arbeitsschritte zu einem guten Plan zu kombinieren

Was passiert im Datenbankserver?



Der Planer (1)

- Planer erhält *Query* aus dem Rewriter
- Drei Phasen:
  1. Preprocessing Phase (Subqueries, Aggregates, WHERE-clauses)
  2. Ermitteln der JOIN-Zugriffspfade (Pfad)
  3. Zusammenfassen der Planknoten, Ausgabe als Plan
- Ein Plan ist das Ergebnis der möglichst günstigsten Kombination der einzelnen Zugriffspfade (Nodes)
- Ggf. rekursive Aufrufe des Planers (Subqueries, SubPlan)
- Unterschiedliche Methoden für Phase zwei.

Der Planer (3)

Exhaustive Search

```
SELECT *
FROM a, b, c, d
WHERE a.col = b.col AND a.col = c.col
      AND a.col = d.col;

{a,b} {a,c} {a,d}
...
-> {a,b,c} {d}
-> {d} {a,b,c}
-> {a,b} {c,d}
...
```

- Annährend "Erschöpfend"
- `from_collapse_limit`: Umschreiben von Subqueries in JOIN-Liste (Flattening)
- `join_collapse_limit`: Umsortieren von expliziten JOINs

Der Planer (4)

GEQO (Genetic Query Optimizer)

- Semi-Randomisierte Suche
- Non-deterministische Auswahl (Änderungen in 9.0)
- Gute Gene (Planknoten) werden rekombiniert

... die fittesten “Gene” gewinnen

Wie macht der Planer das?

- Heranziehen vermuteter Ergebniszeilen (`pg_class.reltuples`)
- Ggf. interpolieren auf Anzahl aktueller phyiskalischer Blöcke (`pg_class.relpages`)
- Kosten für Planknoten (bspw. Selektivität für Indexscan usw.)

. . . und nimmt den „billigsten“.

Wie weiß der Planer das?

- Statistiken, von ANALYZE gesammelt
- Tabellenspezifische Kosten (pg_class.reltuples, pg_class.relpages
- Spaltenstatistiken (pg_statistic, pg_stats)
- Verstellbare Kostenparameter

Kostenparameter

1. `seq_page_cost`: I/O-Kosten für einen sequentiellen Block-Lesevorgang
2. `random_page_cost`: I/O-Kosten für einen zufälligen Block-Lesevorgang
3. `effective_cache_size`: Beeinflusst angenomme Cachegröße und damit Auswahl von Indexscans
4. `cpu_tuple_cost`: CPU-Kosten für das Verarbeiten einer Zeile einer Tabelle
5. `cpu_index_tuple_cost`: CPU-Kosten für das Verarbeiten einer Zeile eines Indexes.
6. `cpu_operator_cost`: Operator- und/oder Prozedurkosten einer CPU.
7. `cpu_tuple_fraction`: Anteil des ersten Tupel aus einem Cursor

credativ

Kostenberechnung (1)

Die Kostenfaktoren beeinflussen die Kostenberechnung. Für einen Sequential Scan erfolgt diese beispielhaft wie folgt:

```
Endkosten = (Anzahl Blöcke * seq_page_cost)
 + (Anzahl Tupel * cpu_tuple_cost)
```

Startkosten sind 0.00, da keine Maßnahmen für das Erzeugen des Planknotens nötig sind.

Kostenberechnung (2)

Neben diesem (einfachen) I/O-Kostenmodell verfügt der Planer über weitere, tiefgreifende Statistiken

- *pg_stats* enthält mittels **ANALYZE** ermittelte Statistiken pro Tabelle und zugehörender Spalte
- `null_fraq`: Anteil NULL-Werte
- `avg_width`: Durchschnittliche Größe in Bytes
- `n_distinct`: Unterschiedliche Werte einer Spalte
- `most_common_vals`: Häufigste Werte
- `most_common_freqs`: Selektivität der häufigsten Werte
- `histogram_bounds`: Gruppierung der Spaltenwerte in ungefähr gleich große Gruppen
- `correlation`: Verteilung der Daten auf der Platte

credativ

Kostenberechnung (3)

```
SELECT * FROM pg_stats WHERE tablename = 'tenk1'
       AND attname = 'hundred';
-[ RECORD 1 ]-----+-------------------------------------
schemaname        | public
tablename         | tenk1
attname           | hundred
null_frac         | 0
avg_width         | 4
n_distinct        | 100
most_common_vals  | {41,26,64,97}
most_common_freqs | {0.0136667,0.0133333,0.0133333,0.0126667}
histogram_bounds  | {0,10,19,30,39,50,60,70,79,89,99}
correlation       | 0.0156366
```

Pläne lesen (1)

- Plan besteht aus Planknoten: jeweils eine in sich abgeschlossene Arbeitseinheit
- Die Schachtelung gibt die jeweilige Abhängigkeit wieder
- Plan erfolgt "Top-Down": Der oberste Planknoten ist der Einstiegspunkt
- Jeder Plan definiert Start-, Endkosten, geschätzte Anzahl Zeilen und die Größe einer Ergebniszeile

Pläne lesen (2)

Jeder Planknoten enthält:

- **cost = Startkosten...Endkosten**
- **rows = Geschätzte Anzahl Zeilen**
- **width = Durschn. Größe einer Zeile**

`EXPLAIN ANALYZE` fügt hinzu:

- **actual time = Startzeit...Endzeit**
- **rows = Tatsächliche Anzahl Zeilen**
- **loops = Wie oft wurde dieser Planknoten ausgeführt**

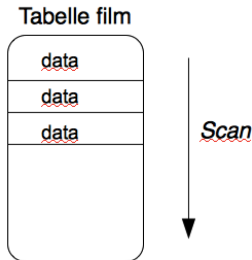
Daneben gibt es weitere Angaben wie z.B. Trigger, Hauptspeicher usw.

Algorithmen im Detail - SeqScan

Seq Scan (Sequential Table Scan):

```
EXPLAIN SELECT t1.* FROM tenk1 t1 WHERE t1.unique2 > 100;
```

```
 Seq Scan on tenk1 t1  (cost=0.00..483.00 rows=9899 width=244)
   Filter: (unique2 > 100)
```



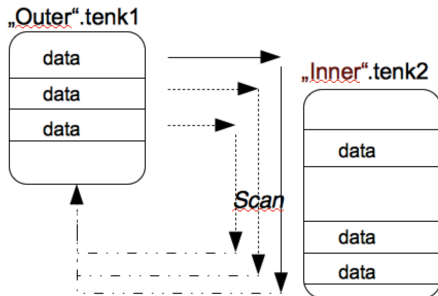
Algorithmen im Detail - Nested Loop (1)

```
EXPLAIN ANALYZE SELECT t1.* FROM tenk1 t1
                            JOIN tenk2 t2 ON (t1.unique1 = t2.unique1)
                            WHERE t2.unique2 > 100;

 Nested Loop  (cost=0.00..4795.00 rows=9899 width=244)
              (actual time=60.151..182.590 rows=9899 loops=1)
   ->  Seq Scan on tenk1 t1  (cost=0.00..458.00 rows=10000 width=244)
                             (actual time=0.010..8.628 rows=10000 loops
   ->  Index Scan using tenk2_unique1 on tenk2 t2
       (cost=0.00..0.42 rows=1 width=4)
       (actual time=0.015..0.015 rows=1 loops=10000)
         Index Cond: (t2.unique1 = t1.unique1)
         Filter: (t2.unique2 > 100)
```

Algorithmen im Detail - Nested Loop (2)

1. *"Outer"* = Äußere Tabelle, *"Inner"* = Innere Tabelle
2. Günstigste Anordnung wird durch den Optimizer bestimmt
3. Äußere Schleife einmal, für jede Zeile wird die Innere Schleife jeweils durchsucht (Aufwand n * m)
4. Durchsuchen der Tabellen anhand verschiedener Strategien (Indexscan, Seqscan usw.)



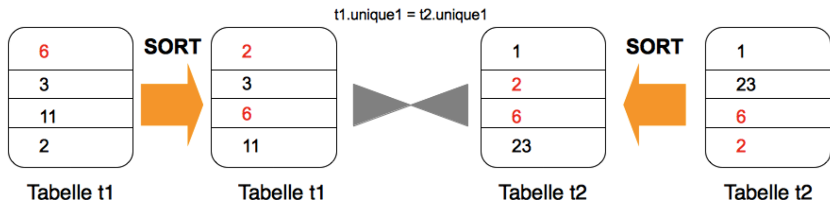
Algorithmen im Detail - Merge Join (1)

```
EXPLAIN ANALYZE SELECT t1.* FROM tenk1 t1
                                 JOIN tenk2 t2 ON (t1.unique1 = t2.unique1)
                                 WHERE t2.unique2 > 100;

 Merge Join  (cost=1126.95..3002.60 rows=9899 width=244)
             (actual time=48.168..124.313 rows=9899 loops=1)
   Merge Cond: (t1.unique1 = t2.unique1)
   ->  Index Scan using tenk1_unique1 on tenk1 t1
       (cost=0.00..1702.17rows=10000 width=244)
       (actual time=20.425..67.893 rows=10000 loops=1)
   ->  Sort  (cost=1126.95..1151.70 rows=9899 width=4)
             (actual time=27.716..30.851 rows=9899 loops=1)
         Sort Key: t2.unique1
         Sort Method:  quicksort  Memory: 926kB
         ->  Seq Scan on tenk2 t2
             (cost=0.00..470.00 rows=9899 width=4)
             (actual time=0.126..12.743 rows=9899 loops=1)
               Filter: (unique2 > 100)
```

Algorithmen im Detail - Merge Join (2)

1. Sortiere Tabelle `tenk1`, `tenk2` anhand JoinKey
2. Lese sortierte Keys `tenk1`, verknüpfe mit Keys `tenk2`
3. Ideal kombinierbar mit Indexscan
4. Tabellen müssen jeweils nur einmal gelesen werden, "Reißverschluß"



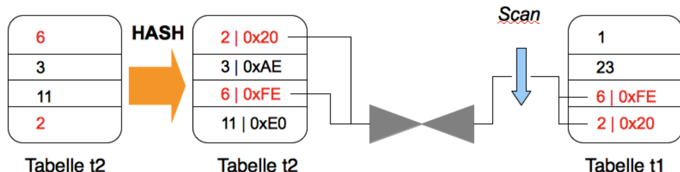
Algorithmen im Detail - Hash Join (1)

```
EXPLAIN ANALYZE SELECT t1.* FROM tenk1 t1
                          JOIN tenk2 t2 ON (t1.unique1 = t2.unique1)
                          WHERE t2.unique2 > 100;

 Hash Join  (cost=593.74..1300.73 rows=9899 width=244)
            (actual time=28.119..60.306 rows=9899 loops=1)
   Hash Cond: (t1.unique1 = t2.unique1)
   ->  Seq Scan on tenk1 t1  (cost=0.00..458.00 rows=10000 width=244)
       (actual time=0.039..8.973 rows=10000 loops=1)
   ->  Hash  (cost=470.00..470.00 rows=9899 width=4)
             (actual time=27.641..27.641 rows=9899 loops=1)
         ->  Seq Scan on tenk2 t2  (cost=0.00..470.00 rows=9899 width=4)
             (actual time=0.113..16.754 rows=9899 loops=1)
               Filter: (unique2 > 100)
```

Algorithmen im Detail - Hash Join (2)

1. Aufbauen einer Hashtabelle (aus `tenk1` oder `tenk2`)
2. Durchsuchen des Joinpartners und Vergleich auf Hashkey
3. Komplett im RAM, für größere Datenmengen
4. Teure Startup-Costs



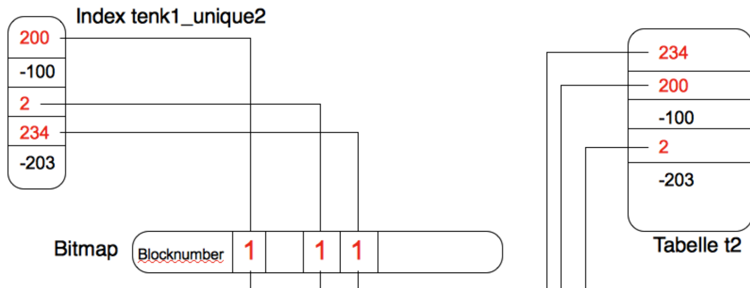
Algorithmen im Detail - Bitmap Index Scan (1)

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE unique2 IN (2, 200, 234, -100, -203);

Bitmap Heap Scan on tenk1  (cost=21.29..39.60 rows=5 width=244)
   (actual time=0.070..0.074 rows=3 loops=1)
   Recheck Cond: (unique2 = ANY ('{2,200,234,-100,-203}'::integer[]))
   ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..21.29 rows=5 width=0)
       (actual time=0.062..0.062 rows=3 loops=1)
         Index Cond: (unique2 = ANY ('{2,200,234,-100,-203}'::integer[]))
```

Algorithmen im Detail - Bitmap Index Scan (2)

1. Scanne Index, erzeuge Bitmap mit Treffern (Blocknummern)
2. Sortiere Bitmap (Blocknummern aufsteigend)
3. Scanne Tabelle anhand der Blocknummern der Bitmap aufsteigend
4. “Lossy” Bitmap (Pages statt Tuple)



Algorithmen im Detail - Index Scan (1)

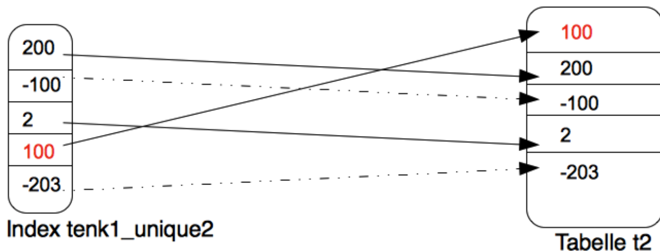
```
EXPLAIN ANALYZE SELECT * FROM tenk1 t2 WHERE unique2 = 100;

 Index Scan using tenk1_unique2 on tenk1 t2  (cost=0.00..8.27 rows=1 width=244)
   (actual time=0.022..0.023 rows=1 loops=1)
   Index Cond: (unique2 = 100)
```

credativ

Algorithmen im Detail - Index Scan (2)

1. Indexscan erfordert Sichtbarkeitsprüfung auf Tabelle pro Indexfetch
2. Dadurch relativ teuer
3. Für kleine Treffermengen geeignet



Join Removal

Optimierungsstrategie um "nutzlose" Tabelle in einem **LEFT JOIN** zu eliminieren

- Nur rechte Seite eines **LEFT JOINS**
- Keine Tupel der Tabelle in der Ergebnismenge
- Rechte Seite eindeutig (erfordert **UNIQUE** Constraint)

```
SELECT a.name
FROM a LEFT JOIN b ON (a.id = b.id)
WHERE a.name LIKE '%foo%';
```

```
 Seq Scan on a
   Filter: (name ~~ '%foo%'::text)
```

Siehe auch `http://blog.credativ.com/de/2010/03/postgresql-optimizer-bits-join-removal.html`

credativ

Semi-/Anti Joins

- Optimierungsstrategie für **EXISTS()/NOT EXISTS()**
- Berücksichtigt Schlüssel nur, sobald diese in der verknüpften Tabelle auftreten (semi) oder nicht (anti)
- Erlaubt das Abarbeiten als impliziter **JOIN**

Siehe auch `http://blog.credativ.com/de/2010/02/postgresql-optimizer-bits-semi-und-anti-joins.html`

credativ

Semi-/Anti Joins - Beispiel

```
SELECT id FROM a WHERE a.id = 200
       AND EXISTS(SELECT id FROM b WHERE a.id2 = b.id);

8.4:
 Nested Loop Semi Join  (cost=0.00..50.18 rows=6 width=4)
   ->  Seq Scan on a  (cost=0.00..24.50 rows=6 width=8)
         Filter: (id = 200)
   ->  Index Scan using b_id_idx on b  (cost=0.00..4.27 rows=1 width=4)
         Index Cond: (id = a.id2)
8.3:
 Seq Scan on a  (cost=0.00..9614.85 rows=3 width=4)
   Filter: ((id = 200) AND (subplan))
   SubPlan
     ->  Index Scan using b_id_idx on b  (cost=0.00..8.27 rows=1 width=
           Index Cond: ($0 = id)
```

SQL Inlining (1)

Unter bestimmten Umständen ist der Optimizer in der Lage, SQL-Funktionen zu "inlinen"

- Geht nur mit reinen SQL-Funktionen
- "Einfacher" SQL-Ausdruck
- Keine Seiteneffekte (**VOLATILE**)
- Darf nicht als **STRICT** deklariert sein
- SQL-Ausdruck darf Resultmenge nicht verändern
- **SECURITY DEFINER** nicht erlaubt

Beispiel...

SQL Inlining (2)

```
CREATE TABLE test
   AS SELECT a FROM generate_series(1, 10000) AS t(a);
CREATE INDEX test_id_idx ON test(a);

CREATE FUNCTION test_f()
RETURNS SETOF test VOLATILE LANGUAGE SQL
AS $$ SELECT * FROM test; $$;

EXPLAIN SELECT * FROM test_f() WHERE a = 100;

 Function Scan on test_f  (cost=0.25..12.75 rows=5 width=4)
   Filter: (a = 100)

ALTER FUNCTION test_f() STABLE;

EXPLAIN SELECT * FROM test_f() WHERE a = 100;

 Index Scan using test_id_idx on test  (cost=0.00..8.29 rows=2 width=4)
   Index Cond: (a = 100)
```

credativ

pg_stat_statements (1)

Sammelt Statistiken über ausgeführte Abfragen

```
# cd <SOURCE>/contrib/auto_explain;
# make install
# psql dbname
```

```
shared_preload_libraries = 'pg_stat_statements'
custom_variable_classes = 'pg_stat_statements'
pg_stat_statements.max = 10000
pg_stat_statements.track = all
```

```
=# CREATE EXTENSION pg_stat_statements;
```

pg_stat_statements (2)

```
#= SELECT * FROM pg_stat_statements WHERE calls > 1 ORDER BY calls DESC
-[ RECORD 4 ]-------+--------------------------------------------
userid              | 10
dbid                | 25446
query               | SELECT abalance FROM pgbench_accounts WHERE aid =
calls               | 4
total_time          | 7e-05
rows                | 4
shared_blks_hit     | 16
shared_blks_read    | 0
shared_blks_written | 0
local_blks_hit      | 0
local_blks_read     | 0
local_blks_written  | 0
temp_blks_read      | 0
temp_blks_written   | 0
```

Auto-Explain (1)

Ab PostgreSQL 8.4

- Schreibt Ausführungspläne in das PostgreSQL-Log
- Umfangreich konfigurierbar
- Erlaubt das Protokollieren von Plänen innerhalb von Prozeduren

credativ

Auto-Explain (2)

```
# cd <SOURCE>/contrib/auto_explain;
# make install
# psql dbname

#= LOAD 'auto_explain';
#= SET auto_explain.log_min_duration = '1ms';
#= SET auto_explain.log_nested_statements TO on;
```

Dankeschön

PQfinish(lecture);

Feedback:
`http://www.postgresql.eu/events/feedback/pgconfde2011`