PostgreSQL auf vielen CPUs

Hans-Jürgen Schönig

www.postgresql-support.de

Ansätze zur Skalierung

PostgreSQL auf einer CPU

- Traditionell läuft eine Query auf nur einer CPU
- Historisch gesehen war das kein Problem
- Mittlerweile ist das ein großes Problem
- Anzahl der Cores steigt stärker als die Taktfrequenz

Hard facts

- Zwischen 1996 und 2004 ist die Single-Thread Performance bei SPECint und SPECfp Benchmarks um >50% Pro Jahr gestiegen.
- Zwischen 2004 und 2012: ~21% pro Jahr.

Skalierung auf viele CPUs

- Ziel ist die Skalierung einer Abfrage auf viele CPUs
- Herausforderungen:
  - Lineare Skalierung
  - Transparenz

Die Grenzen der Skalierung

- Wie gut kann die folgende Berechnung parallelisiert werden?:

```
1 + 2 + 3 + 4
```

- 1 Thread: 3 Schritte
- Optimum: 2 Schnitte
- Parallelisierung ist nicht unendlich

Verschiedene Ansätze

- Historisch: PL/Proxy
- PostgreSQL 9.5: Cybertec “agg”
- ab PostgreSQL 9.6: Core Support

PL/Proxy (1)

- PL/Proxy ist eine Sprache für Stored Procedures, die nur dazu dient, die Abfrage zu verteilen
- Einfacher Sharding Ansatz mittels Stored Procedures

PL/Proxy (2)

- Ein Beispiel:

```
SELECT * FROM freunde('hans');
```

- PL/Proxy hasht (üblicherweise) den Input Wert und berechnet daraus den Shard
- Die Query wird am Shard ausgeführt

PL/Proxy (3)

- PL/Proxy erlaubt auch Parallelisierung:

```
SELECT sum(anzahl) FROM freunde();
```

- PL/Proxy liefert Teilsummen, um die man sich selbst kümmern muss.
- PL/Proxy ist eher “a thing of the past”

Cybertec agg (1)

- Agg läuft speziell auf PostgreSQL 9.5 und bildet die Workload einiger spezieller Kunden exakt ab.
- Es wird vermutlich keine Version für 9.6 geben
- Das Modul wird via shared_preload_libraries geladen

Cybertec agg (2)

- Was tut agg?
- Es gibt in PostgreSQL zwei wichtige Schnittstellen:
  - Customer Worker Processes
  - Custom Plan API
- agg hängt sich zwischen Optimizer und Executor und verändert den Execution Plan

Cybertec agg (3)

- Einfache Aggregate werden auf mehrere Worker Prozesse verteilt
- Teilmengen werden wieder zusammen geführt
- Es geht primär um Aggregate auf große “Fact Tables”
- Die Builtin-Aggregate müssen komplett ersetzt werden

PostgreSQL 9.6 and beyond

In Core Support

- Multi-Core Support ist eine Menge Arbeit
- Man hat über Jahre die entsprechende Infrastruktur geschaffen
  - Dynamische Worker Prozesse (9.4)
  - Dynamischer Shared Memory (9.4)
  - Shared Memory Message Queues (9.4)
  - Error Propagation (9.5)
  - Vieles mehr ...
- 9.6 ist die erste Version, die eine Query auf viele Cores nativ verteilen kann.

Die Demo Daten

- 100 Millionen Zeilen
- 3 Verschiedene Namen
- 4.2 GB Daten, voll gecacht

```
test=# \d t_small
    Table "public.t_small"
 Column |  Type   | Modifiers
--------+---------+-----------
 id     | integer |
 name   | text    |
```

Feature: Parallel Sequential Scans

- Viele Worker Prozesse können an einem Sequential Scan arbeiten
- Das ist speziell bei komplexen Filtern praktisch
- Ein Beispiel:

```
test=# SELECT * FROM t_small WHERE cos(id) = 0;
 id | name
----+------
(0 rows)

Time: 14267.196 ms
```

Parallelität erhöhen


```
test=# SET max_parallel_degree TO 4;
SET
test=# SELECT * FROM t_small WHERE cos(id) = 0;
 id | name
----+------
(0 rows)

Time: 4032.927 ms
```

Gather nodes


```
explain SELECT * FROM t_small WHERE cos(id) = 0;
                        QUERY PLAN
-----------------------------------------------------------
 Gather  (cost=1000.00..1029041.04 rows=500000 width=9)
  Workers Planned: 4
  ->  Parallel Seq Scan on t_small
        (cost=0.00..979291.04 rows=125000 width=9)
      Filter: (cos((id)::double) = '0'::double)
(4 rows)
```

Feature: Parallel Aggreation (1)


```
explain SELECT count(*) FROM t_small WHERE cos(id) = 0;
                              QUERY PLAN
-----------------------------------------------------------------------
 Finalize Aggregate  (cost=979353.96..979353.97 rows=1)
 ->  Gather  (cost=979353.54..979353.95 rows=4 width=8)
       Workers Planned: 4
       ->  Partial Aggregate  ( ... )
        ->  Parallel Seq Scan on t_small ( ... )
              Filter: (cos((id)::double) = '0'::double)
(6 rows)
```

Feature: Parallel Aggreation (2)


```
explain SELECT name, count(*) FROM t_small
WHERE   cos(id) = 0 GROUP BY name;
```

```
 Finalize GroupAggregate  (... rows=3 ...)
  Group Key: name
  -> Sort  (... rows=12 ...)
     Sort Key: name
     -> Gather  (... rows=12 ...)
        Workers Planned: 4
        -> Partial HashAggregate  (... rows=3 ...)
           Group Key: name
           -> Parallel Seq Scan on t_small  ( ... )
              Filter: (cos((id)::double) = '0'::double)
```

Entscheidungsfindung im Optimizer


```
#seq_page_cost = 1.0
#random_page_cost = 4.0
#cpu_tuple_cost = 0.01
#cpu_index_tuple_cost = 0.005
#cpu_operator_cost = 0.0025
#parallel_tuple_cost = 0.1
#parallel_setup_cost = 1000.0
```

Eigene Stored Procedures

- Um zu parallelisieren, muss eine Funktion korrekt markiert sein:
- PARALLEL
  - UNSAFE: Erzwingt serielle Ausführung
  - RESTRICTED: Parallelität ist möglich aber die Funktion darf nur vom Gruppen Leader ausgeführt werden.
  - SAFE: Volle Parallelisierung ist möglich
- UNSAFE ist der Default Wert

Ausblick

- Volle Parallelisierung ist noch viel Arbeit
- Die Infrastruktur für viele Dinge ist vorhanden
- End User müssen sich noch überlegen, wie Queries geschrieben werden

Ein Beispiel


```
test=# explain analyze SELECT name, count(*)
    FROM    t_small
    GROUP BY name;
Time: 5757.218 ms

test=# explain analyze SELECT name, count(*)
    FROM    t_small
    GROUP BY ROLLUP (name);
Time: 94873.436 ms
```

Thank you for your attention

Cybertec Schönig & Schönig GmbH
Hans-Jürgen Schönig
Gröhrmühlgasse 26
A-2700 Wiener Neustadt

www.postgresql-support.de