CYBERTEC
DATA SCIENCE & POSTGRESQL

POSTGRESQL: 5 MINUTES PERFORMANCE DIAGNOSES

HANS-JÜRGEN SCHÖNIG

CYBERTEC Worldwide

Wiener Neustadt
AUSTRIA

Tallinn
ESTONIA

Zurich
SWITZERLAND

Montevideo
URUGUAY

DATA SERVICES

- Artificial Intelligence
- Machine learning
- BIG DATA
- Business Intelligence
- Data Mining
- Etc.

office@cybertec.at

ABOUT CYBERTEC

- Inhouse Entwicklungen
- Internationales Entwicklerteam
- PostgreSQL & Zukunftstechnologien (z.B. Machine Learning, Big Data etc)
- Inhabergeführt seit dem Jahr 2000

WILLHABEN.AT®
hims
Audi
Rappi
TARGET
Auswärtiges Amt
TU WIEN TECHNISCHE UNIVERSITÄT WIEN Vienna | Austria
NOVOMATIC
TOMTOM®
NOKIA Connecting People
SIEMENS
DANONE
Lufthansa
BMW
BOSCH
Bank Austria Member of UniCredit
ncs making IT happen

INSPIRATION

INSPIRED BY
REAL WORLD EXAMPLE

- This content is inspired by a database I have seen last year
- MASSIVE DRAMA:
  - 340 billion rows
  - Oracle reached its limit
  - People tried to solve things with hardware
  - They will fail (after spending cash on Exadata) as data will grow

WHAT THEY DID

- A GUIDELINE to FAILURE:
  - Joining up to 14 tables
  - No pre-aggregation
  - No thoughts on what to query how

WHAT YOU SHOULD LEARN

- Small data sets:
  - Do basically what you want
  - Hardware is gonna bail you out
- Large data sets:
  - Stupid queries are gonna kill your
  - The more data you have, the more you have to think
- There is no “magic parameter”
  - There will NEVER BE ONE !

FAVOR REAL DATA OVER HALLUCINATIONS

WHAT HAPPENS IN THE REAL WORLD

- “We need a bigger server”
- “If we add 10 more disks, it will be faster”
- “More RAM will surely fix things”

DIAGNOSIS

- IDEAS:
  - Get real data and MEASURE
  - Drawing load graphs is (usually pointless)
  - Drawing more images does not fix queries

- ETERNAL TRUTH:
  - QUERIES cause load (not some shitty load graph)

HOW CAN WE EXTRACT REAL DATA

- Logfiles are kinda nice
  - Usually large
  - Need processing

- `pg_stat_statements` is a MUST
  - Contains all you need to fix 85% of all problems

WHAT WE GOT HERE

```
test=# \d pg_stat_statements
View "public.pg_stat_statements"
        Column          |        Type
------------------------+-------------------
Userid                  | oid
dbid                    | oid
queryid                 | bigint
query                   | text
calls                   | bigint
total_time              | double precision
```

CYBERTEC
DATA SCIENCE & POSTGRESQL

DISTRIBUTIONS DO MATTER

```
min_time         | double precision
max_time         | double precision
mean_time        | double precision
stddev_time      | double precision
rows             | bigint
```

CYBERTEC
DATA SCIENCE & POSTGRESQL

CATCHING IS RELATED TO QUERIES

```
shared_blks_hit          | bigint
shared_blks_read         | bigint
shared_blks_dirtied      | bigint
shared_blks_written      | bigint
local_blks_hit           | bigint
local_blks_read          | bigint
local_blks_dirtied       | bigint
local_blks_written       | bigint
```

I/O DOES MATTER (IF THERE IS ANY)

```
temp_blks_read     | bigint
temp_blks_written  | bigint
blk_read_time      | double precision
blk_write_time     | double precision
```

WORKING MIRACLES

```
SELECT substring(query, 1, 50) AS short_query,
       round(total_time::numeric, 2) AS total_time,
       calls, round(mean_time::numeric, 2) AS mean,
       round((100 * total_time / sum(total_time::numeric)
             OVER ())::numeric, 2) AS percentage_overall
FROM         pg_stat_statements
ORDER BY total_time DESC
LIMIT 20;
```

IT GIVES SOMETHING LIKE THIS

short_query	total_time	calls	mean	per..
UPDATE pgb....	126973.96	115832	1.10	55.64
UPDATE pgb....	96855.34	115832	0.84	42.44
UPDATE pgbenc..	2427.00	115832	0.02	1.06
SELECT abalan..	761.74	115832	0.01	0.33
INSERT INTO p..	674.12	115832	0.01	0.30
copy pgbench_..	201.51	1	201.51	0.09
CREATE EXTENS..	47.02	1	47.02	0.02
vacuum analyz..	44.25	1	44.25	0.02
alter table p..	37.82	1	37.82	0.02

INDEXING – THE FORGOTTEN WISDOM

WHAT INDEXING IS REALLY ALL ABOUT

- Missing indexes can fix 70%+ of all performance problems
- Thank you, users, for funding my winter holiday ;)

HOW CAN WE TRACK DOWN MISSING INDEXES?

- We look for . . .
  - expensive scans
  - happening often
- Do you really want to read 10 million rows 10 million times?

A MAGIC QUERY

```
SELECT schemaname, relname, seq_scan, seq_tup_read,
       idx_scan, seq_tup_read / seq_scan AS avg
FROM   pg_stat_user_tables
WHERE seq_scan > 0
ORDER BY seq_tup_read DESC;
```

OBSERVATIONS

- Usually those tables listed here will show up in pg_stat_statements too
- You will usually see:
  - Potential missing indexes
  - Pointless operations

POTENTIAL SOLUTIONS

AGGREGATES AND JOINS (1)

```
test=# CREATE TABLE t_gender (id int, name text);
CREATE TABLE
test=# INSERT INTO t_gender
      VALUES (1, 'male'), (2, 'female');
INSERT 0 2
```

AGGREGATES AND JOINS (2)

```
test=# CREATE TABLE t_person (
        id       serial,
        gender   int,
        data     char(40)
);
CREATE TABLE
```

AGGREGATES AND JOINS (3)

```
test=# INSERT INTO t_person (gender, data)
      SELECT x % 2 + 1, 'data'
      FROM   generate_series(1, 5000000) AS x;
INSERT 0 5000000
```

SIMPLE ANALYSIS

```
test=# SELECT name, count(*)
      FROM  t_gender AS a, t_person AS b
      WHERE a.id = b.gender
      GROUP BY 1;
   name | count
--------+---------
 female | 2500000
 male   | 2500000
(2 rows)
Time: 961.034 ms
```

CAN WE SPEED IT UP?

- Does anybody see a way to make this faster?
- The answer is “deep” inside the planner

LET US TRY THIS ONE

```
test=# WITH x AS
(
      SELECT gender, count(*) AS res
      FROM t_person AS a
      GROUP BY 1
)
SELECT name, res
FROM x, t_gender AS y
WHERE x.gender = y.id;
... <same result> ...
Time: 526.472 ms
```

CYBERTEC
DATA SCIENCE & POSTGRESQL

HOW DID IT HAPPEN?

- We do not understand . . .
- It must be a miracle ;)

UNDERSTANDING THE PLANNER (1)

- The answer is deep inside the planner
- Let us see what happens if we use just one CPU core:

```
test=# SET max_parallel_workers_per_gather TO 0;
SET
```

UNDERSTANDING THE PLANNER (2)

```
explain SELECT name, count(*)
      FROM t_gender AS a, t_person AS b
      WHERE a.id = b.gender GROUP BY 1;
             QUERY PLAN
---------------------------------
HashAggregate ...
 Group Key: a.name
 -> Hash Join (rows=5000034)
    Hash Cond: (b.gender = a.id)
    -> Seq Scan on t_person b (rows=5000034)
    -> Hash (cost=1.02..1.02 rows=2 width=10)
       -> Seq Scan on t_gender a (rows=2)
```

UNDERSTANDING THE PLANNER (3)

- The join is performed BEFORE the aggregation
  - Millions of lookups
- This causes the change in performance

UNDERSTANDING THE PLANNER (4)

```
test=# explain WITH x AS
(
      SELECT gender, count(*) AS res
      FROM t_person AS a
      GROUP BY 1
)
SELECT name, res
FROM   x, t_gender AS y
WHERE x.gender = y.id;
```

UNDERSTANDING THE PLANNER (5)

```
            QUERY PLAN
---------------------------------
Hash Join (rows=2)
 Hash Cond: (y.id = x.gender)
 CTE x
 -> HashAggregate (rows=2)
    Group Key: a.gender
    -> Seq Scan on t_person a
      (rows=5000034)
 -> Seq Scan on t_gender y (rows=2)
 -> Hash (rows=2)
    -> CTE Scan on x (rows=2)
```

LESSONS LEARNED

- Difference is irrelevant if your amount of data is very small
- Small things can make a difference
- Good news: An in-core fix is on the way for (maybe) PostgreSQL 12.0?

ONE MORE CLASSICAL EXAMPLE

- Processing A LOT of data
  - Suppose we have 20 years worth of data
  - 1 billion rows per year

```
SELECT sensor, count(temp)
FROM   t_sensor
WHERE  t BETWEEN '2014-01-01'
     AND '2014-12-31'
GROUP BY sensor;
```

OBSERVATIONS

- Reading 1 billion out of 20 billion rows can be slow
- A classical btree might be a nightmare too
  - A lot of random I/O
  - Size is a round 20.000.000.000 * 25 bytes
- We can do A LOT better

IDEAS

- Partition data by year
  - A sequential scan on 1 billion rows is A LOT better than using a btree
  - The planner will automatically kick out unnecessary partitions
- Alternatively:
  - Use brin indexes (Block range indexes)

AN EXAMPLE (1)

```
test=# CREATE INDEX idx_btree ON t_person (id);
CREATE INDEX
Time: 1542.177 ms (00:01.542)

test=# CREATE INDEX idx_brin ON t_person USING brin(id);
CREATE INDEX
Time: 721.838 ms
```

AN EXAMPLE (1)

```
test=# \di+
              List of relations
   Name     |  Type |    Table     | Size
------------+-------+--------------+--------
 idx_brin   | index | t_person     | 48 kB
 idx_btree  | index | t_person     | 107 MB
(2 rows)
```

CYBERTEC
DATA SCIENCE & POSTGRESQL

BRIN AT WORK

- Takes 128 blocks
  - Stores min + max value of the block
- Super small (2000 x smaller than btrees)
- Only works well when there is correlation

DOING MANY THINGS AT ONCE

CYBERTEC
DATA SCIENCE & POSTGRESQL

PASSING OVER DATA TOO OFTEN

- One source of trouble is to read data too often

- Some ideas:
  - Use grouping sets and partial aggregates
  - Use synchronous sequential scans
  - Use pre-aggregation

GROUPING SETS: DOING MORE AT ONCE

- Preparing some data

```
test=# ALTER TABLE t_person
   ADD COLUMN age int DEFAULT random()*100;
ALTER TABLE

test=# SELECT * FROM t_person LIMIT 4;
   id    |  gender |    data    | age
---------+---------+-----------+-----
5000001 |       2 | data      | 78
5000002 |       1 | data      | 26
5000003 |       2 | data      | 33
5000004 |       1 | data      | 55
```

CYBERTEC
DATA SCIENCE & POSTGRESQL

ADDING PARTIAL AGGREGATES AND ROLLUP

```
test=# SELECT name,
       count(*) AS everybody,
       count(*) FILTER (WHERE age < 50) AS young,
       count(*) FILTER (WHERE age >= 50) AS censored
FROM   t_gender AS a, t_person AS b
WHERE  a.id = b.gender
GROUP BY ROLLUP(1)
ORDER BY 1;
  name   | everybody |  young   | censored
---------+-----------+----------+----------
 female  |   2500000 | 1238156 | 1261844
 male    |   2500000 | 1238403 | 1261597
         |   5000000 | 2476559 | 2523441
```

CEO

Hans-Jürgen Schönig

MAIL hs@cybertec.at
PHONE +43 2622 930 22-2
WEB www.cybertec-postgresql.com

CYBERTEC
DATA SCIENCE & POSTGRESQL