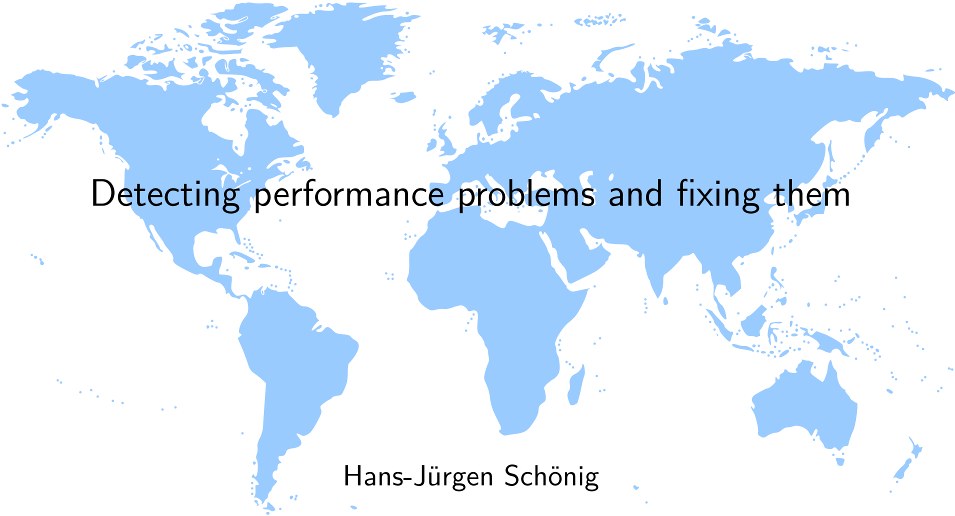

Detecting performance problems and fixing them

Hans-Jürgen Schönig

www.postgresql-support.de

About Cybertec

Cybertec: The PostgreSQL Company

- We provide
  - 24x7 support for PostgreSQL
  - professional training
  - PostgreSQL consulting
  - PostgreSQL high-availability

Scope of this training

Topics

- How to detect performance problems
- Tracking down slow queries
- Hunting I/O problems
- Optimizing postgresql.conf for speed
- Optimizing storage
- Finding slow stores procedures
- Trading durability for speed

Gather data

pg_stat_statements: A must for everybody

- Finding performance problems is impossible without data
- To stress this point:
  - No data, no improvements
- pg_stat_statements is the best way to approach the problem

Installing pg_stat_statements

- adjust postgresql.conf:

```
shared_preload_libraries = 'pg_stat_statements'
```

- restart the database
- create the extension in your database of choice

```
CREATE EXTENSION pg_stat_statements;
```

Finding relevance (1)

- NEVER EVER read pg_stat_statements without ordering the table
- People tend to get stuck and inspect irrelevant data
- ORDER BY:
  - total_time DESC
  - calls DESC
  - I/O timing, etc.
- Consider building a view

http://www.cybertec.at/2015/10/pg_stat_statements-the-way-i-like-it/

Finding relevance (2)

- Work top down
- The 10th row can by definition not contribute more than 10% to the overall problem
- Try to reduce the amount of information you are looking at

Configuring pg_stat_statements (1)

- Queries might be cut off
  - Especially relevant to Java developers
  - ORMs tend to produce insanely long SQL statements
- Raise ...

```
track_activity_query_size = 1024
```

Configuring pg_stat_statements (2)

- pg_stat_statements.max: How many statements are tracked?
- pg_stat_statements.save: Specifies whether to save statement statistics across server shutdowns.

Resetting data

```
SELECT pg_stat_statements_reset();
```

pg_stat_statements: Overhead

- Use at least PostgreSQL 9.2
- Older versions are basically not usable
- Overhead of new versions can be seen as “noise”
- The module can always be activated
- Having no data is ways more expensive

Inspecting disk wait

- CPU consumption can be measured easily
- By default pg_stat_statements does not show disk wait
- Consider activating track_io_timing
  - It is off by default
  - Overhead can be substantial

Use pg_test_timing

```
iMac:~ hs$ pg_test_timing
Testing timing overhead for 3 seconds.
Per loop time including overhead: 38.28 nsec
Histogram of timing durations:
< usec   % of total      count
     1     96.21594   75400031
     2      3.78017    2962348
     4      0.00016        123
```

pg_test_timing: Considerations

- Values between 14 and 1900 nsec have been observed
- Virtualization can be a speed killer
- Consider: Time has to be checked millions of times
- When track_io_timing is on blk_read_time and blk_write_time will contain data.
- I/O time can be compared to CPU time in this case

Inspecting queries

Inspecting execution plans

- Once you have identified slow queries, check out how PostgreSQL handles them
- EXPLAIN ANALYZE is the key to success
- Try to figure out, where time is burned
- Check for errors in those estimates

Using EXPLAIN

```
test=# \h EXPLAIN
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 }
```

Check for missing indexes

- Before you look at
  - filesystem
  - RAID level
  - memory
  - etc
- CHECK FOR MISSING INDEXES !!!
  - will solve at least 60% of all common performance problems

A word of caution

- There is no such thing as "almost correct indexing"
- A single missing index can turn a perfect system in a nightmare.
- Doing too much too often
  - No way to fix that with hardware

Finding missing indexes

- Use our "miracle query":

```
SELECT  schemaname, relname, seq_scan, seq_tup_read,
    seq_tup_read / seq_scan AS ratio, idx_scan
FROM    pg_stat_user_tables
WHERE   seq_scan > 0
ORDER BY seq_tup_read DESC
LIMIT 15;
```

What to look for?

- Look for large tables scanned too often
- There will always be sequential scans
  - Small tables
  - Backups
  - Analytics
  - etc.
- Reading large tables too often is poison
- Reasonable sequential scans are perfectly fine

A classical example

- Can you see the problem?

```
CREATE TABLE t_user
(
    id          serial PRIMARY key,
    username    text,
    passwd      text
);
CREATE TABLE
```

Prevent “over indexing”

- Too many indexes will slow down writes
- Indexing everything is also bad
- Look for indexes, which are never used

Finding pointless indexes

```
SELECT  relname, indexrelname, idx_scan,
    pg_relation_size(indexrelid),
    sum(pg_relation_size(indexrelid))
        OVER (ORDER BY idx_scan, indexrelid)
FROM    pg_stat_user_indexes
ORDER BY 3;
```

- Real work problem seen at a client: 74% of disk space was occupied by indexes, which were NEVER used (= 0 times)
- It ruined performance completely

Looking for wrong estimates

```
test=# explain (analyze true, verbose true)
    SELECT count(*) FROM generate_series(1, 1000000);
                        QUERY PLAN
---------------------------------------------------------
 Aggregate  (cost=12.50..12.51 rows=1 width=8)
    (actual time=233.053..233.053 rows=1 loops=1)
  Output: count(*)
  ->  Function Scan on generate_series
    (cost=0.00..10.00 rows=1000)
    (actual time=84.598..174.208 rows=1000000 loops=1)
        Function Call: generate_series(1, 1000000)
(7 rows)
```

What we see

- Estimates are ways off (due to the function call)
- PostgreSQL can (in most cases) not look into functions
- Various execution times are shown
- HINT: Try to see, where time “jumps”

Startup costs. vs. total costs

- Startup costs: When does this node yield the first row.
- Total costs: How long does it take to produce the last row.

Adjusting costs

- The cost model can be adjusted
- PostgreSQL provides various parameters to do that
- However, it requires a deep understanding of what is going on
- Be aware of side effects

Best parameters to adjust I/O costs

- seq_page_cost: Adjust costs of sequential reads
- random_page_cost: Cost to read a random block
  - On SSDs the default can be changed from 4 to 1 to adjust for low seek times
- effective_cache_size: Tells the optimizer about how much RAM to expect
- Keep in mind: The consequences can be “unexpected” in some cases

Nested loops: My favorite pitfall (1)

- What a nested loop does

```
for x in a:
    for y in b:
        if  x == y:
            yield row
```

Nested loops: My favorite pitfall (2)

- What happens if the nested loop is underestimated?
- What are the symptoms?
  - CPU will go through the roof
  - Runtime might vary A LOT depending on input parameters
  - Changes might occur sudden
- Try and see what changes:

```
SET enable_nestloop TO off;
```

- This is pretty advanced already

Exercise:

- Run the following query and tell me where it might go wrong
- Where is the planner wrong?
- Is it a problem or not?

```
SELECT * FROM pg_stats;
```

Exercise: Remarks ...

- There are no statistics for the output of functions
- PostgreSQL has to make static assumptions
- This can fool the planner
- Wrong estimates can lead to unexpected stuff.

Function statistics (1)

```
test=# explain SELECT *
    FROM generate_series(1, 10000000) AS x WHERE x < 0;
                  QUERY PLAN
--------------------------------------------
 Function Scan on generate_series x
    (cost=0.00..12.50 rows=333 width=4)
   Filter: (x < 0)
```

Function statistics (2)

```
test=# explain SELECT *
    FROM generate_series(1, 10000000) AS x WHERE x > 0;
               QUERY PLAN
--------------------------------------------
 Function Scan on generate_series x
    (cost=0.00..12.50 rows=333 width=4)
   Filter: (x > 0)
```

explain (buffers true, ...)

- "buffers true" will tell us about the I/O behavior of the query
- You can also see that in pg_stat_statements
- The more buffers the query needs, the more runtime might fluctuate (a query can easily be 100x slower or faster)
- Unstable runtime is often caused by unpredictable caching
- Low cache rates are not as bad for sequential scans as for index scans
- Watch out for random I/O (= most expensive)

Buffers: Side note

- Keep in mind that a cache miss does not necessarily mean "disk hits"
- The filesystem cache is there for you
- Try to calculate average costs of a block to get an idea of how "random" you are.

Hidden "performance" problems (1)

What is potentially wrong in the following example?

```
CREATE TABLE a (aid int);
CREATE TABLE b (bid int);

INSERT INTO a VALUES (1), (2), (3);
INSERT INTO b VALUES (2), (3), (4);
```

Hidden “performance” problems (2)

- How many rows do you expect?

```
test=# SELECT *
    FROM a LEFT JOIN b
        ON (aid = bid AND bid = 2);
 aid | bid
-----+-----
   1 |
   2 |   2
   3 |
(3 rows)
```

Hidden “performance” problems (3)

- This one comes hidden as performance problem but in fact it is a logical problem
- Aggregates might hide the logical mistake.
- Watch out for outer joins
- Most people do NOT know how to write them properly
- Expect semantic errors (double check !)

Obvious stuff: Unfortunately relevant

- Go to www.google.com and search for “USA”
- You will find millions of hits
  - Google will tell you so
  - However, not more than around 30 pages are really available
- Why does everybody else want an exact count?
- Exact counting in search forms is a MAJOR performance problem
  - Yes, this is an obvious thing

I/O bottlenecks

A simple example:

- 10.000 INSERT statements (single transactions)

```
CREATE TABLE t_test (id int);

INSERT INTO t_test VALUES (1);
INSERT INTO t_test VALUES (1);
...
INSERT INTO t_test VALUES (1);
```

COMMIT: The impact of disk flushes

- Depending on your hardware runtime might fluctuate
  - we have seen 1 sec - 3 minutes 50 seconds
- Remember: On COMMIT PostgreSQL has to flush to disk
- CPU is not the problem here
- Copying large files to measure throughput is pointless

Testing disk flushes: pg_test_fsync

- Can help to see, how your I/O system behaves
- Easy to use

```
iMac:~ hs$ pg_test_fsync --help
Usage: pg_test_fsync [-f FILENAME] [-s SECS-PER-TEST]
```

Trading speed for durability

- synchronous_commit tells PostgreSQL what to do on COMMIT
  - on: Flush EVERY transaction to disk. No data is lost after a crash. Major performance bottleneck for small transactions.
  - off: Introduces a potential window of data loss (= 3 x wal_writer_delay). This is fine for many applications. Pointless if your transactions are large.
- commit_delay: Tells a session to wait hoping that other transactions will commit at the same time. The idea is to save on disk flushes

commit_delay: Grouping COMMITs

- Wait for a couple of microseconds before really flushing the WAL
- It is a synchronous method to commit transactions
- Makes sense if you have many concurrent transactions

commit_siblings: Ensuring concurrency

- Minimum number of concurrent open transactions required before using the commit_delay setting.
- Feel free to experiment.
- It can be hard to set this parameter “correctly” as concurrency and transaction sizes might vary.

Reducing I/O: Transaction log bypasses

- There are ways to bypass the PostgreSQL transaction log in some cases
- Two major features:
  - CREATE UNLOGGED TABLE
  - Specially designed transactions

CREATE UNLOGGED TABLE

- Ideal for staging table
- The table can be used just like any other table
- Will reduce I/O significantly (no WAL written)
- Downsides:
  - Table is guaranteed to be empty in case the server crashes

Designing transactions for the WAL bypass

```
BEGIN;
CREATE TABLE case1 (....);
COPY case1 FROM ...
COMMIT;

BEGIN;
TRUNCATE case2 ...
COPY case2 FROM ...;
COMMIT;
```

WAL-bypass: case1

- Flush the table at the end of the transaction
- Nobody but you can see the table
  - Concurrency is not the issue here
  - Just flush the data file

WAL-bypass: case2

- TRUNCATE puts a file in quarantine
  - Data file will be removed from disk on commit
  - Data file will be needed on rollback
- TRUNCATE creates a table lock
  - No concurrency
- At the end:
  - Flush the new table file or
  - Take the old data file
- There is never a case requiring files to be repaired using the WAL

Using tablespaces

- Adding tablespaces only makes sense if you add hardware to the system.
- Having 10 tablespaces on the same disk is pointless
- Splitting up WAL, data, and indexes can make sense
  - Same rules apply for most database systems

- Tablespaces will be even more useful as parallel queries take off
- Scaling up I/O can be a major issue in analytics

Choosing a filesystem

- XFS and ext4 are absolutely fine.
- Avoid COW (= Copy-On-Write) filesystems
  - btfs and alike are NOT GOOD for database work
  - I/O tends to be too random
- EXPERIENCE: Before blaming the filesystem, consider fixing indexing

Improving the I/O configuration

- PostgreSQL cannot keep the transaction log forever
- At some point it has to be recycled
- To recycle WAL PostgreSQL has to ensure that data has safely reached the data files:
  - So called "checkpoints" are here to ensure that
- Checkpoint distances, etc. can be configured

Checkpoint settings (1)

- min_wal_size: If WAL disk usage is below this value, old WAL is recycled and not removed.
- max_wal_size: Maximum size of the WAL (soft limit). This can have an impact on recovery times after a crash
- checkpoint_timeout: Maximum time between two checkpoints

Checkpoint settings (2)

- checkpoint_warning: Tells us that the server checkpoints too frequently.
  - Checkpointing too often is not risky
  - It "only" leads to bad performance
  - Can happen during bulk loading, etc.
- checkpoint_completion_target: Fraction of total time between checkpoints.
  - Do you want short or long checkpoints?
  - It is mostly about flattening I/O spikes

General advise

- Checkpointing too frequently is bad for performance
- Long checkpoint distances lead to longer recovery on startup
- If you have a LOT of memory:
  - Adjust kernel settings (vm.dirty_ratio, etc.)
- PostgreSQL 9.6 has some onboard means to handle a lot with onboard variables
  - Older versions need some more adjustments on the kernel side

Linux kernel settings (1):

- vm.dirty_background_ratio: is the percentage of system memory that can be filled with “dirty” pages before pdflush, etc. kick in
- vm.dirty_ratio: is the absolute maximum amount of system memory that can be filled with dirty pages before everything must get committed to disk
- vm.dirty_background_bytes and vm.dirty_bytes are another way to specify these parameters. If you set the *bytes version the* ratio version will become 0, and vice-versa.

Linux kernel settings (2):

- vm.dirty_expire_centisecs is how long (3000 = 30 seconds) data can be in cache before it has to be written. When the pdflush/flush/kdmflush processes kick in they will check to see how old a dirty page is, and if it's older than this value it'll be written asynchronously to disk.

Adjusting memory parameters

Adjusting memory parameters:

- The following memory parameters will need a review:
  - shared_buffers
  - work_mem
  - maintenance_work_mem
  - wal_buffers
  - temp_buffers
  - effective_cache_size

shared_buffers: The PostgreSQL I/O cache

- Memory is allocated on startup and never resized
- 25 - 40% of system memory
- Correct value depends on workload
- Use larger values in case of large checkpoint distances
- PostgreSQL relies on filesystem caching as well
- HINT: DO NOT use too much
  - High values can turn against you
  - Rule of thumb: max. 8-16GB

work_mem

- Used by operations such as sorting, grouping, etc.
- Will be allocated on demand (per operation)
- It is a local value - not a global one

work_mem in action (1):

```
CREATE TABLE t_test (id serial, name text);
INSERT INTO t_test (name)
    SELECT 'hans' FROM generate_series(1, 100000);
INSERT INTO t_test (name)
    SELECT 'paul' FROM generate_series(1, 100000);
ANALYZE;
```

work_mem in action (2):

```
test=# explain SELECT name, count(*)
    FROM    t_test GROUP BY 1;
                  QUERY PLAN
-------------------------------------------------------
 HashAggregate  (... rows=2 width=13)
   Group Key: name
   ->  Seq Scan on t_test  (... rows=200000 width=5)
(3 rows)
```

work_mem in action (3):

```
test=# explain SELECT id, count(*)
    FROM    t_test GROUP BY 1;
                  QUERY PLAN
--------------------------------------------
 GroupAggregate  (... rows=200000 width=12)
   Group Key: id
   ->  Sort  (.. rows=200000 width=4)
         Sort Key: id
         ->  Seq Scan on t_test
         (... rows=200000 width=4)
(5 rows)
```

work_mem in action (4):

```
SET work_mem TO '1 GB';
explain SELECT id, count(*) FROM t_test GROUP BY 1;
                    QUERY PLAN
----------------------------------------------------
 HashAggregate  (... rows=200000 width=12)
   Group Key: id
   ->  Seq Scan on t_test  (... rows=200000 width=4)
(3 rows)
```

work_mem in action (5):

- work_mem will also speed up sorting
- if data does not fit into work_mem, PostgreSQL has to go to disk, which is expensive

maintenance_work_mem

- The same as work_mem but used for administrative tasks such as
  - CREATE INDEX
  - ALTER TABLE
  - VACUUM
  - etc.

temp_buffers

- Sets the maximum number of temporary buffers used by each database session.
- These are session-local buffers used only for access to temporary tables.

wal_buffers

- Usually auto-tuned
- Defines the amount of shared memory used to store unwritten WAL
- Historically this was 64kb, which caused SERIOUS performance issues
- Consider using 16MB+
- Extremely large values are not beneficial, however (but not counterproductive either)

effective_cache_size

- The operating system caches too
- Telling PostgreSQL about the total amount of memory in your system makes sense
- I/O estimates for index pages can be adjusted by the planner
- Rule of thumb: 70% of total RAM

Stored procedures

Tracking down slow stored procedures

- Stored procedures can be a major performance bottleneck
- Fortunately PostgreSQL provides us with runtime statistics
- Consider setting track_function = 'all' in postgresql.conf

Inspecting pg_stat_user_functions

```
test=# \d pg_stat_user_functions
 View "pg_catalog.pg_stat_user_functions"
   Column    |       Type        | Modifiers
------------+-------------------+-----------
 funcid     | oid               |
 schemaname | name              |
 funcname   | name              |
 calls      | bigint            |
 total_time | double precision  |
 self_time  | double precision  |
```

total_time vs. self_time

- A function can have high total_time but low self_time
  - Usually wrapper functions
- Focus more on self_time
  - Time is really lost in those functions

Frequent causes of slow functions

- By default functions are VOLATILE
  - Functions might be called too often in this case
- Consider using:
  - STABLE: The function will return the same value given the same parameters inside a transaction.
  - IMMUTABLE: The function will always return the same value given the same input regardless of the transaction
- Examples: random() is volatile, now() is “stable”, pi() and cos(x) are “immutable”

Impacts on performance

```
SELECT * FROM tab WHERE value = random();
SELECT * FROM tab WHERE value = now();
SELECT * FROM tab WHERE value = pi();
```

- The first query CANNOT use indexes
  - Can be a MAJOR performance bottleneck (!)
- The 2nd and 3rd query can

Contact

Contact data

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

Website: www.postgresql-support.de

Follow us on Twitter: @PostgresSupport, postgresql_007